# Exercises — General Tree LCRS

version **#7be580532266ed398481e31366afcc24b1950c2a**


**Assistants C/Unix 2023** <assistants@tickets.assistants.epita.fr>

## Copyright



## Contents



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

**File Tree**

```
general_tree_lcrs/
├── Makefile  (to submit)
├── gtree.c  (to submit)
├── gtree.h
├── print.c  (to submit)
├── print.h
├── serialize.c  (to submit)
└── serialize.h
```

**Makefile**

- library: Produce the libgtree.a archive
- clean: Delete everything produced by make

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

- malloc(3)
- calloc(3)
- free(3)
- fprintf(3)
- fopen(3)
- fclose(3)
- fgets(3)
- sscanf(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

A general tree is a recursive arborescent datastructure with a set of nodes. Like binary tree, the first node is called the **root**, but each node contains a list of children, not only 2.

You are going to re-implement a general tree, but using left child right sibling representation, with this representation, you will emulate a general tree with a binary tree. For a node, his left will be his first child and his right will be his sibling.

In order to simplify the implementation, this general tree will not support multiple nodes containing the same data.

The structure used for this exercise is the following:

```
struct gtree
{
    char data;
    struct gtree *child;
    struct gtree *sibling;
};
```

Notes:

- Manage the memory wisely.
- Remember left child right sibling is just a representation.

The following figures will be used for examples.

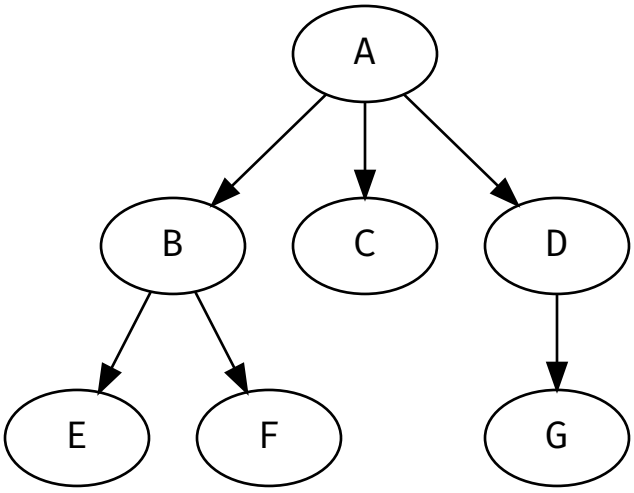


Fig. 1: General tree dynamic n-uplet representation

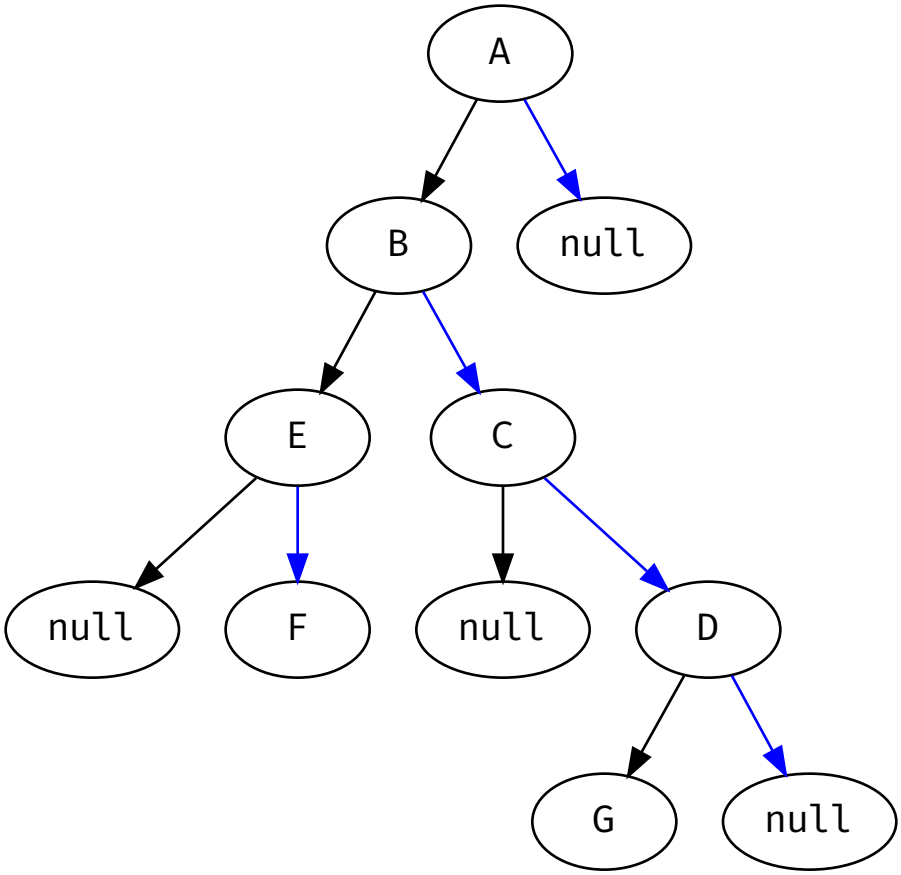


Fig. 2: General tree dynamic left child right siblings representation

## 2 Makefile

Your makefile must define at least the following targets:

- **library**: Produce the *libgtree.a* library at the root of the exercise directory. This must be the first target.
- **clean**: Delete everything produced by make.

## 3 Gtree

In this part, you will implement basic functions for tree manipulation.

Write the following functions:

```
struct gtree *gtree_create_node(char data);
```

This function creates and returns a pointer to a new node. If `malloc` failed, returns `NULL`.

```
int gtree_add_child(struct gtree *parent, struct gtree *child);
```

This function adds the child `child` to the node `parent`. If `parent` and/or `child` are `NULL`, returns 0, 1 otherwise.

```
struct gtree *gtree_search_node(struct gtree *root, const char data);
```

This function searches in the tree of root `root` the node containing `data` and returns it. If this node does not exist, returns `NULL`.

```
void gtree_free(struct gtree *root);
```

This function frees all the tree of root `root`.

```
int gtree_del_node(struct gtree *root, const char data);
```

This function searches in the tree of root `root` the node containing `data`. If this node exists and is not the `root`, the function removes it and all its sub-tree and returns 1. Be careful, you should destroy its children but not its sigblings. If this node is the `root`, the function does nothing and returns -1. Otherwise, the function returns 0.

## 4 Print

Write the following function:

```
void gtree_print_preorder(const struct gtree *root);
```

This functions displays on the standard output all the elements of the tree of root `root` with a pre-order depth first traversal, separated by spaces.

The output format is:

```
42sh$: ./print_prefix | cat -e
A B E F C D G$
```

## 5 Dot

Write the following functions:

```c
int gtree_serialize(const char *filename, const struct gtree *root);
```

This function writes to the file `filename` the tree of root `root` represented using the `DOT` format. The nodes are indexed by their values. We ask you to do a depth-first traversal of the tree, and print all edges connecting the nodes before recursing. If you cannot open the file for writing, return -1, 0 otherwise. If the file `filename` does not exist, you must create it. If it does, you must override its content.

Example:

```
digraph tree {
A -> B;
A -> C;
A -> D;
B -> E;
B -> F;
D -> G;
}
```

You can then visualize the tree with the command:

```
42sh$ dot -Tpng output.dot -o output.png
```

```c
struct gtree *gtree_deserialize(const char *filename);
```

This function reads a previously serialized gtree from file `filename` and returns a pointer to the corresponding `gtree`. If the tree is empty or the file cannot be opened for reading, returns `NULL`.

Notes:

- You do not have to handle invalid files.
- Input and output formats are exactly the same.

A empty tree is represented like this:

```
42sh$ cat -e empty.dot
digraph tree {$
}$
```

A tree containing only one node is represented like this:

```
42sh$ cat -e empty.dot
digraph tree {$
```

(continues on next page)

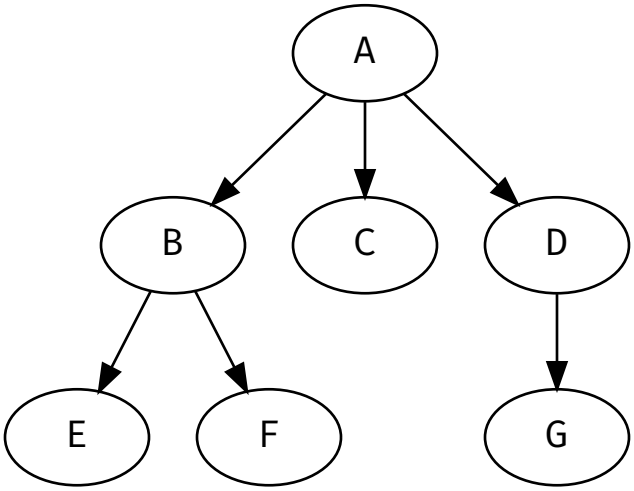


Fig. 3: output.png

(continued from previous page)

```
VALUE1;$
}$
```

An edge of the tree is represented like this:

```
42sh$ cat -e body.dot
VALUE1 -> VALUE2;$
```

VALUE1 and VALUE2 are a single character and match the regex [a-zA-Z].

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