DM - Greffes génétiquement modifiées d'un arbre binaire

Dans ce DM, on se propose d'implémenter une greffe génétiquement modifiée d'arbres binaires. Pour vérifier l'opération, on mettra au point un langage de description des arbres binaires afin de pouvoir sauvegarder et charger en mémoire un tel arbre binaire.

1 Introduction

La formule logique $\Big((P \wedge Q) \Longrightarrow R\Big) \vee S \Longrightarrow S$ peut s'écrire en utilisant une forme développée de l'implication : $U \Longrightarrow V \equiv (\neg U) \vee V$.

En notant A l'arbre syntaxique de $\Big((P \wedge Q) \Longrightarrow R\Big) \vee S \Longrightarrow S$ et B l'arbre syntaxique de $(\neg U) \vee V$ (voir Figure 1a et 1b), on peut réaliser une *greffe* de l'arbre B sur l'arbre A sur chaque nœud `n` de A ayant comme étiquette $\Longrightarrow$. Le sous-arbre gauche de `n` remplacera la feuille U de B ; le sous-arbre droit de `n` remplacera la feuille V de B. Il s'agit en réalité de la substitution des nœuds contenant $\Longrightarrow$ par un autre arbre.

L'arbre obtenu après la greffe est représenté Figure 1c.

2 Partie 1 : Greffe génétiquement modifiée

Dans ce devoir, nous allons nous intéresser à une greffe légèrement différente de celle des formules logiques.

Nous ne considérerons que des arbres binaires dont l'étiquette peut être une chaine de caractères. Ainsi, nous utiliserons la structure suivante :

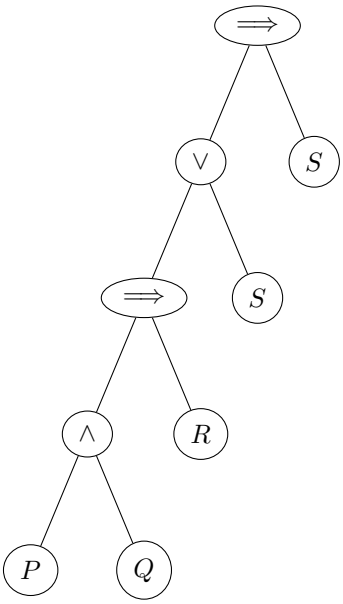
```c
typedef struct _noeud {
    char * val;
    struct _noeud *fg, *fd;
} Noeud, *Arbre;
```

2.1 Définition

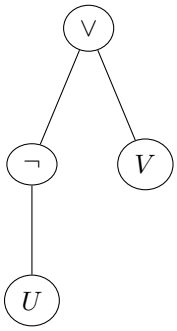
Si A et B sont deux arbres binaires, la greffe génétiquement modifiée de l'arbre B sur l'arbre A est l'arbre A modifiée comme suit :

> Si `n` est un nœud de A ayant même étiquette que la racine de B, alors le greffon G est obtenu par :
>
> — G est initialement une copie de B ;
>
> — le sous-arbre de gauche de `n` est greffé sur tous les nœuds de G sans fils gauche ;
>
> — le sous-arbre de droite de `n` est greffé sur tous les nœuds de G sans fils droit.
>
> Le nœud `n` devient finalement l'arbre G.

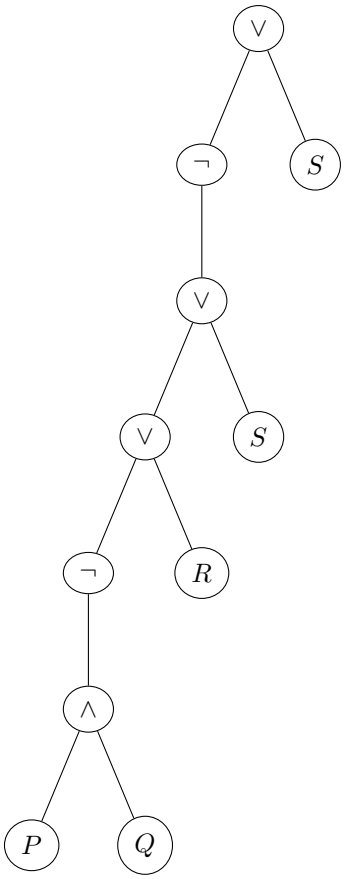
Les greffes se font des feuilles vers la racine sur les nœuds initialement dans A.



(a) Arbre A.



(b) Arbre B.



(c) Greffe de l'arbre B sur l'arbre A.

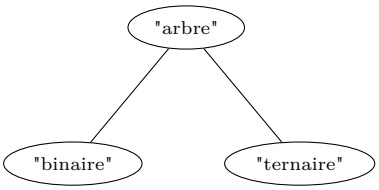
FIGURE 1 – Un exemple de greffe

2.2 Exemples de greffes

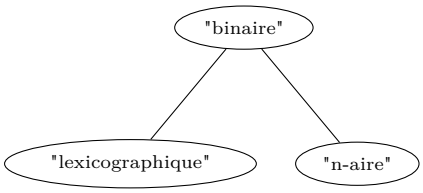
Même en partant d'arbres relativement simples, l'opération de greffe peut produire des arbres rapidement compliqué. Voici trois exemples, classés par ordre de compléxité.

2.2.1 Exemple 1

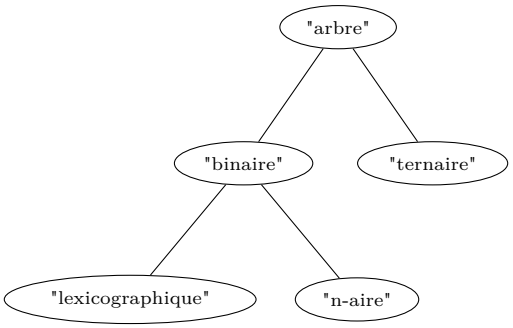
Arbre A_1 recevant la greffe :



Greffon B :

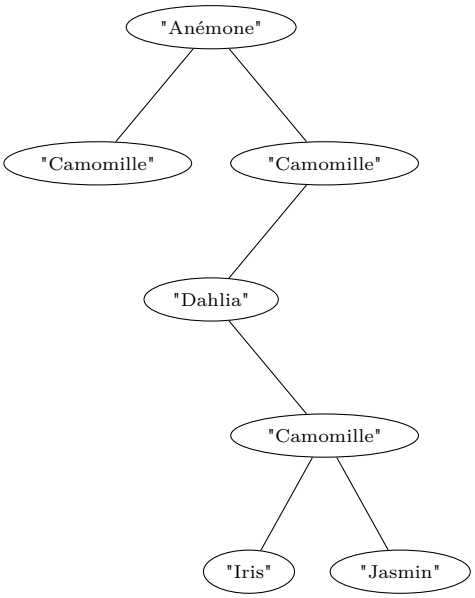


Résultat de la greffe de B sur A_1 :

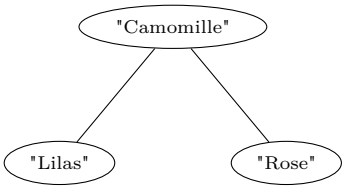


2.2.2 Exemple 2

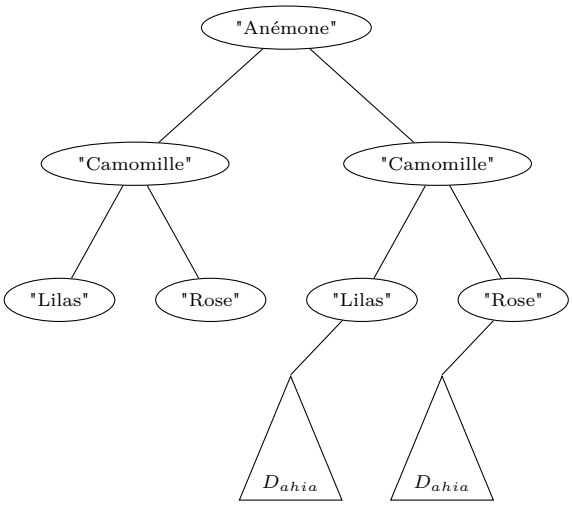
Arbre A_2 recevant la greffe :



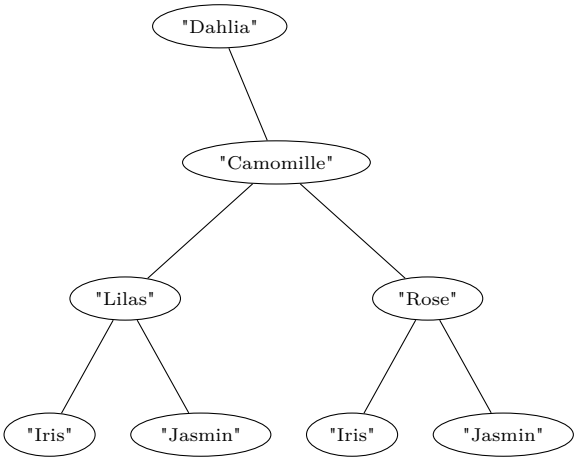
Greffon C :



Résultat de la greffe de C sur A_2 :

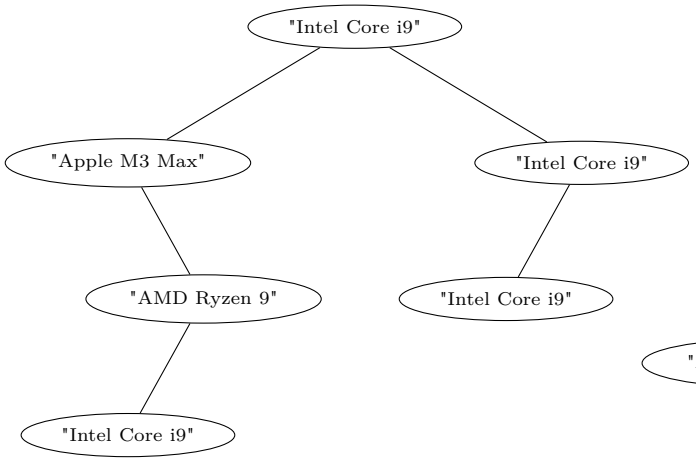


Arbre D_{ahia} :

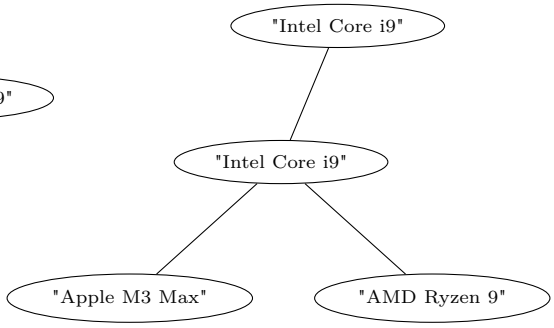


2.2.3 Exemple 3

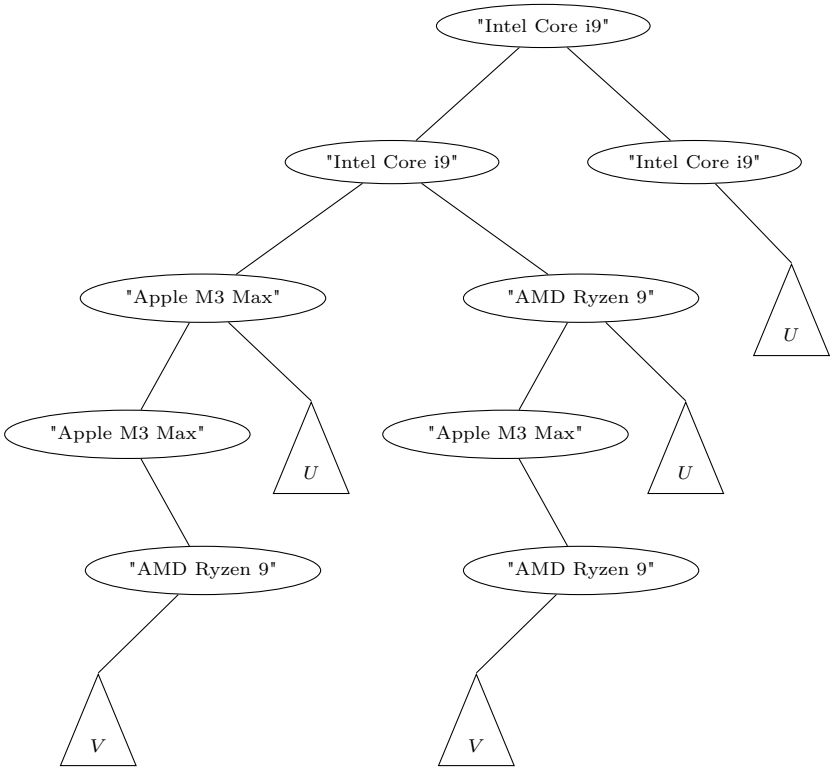
Arbre A_3 recevant la greffe :



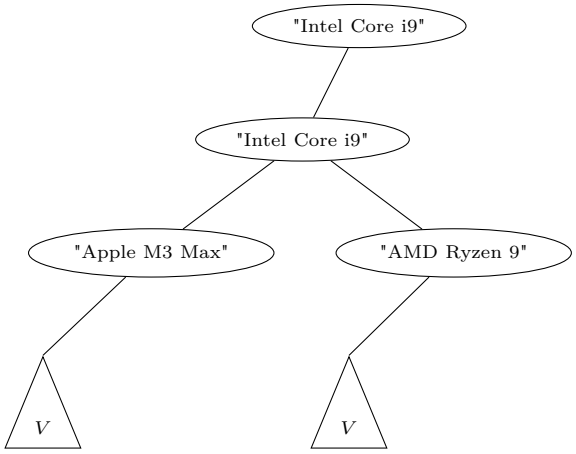
Greffon D :



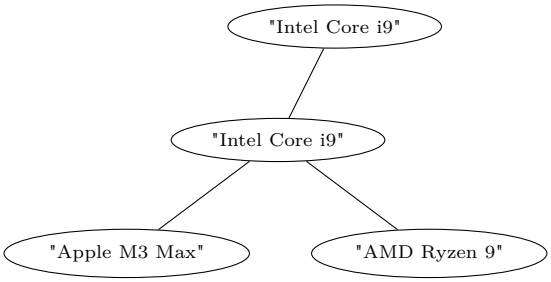
Résultat de la greffe de D sur A_3 :



Arbre U :



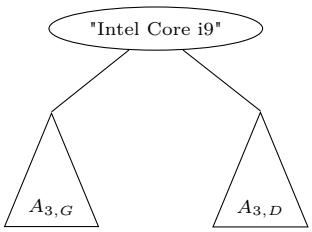
Arbre V :



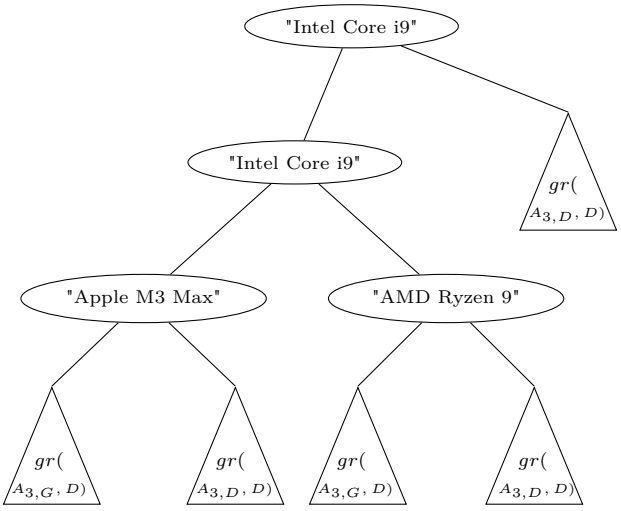
Explications de la greffe de D sur A_3 :

Notons $gr(A, G)$ la greffe de l'arbre G sur l'arbre A. Ainsi, la greffe de D sur A_3 s'effectue comme suit :

Arbre A_3 initial :

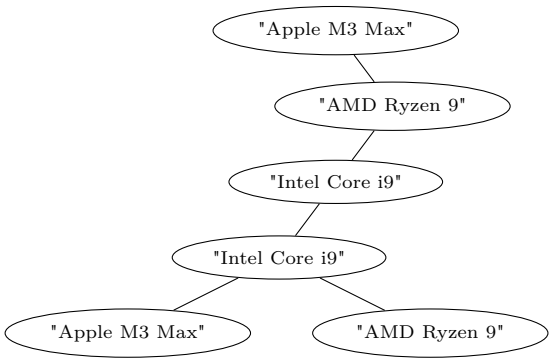


Arbre A_3 après la greffe de D :

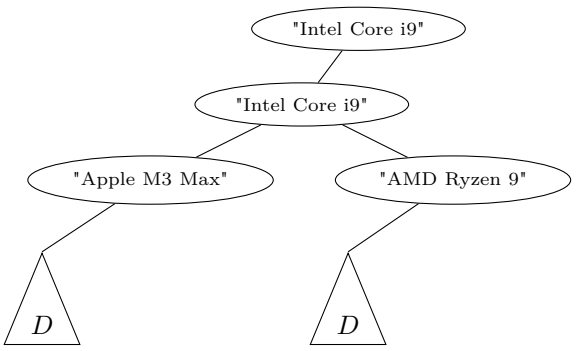


Il reste à déterminer les arbres $gr(A_{3,G}, D)$ et $gr(A_{3,D}, D)$ récursivement :

Arbre $A_{3,G}$ après la greffe de D :



Arbre $A_{3,D}$ après la greffe de D :



2.3 Travail demandé

A ce stade du devoir maison, il vous est demandé de :

1. créer un fichier `arbres_binaires.h` contenant la structure permettant de manipuler les arbres et les déclarations de fonctions que vous jugerez nécessaire pour pouvoir utiliser un arbre binaire dans un autre fichier `.c`.
2. créer un fichier `arbres_binaires.c` contenant :
   - une fonction `Noeud * alloue_noeud(char * s)` qui réserve l'espace mémoire nécessaire à un nœud et renverra l'addresse mémoire où sera stocké le nœud ou `NULL` en cas d'echec.

     N.B. : La chaine de caractères `s` sera recopiée après allocation dynamique.
   - une fonction `void liberer(Arbre * A)` qui libére tout l'espace mémoire utilisé pour stocker l'arbre `*A`.
   - les fonction `Arbre cree_A_1(void)`, `Arbre cree_A_2(void)` et `Arbre cree_A_3(void)` qui renvoie respectivement les arbres A_1, A_2 et A_3 une fois créé en mémoire.
3. créer un fichier `greffe.c`, et son homologue `greffe.h`, contenant :
   - la fonction `int copie(Arbre * dest, Arbre source)` qui copie dans `*dest` l'arbre stocké dans `source`. Celle-ci renverra 1 en cas de succès et 0 en cas d'echec.
   - la fonction `int expansion(Arbre * A, Arbre B)` qui modifie l'arbre `*A` de sorte qu'après l'appel de fonctions, celui-c contienne la greffe de l'arbre `B` sur `A`. Celle-ci renverra 1 en cas de succès et 0 en cas d'échec.

Evidemment, vous vérifierez par des tests dans un `main` indépendant chacune de vos fonctions.

3 Partie 2 : Création et sauvegarde d'un arbre

Dans cette partie, nous allons implémenter deux manières de créer un arbre en mémoire. La première s'effectuera à l'aide d'une saisie de l'utilisateur tandis que la seconde se fera grâce à la lecture d'un fichier de sauvegarde, *i.e.* d'un fichier de données sérialisées.

Rappelons que, d'après wikipédia, la *sérialisation* est "le codage d'une information sous la forme d'une suite d'informations plus petites [...] pour, par exemple, sa sauvegarde (persistance) ou son transport sur le réseau (proxy, RPC). L'activité réciproque, visant à décoder cette suite pour créer une copie conforme de l'information d'origine, s'appelle la *désérialisation*".

3.1 Codage d'un arbre et création à la volée

Une manière de créer un arbre en mémoire est de le créer à la volée par une saisie de l'utilisateur suivant le parcours en profondeur préfixe de l'arbre que l'on souhaite créer. Un nœud vide sera représenté par l'entier 0, tandis qu'un nœud non vide sera représenté par l'entier 1, suivi de la chaine de caractère qu'il représente, placée entre des guillemets.

On notera que les chaines de caractères à l'intérieur d'un nœud peuvent contenir plusieurs mots, et même des nombres !

Par exemple, les séquences à saisir pour construire en mémoire les arbres A_1, A_2 et A_3 sont données dans Figure 2. Pour la saisie utilisateur d'une telle phrase, on pourra utiliser la fonction `fgets`.

Arbre A_1 :

```
1 "arbre\n" 1 "binaire\n" 0 0 1 "ternaire\n" 0 0
```

Arbre A_2 :

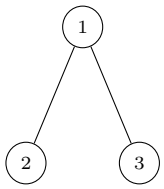
```
1 "Anémone\n" 1 "Camomille\n" 0 0 1 "Camomille\n" 1
"Dahlia\n"
0 1 "Camomille\n" 1 "Iris\n" 0 0 1 "Jasmin\n" 0 0
```

Arbre A_3 :

```
1 "Intel Core i9\n" 1 "Apple M3 Max\n" 0 1 "AMD Ryzen 9\n" 1
"Intel Core i9\n" 0 0 0 1 "Intel Core i9\n" 1 "Intel Core i9\n" 0 0 0
```

FIGURE 2 – Séquences à saisir pour créer les arbres A_1, A_2 et A_3.

h



(a) Un arbre

```
Valeur : 1
Gauche :
    Valeur : 2
    Gauche : NULL
    Droite : NULL
Droite :
    Valeur : 3
    Gauche : NULL
    Droite : NULL
```

(b) Le fichier `.saage` associé

FIGURE 3 – Un arbre simple et son fichier `saage` associé.

3.2 Définition d'un format de sérialisation simple

Afin de pouvoir sauvegarder facilement un arbre, échanger des fichiers et vérifier le fonctionnement de l'opération de greffe génétiquement modifiée définie ci-dessus, nous allons définir un standard de sérialisation des arbres binaires, nommé *"sérialisation arbre à Gustave Eiffel"*. Les informations seront sauvegardées dans un fichier texte dont l'extension sera `.saage`. Pour des raisons de clarté, la taille de celui-ci ne sera peut être pas plus petite que les données elle-même...

Dans ce standard, un nœud d'un arbre binaire sera décrit comme suit :

```
Valeur : ...
Gauche : ...
Droite : ...
```

Les valeurs associées à `Gauche` et `Droite` peuvent être soit `NULL`, soit être elle-même un nœud. Dans ce dernier cas, les données seront écrites sur une nouvelle ligne, l'indentation ayant augmentée de quatre espaces.

Un premier exemple de ce format de sérialisation est donné Figure 3. Le second exemple concerne les arbres A_1, B et la greffe de B sur A_1, serialisés dans un fichier `saage`. Leur contenu est détruit Figure 4.

```
Valeur : arbre
Gauche :
    Valeur : binaire
    Gauche :  NULL
    Droite :  NULL
Droite :
    Valeur : ternaire
    Gauche :  NULL
    Droite :  NULL
```

(a) `A_1_avant_greffe.saage`

```
Valeur : binaire
Gauche :
    Valeur : lexicographique
    Gauche :  NULL
    Droite :  NULL
Droite :
    Valeur : n-aire
    Gauche :  NULL
    Droite :  NULL
```

(b) `B.saage`

```
Valeur : arbre
Gauche :
    Valeur : binaire
    Gauche :
        Valeur : lexicographique
        Gauche : NULL
        Droite :  NULL
    Droite :
        Valeur : n-aire
        Gauche : NULL
        Droite :  NULL
Droite :
    Valeur : ternaire
    Gauche :  NULL
    Droite :  NULL
```

(c) `A_1_apres_greffe.saage`

FIGURE 4 – Trois exemples de fichiers `saage`

3.3 Travail demandé

A ce stade, il vous est demandé de

1. compléter le fichier `arbres_binaires.c` et son `.h` associé, en y ajoutant la fonction `int construit_arbre(Arbre *a)` qui permet à l'utilisateur de saisir le parcours profondeur infixe d'un arbre et de créer et stocker celui-ci dans `*A`.
2. créer un fichier `saage.c`, et son `.h` associé, contenant :
   - la fonction `int serialise(char * nom_de_fichier, Arbre A)` qui crée un fichier `saage` contenant la description de l'arbre `A`.

     La fonction renverra l'entier 1 si la sérialisation s'est passée correctement et 0 sinon. En cas d'échec, le fichier potentiellement crée sera supprimé.
   - la fonction `int deserialise(char * nom_de_fichier, Arbre * A)` qui lit le fichier `saage` dont le nom est passé en paramètre et crée l'arbre qu'il décrit dans `*A`.

     La fonction renverra l'entier 1 si l'arbre `*A` a été correctement crée et 0 sinon. En cas d'échec, l'arbre crée dans `*A` sera entièrement libéré.

Attention : Pour ces fonctions, on portera un grand soin à vérifier que chaque appel de fonction ait eu lieu correctement (par exemple, pour des appels à `sprintf`, `fopen`, `fclose`, `fgets`, `sscanf`, ...)

Evidemment, vous vérifierez par des tests dans un `main` indépendant chacune de ces deux fonctions.

4 Partie 3 : le `main`

Ecrire un `main` qui permet d'avoir le comportement suivant :

- `saage -E fichier.saage` crée une sauvegarde dans le fichier `.saage` d'un arbre saisi par l'utilisateur au clavier.
- `saage -G s.saage g.saage` crée l'arbre où le greffon `g.saage` est appliqué à l'arbre source stocké dans `s.saage`. Le résultat de la greffe sera fournis sur la sortie standard au format `.saage`

Remarques importantes :

1. Aucun autre texte que l'arbre obtenu et affiché au format `.saage` ne devra être ajouté sur la sortie standard.
2. Toute la mémoire allouée devra être libérée à l'issue du processus.

5 Conditions de rendus

Ce devoir est à réaliser en binôme au sein d'un même groupe de TP.

Le devoir est à rendre sur le e-learning du cours pour le dimanche 10 mars 2024, 23h59. Un fichier `.zip` y sera déposé contenant a minima les fichiers `arbres_binaires.c`, `greffe.c`, `saage.c` et leurs équivalent `.h`. Un `makefile` y sera aussi présent. Il sera possible aussi d'y ajouter des `.saage` de tests.

L'archive contiendra aussi un fichier `rapport.pdf` décrivant les fonctions implémentées, en particulier les fonctions intermédiaires que vous aurez décidé d'introduire, ainsi que les difficultés rencontrées et la répartition du travail au sein du binôme.

L'archive représentera un projet organisé suivant les préceptes du cours de `Perfectionnement C`. Le nom de ce fichier `.zip` sera formaté par `nom1_nom2.zip` où `nom1` et `nom2` désigne les noms de famille des membres du binôme.

La correction s'effectuera en partie de manière automatique. Il est donc particulièrement important que vous respectiez scrupuleusement les noms de fonctions demandées.

Une grande attention devra être portée à éviter toute fuite mémoire. Pour vérifier cela, vous pourrez utiliser l'outil `valgrind` sous `Linux`. L'absence de fuite de mémoire sera un élément important de la notation.