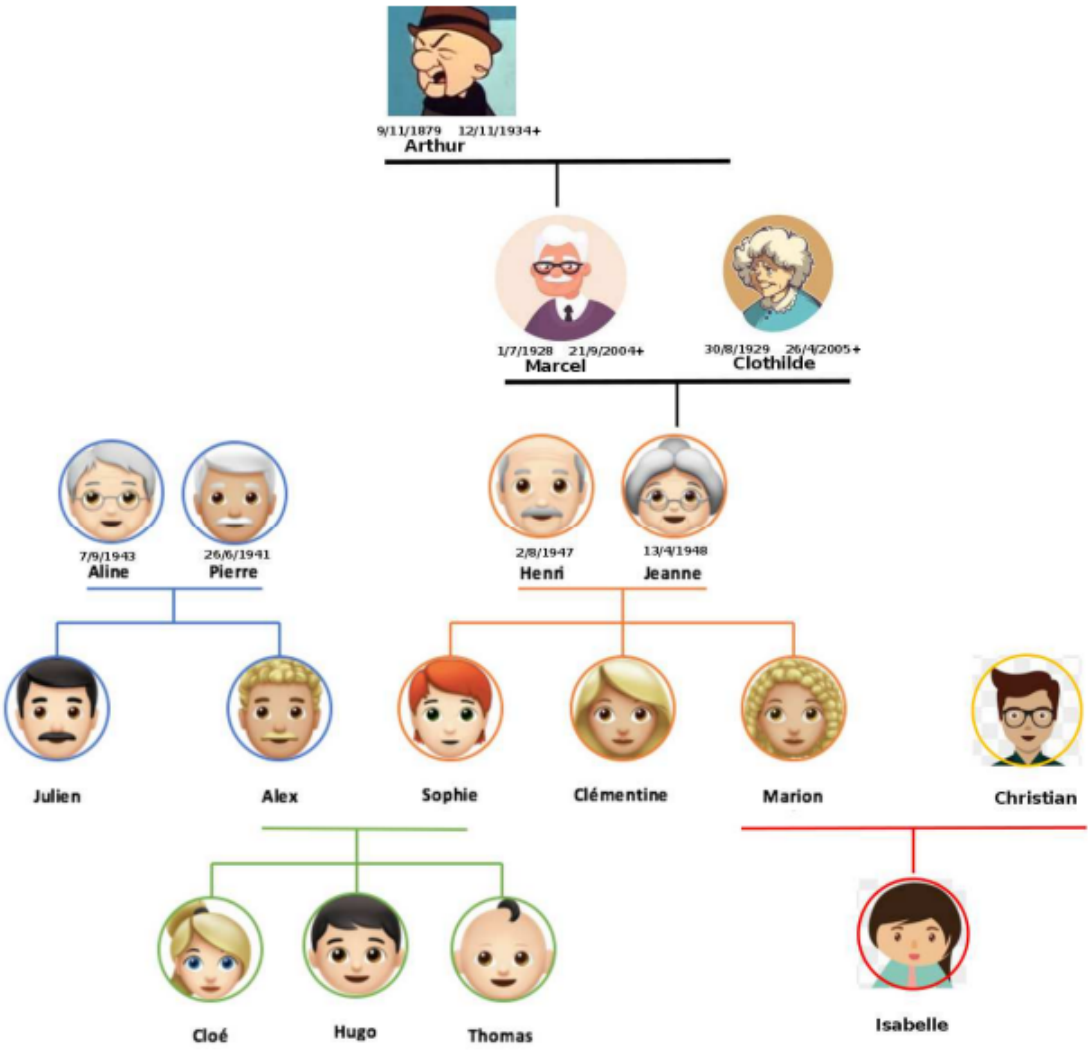
# Rapport Arbre Généalogique

ZARIKIAN Hayk et DUBERT Alexandre

Avril-Mai 2022



# Table des matières

**1 Introduction** **3**

**2 Réalisation** **3**
2.1 Construction de l'arbre généalogique . . . . . 3
2.1.1 genealogieInit(),genealogieFree(),get() . . . . . 3
2.1.2 adj() . . . . . 4
2.2 Liens de parenté . . . . . 4
2.2.1 freres_soeurs() . . . . . 4
2.2.2 cousins() . . . . . 4
2.3 Affichage . . . . . 5
2.3.1 affiche_freres_soeurs() . . . . . 5
2.3.2 affiche_enfants() . . . . . 5
2.3.3 affiche_cousins() . . . . . 5
2.3.4 affiche_oncles() . . . . . 5
2.4 Parcours de l'arbre . . . . . 5
2.4.1 ancetre() . . . . . 5
2.4.2 ancetreCommun() . . . . . 5
2.4.3 plus_ancien() . . . . . 6
2.4.4 affiche_parente() . . . . . 6
2.4.5 affiche_descendance() . . . . . 7
2.5 Fusion d'arbres généalogiques . . . . . 7
2.5.1 genealogieFusion() . . . . . 7

**3 Jeu de test** **7**

**4 Conclusion** **8**

# 1 Introduction

Ce rapport est dédié à notre projet de structures de données et algorithmes 2 dans le cadre de notre deuxième année de licence d'Informatique à l'UFR de Mathématiques et d'Informatique à l'Université de Strasbourg.
L'objectif de ce projet était de gérer un arbre généalogique dans un cas simple et parfaitement idéal. Ce projet, réalisé en langage C, doit permettre la création et la destruction d'une généalogie, l'adjonction d'individu, la recherche de parents et les tests de parenté. Le but étant d'aboutir dans un premier temps à une gestion fonctionnelle, et dans un deuxième temps de limiter la complexité du programme.
Nous avons choisi de travailler en binôme. Pour ce faire, nous avons utilisé un répertoire GIT sur le GitLab de l'université. Nous nous sommes réparti les tâches au préalable, et dès que nous finissions une fonction, nous l'envoyions sur le dépôt distant.
Pour nous aider les structures suivantes étaient déjà définies :

```
typedef struct s_genealogie {
       Individu      *tab;              // tableau alloué dynamiquement
       Nat           id_cur;            // identifiant actuel
       Nat           taille_max_tab;    // taille max du tableau
} Genealogie;
```

FIGURE 1 – Structure Généalogie

```
#define LG_MAX 64

typedef struct s_date { unsigned short jour, mois, annee; } date;
typedef unsigned int ident;

typedef struct s_individu {

       char nom[LG_MAX];
       date  naissance, deces;
       ident  pere, mere, cadet, faine;

} Individu;
```

FIGURE 2 – Structure Individu

Informations importantes :
- La généalogie contient un tableau d'individu
- Les individus sont liés entre eux par leur ident
- L'indice d'un individu dans la généalogie est son ident incrémenté de 1
- L'ident Ω désigne les individus inexistants dans la généalogie
- Dans le cas d'un père, d'une mère, d'un fils aîné ou d'un cadet inconnu, le champ correspondant de l'individu vaut Ω

# 2 Réalisation

## 2.1 Construction de l'arbre généalogique

### 2.1.1 genealogieInit(),genealogieFree(),get()

L'initialisation de l'arbre généalogique se fait via la fonction `void genealogieInit(Genealogie *g)` qui va initialiser la taille max du tableau à `TAILLE_INIT` (50), l'id courant qui sera initialisé à Ω (0) et le tableau d'individus qui sera alloué dans la mémoire avec la taille max (réalloc si besoin). Une fonction d'accès `Individu* get(Genealogie *g, ident x)` est mise à disposition

pour faciliter le codage. La fonction de désallocation `void genealogieFree(Genealogie *g)` est cruciale pour éviter les fuites de mémoire lorsque l'arbre n'est plus utilisé !

### 2.1.2 adj()

Pour cette fonction, il faut d'abord s'assurer d'avoir assez de place dans le tableau, ensuite nous créons un nouvel individu. La difficulté réside dans le fait que les champs `faine` des parents et `cadet` des frères et sœurs de l'individu doivent être mis à jour. Voici comment nous avons résolu ce problème : nous avons pris un des parents de l'individu qui est dans la généalogie, s'il n'a pas de fils aîné alors ce nouvel individu est le fils aîné de ses parents. Sinon nous créons une liste de tous ses enfants ainsi que du nouvel individu, puis nous trions cette liste par date de naissance. Le premier élément de la liste est le nouveau fils aîné et le cadet de chaque fils est l'élément suivant de la liste. Nous avons limité la taille du tableau à 15 car nous avons estimé que nous pouvions plafonner le nombre d'enfants par couple à 15. Pour le tri du tableau nous avons utilisé l'algorithme du bubble sort, nous l'avons choisi avant tout pour sa simplicité. Nous avions conçu la fonction `int compDate(date date1, date date2)` pour qu'elle retourne un booléen mais pour un usage interne dans la fonction de fusion nous avons fait en sorte qu'elle retourne un entier.

**Algorithm 1** adj

```
ind ← initIndividu
g− > tab[g− > id_cur] ← ind
if aUnPere(ind) then
  parent ← ind.pere
else
  parent ← ind.mere
end if
if aUnPere(ind)  OU  aUneMere(ind) then
  if !aUnFils(parent) then
    ind.pere.faine ← ind // idem pour mere
  else
    fils_suiv ← parent.faine
    for i de 0 a 14  ET  fils_suiv ≠ Ω do
      ajout(liste_fils, fils_suiv)
      fils_suiv ← fils_suiv.cadet
    end for
    ajout(liste_fils, ind)
    tri_par_date(liste_fils)
    mise à jour des champs faine et cadet
  end if
end if
```

## 2.2 Liens de parenté

### 2.2.1 freres_soeurs()

La fonction `bool frere_soeurs(Genealogie *g, ident x, ident y)` retourne un booléen indiquant si l'un est le frère ou la sœur de l'autre, on récupère alors le père et la mère des deux individus car il est possible que l'un des individus n'ait qu'un seul parent, c'est pour cela que nous faisont les test nécessaires. Si le père de l'un est le père de l'autre alors les deux individus sont frères ou sœurs, donc la fonction renvoie true.

### 2.2.2 cousins()

La fonction `bool cousins(Genealogie *g, ident x, ident y)` retourne un booléen indiquant si l'un est le cousin/cousine de l'autre, pour cela on récupère le père et la mère des deux

individus puis nous utilisons la fonction précédente pour vérifier si l'un des parents de l'un et de l'autre ont un lien fraternel, si c'est le cas la fonction renvoie true.

## 2.3 Affichage

### 2.3.1 affiche_freres_soeurs()

Cette fonction `void affiche_freres_soeurs(Genealogie *g,ident x)` affiche les frères ou soeurs de l'individu x spécifié en argument, on cherche à récupérer le père de l'individu x en question pour ensuite récupérer son fils aîné, car nous ne savons pas si l'individu x spécifié est l'aîné. Puis nous allons utiliser une boucle qui va parcourir de l'ainé au dernier frère tout en affichant le nom à chaque passage d'un fils à un autre.

### 2.3.2 affiche_enfants()

La fonction `void affiche_enfants(Genealogie *g, ident x)` va permettre l'affichage des enfants de l'individu x. Il suffit de récupérer le fils aîné de l'individu, x puis de parcourir une boucle allant de l'aîné au dernier fils en affichant le nom à chaque itération. Nous aurions très bien pu récupérer l'aîné, l'afficher puis afficher ses frères et sœur via la fonction précédente mais il y aurait eut plus de vérification inutiles.

### 2.3.3 affiche_cousins()

Pour cette fonction `void affiche_cousins(Genealogie *g, ident x)`, nous avons dû utiliser la fonction annexe `void affiche_enfants_freres_soeurs(Genealogie *g, ident x)` qui va récupérer le père ou mère de l'individu x puis récupérer son fils aîné. Ensuite elle parcourt de l'ainé au dernier fils en affichant leurs enfants via la fonction `void affiche_enfants(Genealogie *g, ident x)`. Pour revenir à notre fonction qui affiche les cousins de ident x nous allons récupérer le père et la mère de l'individu x puis afficher les enfants des frères et soeurs du père et mère de l'individu x car les enfants des frères et soeurs des parents de l'individu x sont les cousins/cousines de l'individu x.

### 2.3.4 affiche_oncles()

La fonction d'affichage `void affiche_oncles(Genealogie *g, ident x)` est assez simple puisque nous récupérons le père et la mère de l'individu x pour ensuite faire appel à la fonction `void affiche_freres_soeurs(Genealogie *g, ident x)` deux fois, l'une avec le père en argument et l'autre avec la mère en argument. Nous aurons donc l'affichage des frères et soeurs du père et de la mère de l'ident x c'est-à-dire les oncles et les tantes.

## 2.4 Parcours de l'arbre

### 2.4.1 ancetre()

Les ancêtres de y sont les nœuds de l'arbre binaire de racine y. Donc vérifier si x est l'ancêtre de y revient à une recherche dans un arbre binaire. Ainsi nous avons choisi de faire cette recherche par un parcours en profondeur d'abord. Si $x = y$ alors la fonction renvoie vrai, si $x = \Omega$, la fonction renvoie faux et sinon on renvoie le OU de l'appel récursif avec le père de y et de celui avec la mère de y.

### 2.4.2 ancetreCommun()

La fonction renvoie simplement le OU du test si x est l'ancêtre de y, de l'appel récursif avec le père de x et de l'appel récursif avec la mère de x. Cette solution est très simple mais explose en terme de complexité car chaque appel effectue un parcours d'arbre et génère deux autres appels.

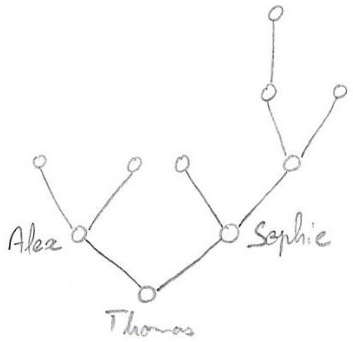



FIGURE 3 – Ancêtres de Thomas, on remarque qu'il s'agit d'un arbre binaire de racine Thomas

#### 2.4.3 plus_ancien()

La condition d'arrêt de la récursivité est que l'individu est une feuille (il n'a pas de parents), dans ce cas on renvoie `ident` de l'individu. Sinon, s'il n'a qu'un seul parent, on retourne l'appel récursif sur ce parent (le plus ancien est forcément du côté de son unique parent). Dans un dernier cas, s'il a ses deux parents, on stocke le retour de l'appel récursif sur son père et sur sa mère dans deux variables, puis on obtient le plus ancien des deux en appelant la fonction `compDate` sur ces deux variables.

#### 2.4.4 affiche_parente()

La parenté d'un individu est l'arbre de racine x. Pour afficher les individus génération par génération, il n'y a pas le choix, il faut faire un parcours en largeur d'abord. Nous avons utilisé l'algorithme vu en cours, donc nous avons dû implanter une structure de file. Pour nous faciliter le travail, nous avons considéré l'arbre comme un arbre binaire, pour chaque individu plutôt que de prendre son père et sa mère on prend l'aîné de la fratrie de son père et de sa mère. Pour afficher la fratrie, nous faisons simplement appel à une fonction `affiche_cadet`. Le marquage des nœuds déjà visité a été fait à l'aide d'une deuxième file, tous les nœuds marqués appartient à cette file.

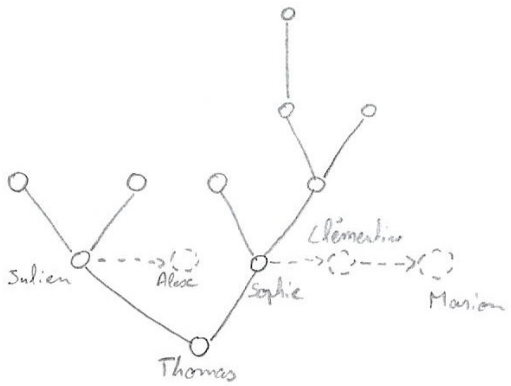



FIGURE 4 – Parenté de Thomas, la partie en pointillé n'est pas ajoutée à la file mais est affichée grâce à affiche_cadet

L'affichage des génération a été compliqué à mettre en place, mais nous avons finalement trouvé une méthode qui fonctionne très bien. Lors du première appel, il n'y a que les deux parents de l'individu dans la file, donc `taille_generation` vaut 2. Dès que la tête de file est traitée on décrémente `taille_generation`. Lorsque `taille_generation` vaut zéro, on sait d'une part que la

génération actuelle a été affichée en entier, et d'autre part que la taille de la prochaine génération est la longueur actuelle de la file.

#### 2.4.5 affiche_descendance()

La fonction `void affiche_descendance(Genealogie *g, ident x)` affiche la descendance de l'individu x, nous avons utilisé une variable statique, il est vrai qu'en terme de bonne pratique une variable statique n'est pas conseillée mais dans ce cas là, elle s'avère assez intéressante car elle permet de garder la valeur simplement, ce qui va permettre l'affichage des niveaux de descendance. Tout d'abord, la fonction récupère l'aîné de l'individu x, puis affiche ses enfants et les enfants de ses frères et sœurs ensuite renvoie la fonction de descendance avec l'aîné de l'individu x. Finalement, c'est une fonction récursive qui prend en argument l'aîné de l'aîné en incrémentant la génération de 1 et en affichant les individus de cette génération, jusqu'à que l'aîné sois égal à $\Omega$ ce qui va réinitialiser la génération à 1. Nous avons eu quelques difficultés à récupérer la génération de manière correcte d'où l'utilisation du statique, nous sommes dans un cas où la variable statique retrouve toujours son point initial (à 1) car le dernier appel à cette fonction rentre dans la condition où l'ident vaut $\Omega$.

### 2.5 Fusion d'arbres généalogiques

#### 2.5.1 genealogieFusion()

La fusion d'une généalogie est l'union de deux généalogies jointes ou disjointes. Dans le cas de généalogies disjointes on obtient une forêt, et dans le cas de généalogies jointes un unique arbre enraciné sur l'intersection des ensembles. Dans les deux cas il faut préserver les relations entre individus lors de la fusion. Par exemple si mon père était l'`ident` 3 dans ma généalogie d'origine et que dans la fusion l'`ident` 29 lui est attribué, alors il faudra mettre à jour mon champ père avec son nouvel `ident` (29 et non 3). Et dans le deuxième cas uniquement, il faut fusionner les doublons. Par exemple dans une généalogie j'ai un père seul et dans l'autre j'ai une mère seule, alors dans la fusion j'aurai un père et une mère, avec les identifiants de la nouvelle généalogie. Mes frères et sœurs aussi devront avoir ces champs mis à jour. Pour la fusion des doublons il n'y aura pas beaucoup de tests à effectuer car on ignore les incohérences.
Pour commencer on copie simplement la généalogie la plus grande dans la fusion. Ensuite on effectue les deux opérations précédente de l'autre généalogie vers la généalogie résultat.
Pour faire faire correspondre un `ident` dans la généalogie d'origine avec celui de la nouvelle nous avons mis en place une table de correspondance. Elle est dynamiquement allouée en fonction de la taille de la plus petite généalogie, car c'est uniquement les individus de celle-ci qui verront leur `ident` changer lors de la fusion.
Pour chaque individu de la plus petite généalogie on fait un test d'appartenance à la fusion (qui au début de la fonction est la copie de la plus grande généalogie). Si ce test est faux, on créé l'individu dans la fusion, avec ses champs pere et mere issus de notre table de correspondance (si père et mère il y a alors ils sont forcément dans la table, car `ident` d'un individu est toujours supérieur à celui de ses descendants d'après les règles de construction de la généalogie). Puis on met à jour la table avec son `ident` dans la fusion. Sinon, si c'est un doublon, on ajoute `ident` de la fusion à la table, puis on fusionne ses parents.

## 3 Jeu de test

Afin de s'assurer du bon fonctionnement de la gestion de généalogie, nous avons mis en place un jeu de test. Ce jeu de test comprend un appel à toutes nos fonctions, dans des cas ni trop triviaux ni trop alambiqués. Les résultats sont conformes à ce que nous attendions. Pour faire ces tests il suffit d'appeler l'exécutable `bin/prog`.

# 4 Conclusion

Ce projet nous a beaucoup apporté sur deux plans. Premièrement sur le plan de la pratique dans la manipulation des arbres. En effet, il a fallut s'approprier la structure de données, la comprendre, trouver les moyens de la parcourir et d'accéder aux données. Sur ce point là, la partie 4 sur le parcours était très intéressante, nous avons eu l'occasion de manipuler le parcours en profondeur et en largeur. Et deuxièmement sur le plan du travail d'équipe. Le partage du travail était très équilibré, la communication très bonne. Nous nous sommes mutuellement donnés des idées et conseillés. L'utilisation d'un dépôt GIT était une bonne idée et nous a fait gagner en efficacité.
Finalement ce projet a été très plaisant à mener à bien, le résultat est très satisfaisant et nous avons beaucoup progressé.