Projet : manipulation d'un arbre généalogique
Semestre impair 2023-2024

1 Introduction

Le but de ce projet est de manipuler un arbre généalogique en mettant en œuvre les principes vus en CTD et TP. Il s'agit d'un modèle simplifié d'arbre généalogique qui ne prend pas en compte la réalité des situations familiales (adoptions, etc.). Il a pour seul objectif de vous faire progresser dans votre apprentissage du langage C.

1.1 Intégrité académique

Le travail doit être réalisé individuellement. Vous pouvez discuter du sujet avec vos camarades mais il est strictement interdit de copier-coller du code. L'équipe pédagogique utilise des outils de mesure de similarité de codes sources. Tout plagiat sera sévèrement sanctionné.

1.2 Évaluation

Le travail que vous allez produire sera évalué :

- à partir d'un dépôt sur Moodle que vous devrez effectuer au plus tard le lundi 11 décembre 2023 à 23h59 (n'attendez pas le dernier moment...) ; quelle que soit la raison, si le dépôt est vide le 11 décembre à minuit alors vous aurez 0 ; il est donc conseillé de déposer votre projet avant, même s'il n'est pas fini ; mais la version déposée doit pouvoir être compilée sans erreur ;
- à partir d'un oral qui se déroulera le mercredi 13 décembre 2023 :
  - à 7h45 pour les groupes B11, B12 et B21 ;
  - à 15h45 pour les groupes A11, A12, A21, A22 et A31.

1.3 Barème

Le projet comporte trois niveaux de difficulté croissante, numérotés de 1 à 3. La réalisation complète et exacte de chaque niveau donne la possibilité d'obtenir une certaine note maximale, comme cela est indiqué dans le tableau suivant :

Niveau	Note maximale
1	8
2	14
3	20

Pour commencer un niveau, il faut avoir complètement réalisé le niveau précédent. Par exemple, si vous n'avez pas terminé le niveau 2 et si vous réalisez une partie du niveau 3, vous serez tout de même noté sur 14. Un niveau est considéré comme réalisé quand toutes ses fonctionnalités sont programmées et que le programme correspondant au niveau s'exécute sans erreur.
Attention : si le projet que vous avez déposé sur Moodle ne peut pas être compilé sans erreur en utilisant la commande `make`, vous aurez automatiquement la note 0.

1.4 Dépôt sur Moodle

Vous déposerez sur Moodle un seul fichier archive au format ZIP qui doit être produit en suivant les directives suivantes.

1. Créez un répertoire dont le nom doit suivre le modèle :
   NOM-PRENOM-NUMETU
   où :
   - NOM, PRENOM et NUMETU doivent être remplacés, respectivement et dans cet ordre, par votre nom de famille en majuscules, votre prénom en majuscules et votre numéro d'étudiant (8 chiffres) ;
   - n'apparaît aucune lettre accentuée ;
   - n'apparaît aucun espace ni apostrophe, les éventuels espaces et apostrophes dans votre nom ou votre prénom devant être remplacés par des tirets - (signe moins).
2. Placez dans ce répertoire :
   - les fichiers sources des modules de votre projet, c'est-à-dire les fichiers d'extensions `.h` et `.c` ;
   - les fichiers contenant la fonction principale (`main`) de chaque niveau (`testidentite.c` pour le niveau 1, `testgenea.c` pour le niveau 2, `visuarbre.c` et `visuarbreasc.c` pour le niveau 3) ;
   - les fichiers de description des dépendances de nom `Makefile`n, où n vaut 1, 2, 3 et 4, si vous avez réalisé tous les niveaux, permettant, grâce à la commande `make -f Makefile`n, de produire l'exécutable correspondant ;
   - un fichier de texte (obtenu avec un simple éditeur de texte) de nom `niveau-`n`.txt` où vous remplacerez n par le numéro du niveau que vous avez réalisé (1, 2 ou 3) ; ce fichier pourra être vide ou contenir d'éventuelles informations destinées à expliquer les choix que vous aurez faits.

   Vos fichiers sources doivent être indentés et contenir des commentaires permettant de bien comprendre votre travail. Aucun autre fichier (objets, exécutable, données, résultats, etc.) ne doit être présent dans le répertoire.
3. Créez une archive au format ZIP de ce répertoire ; cette archive doit porter le même nom que le répertoire auquel est ajoutée l'extension `.zip` ; pour produire cette archive, dans le terminal, vous pouvez vous placer dans le répertoire parent du répertoire contenant vos fichiers sources et utiliser la commande (␣ représente un caractère espace) :
   `zip␣`NOM-PRENOM-NUMETU`.zip␣`NOM-PRENOM-NUMETU`/*`

Respectez bien toutes ces consignes car des tests et des vérifications automatiques seront effectués sur votre projet.

1.5 Oral

Au moment de l'oral, vous devrez être prêt à :
- faire fonctionner votre programme ;
- répondre aux questions de l'enseignant ;
- effectuer des tests ou des modifications demandées par l'enseignant.

2 Niveau 1 : identité d'une personne

2.1 Module `identite`

Il s'agit d'écrire un module[1] `identite` qui permet de gérer l'identité d'une personne en utilisant la structure de données privée suivante :

```
#define LG_DATE 10 // Nombre de caractères d'une date sous la forme jj/mm/aaaa
```

1. Pour chaque module que vous devez écrire, vous devez appliquer les principes vus en CTD concernant la programmation modulaire, notamment l'encapsulation des données et des traitements.

```c
// Identité d'une personne
struct sIdentite
{
  int Identifiant; // Identifiant unique
  char *Nom;  // Nom (chaîne de caractères allouée dynamiquement)
  char *Prenom;  // Prénom (chaîne de caractères allouée dynamiquement)
  char Sexe; // 'F' ou 'M'
  char DateNaissance[LG_DATE+1]; // Date de naissance sous la forme jj/mm/aaaa
};
```

et le type public suivant :

```c
// Type permettant de manipuler l'identité d'une personne
typedef struct sIdentite *tIdentite;
```

Exemple : la représentation mémoire de l'identité d'une personne de sexe masculin, dont l'identifiant est 16, dont le nom est DUFF, dont le prénom est John et dont la date de naissance est le 13 décembre 2001 devra être celle qui est présentée sur la figure 1.

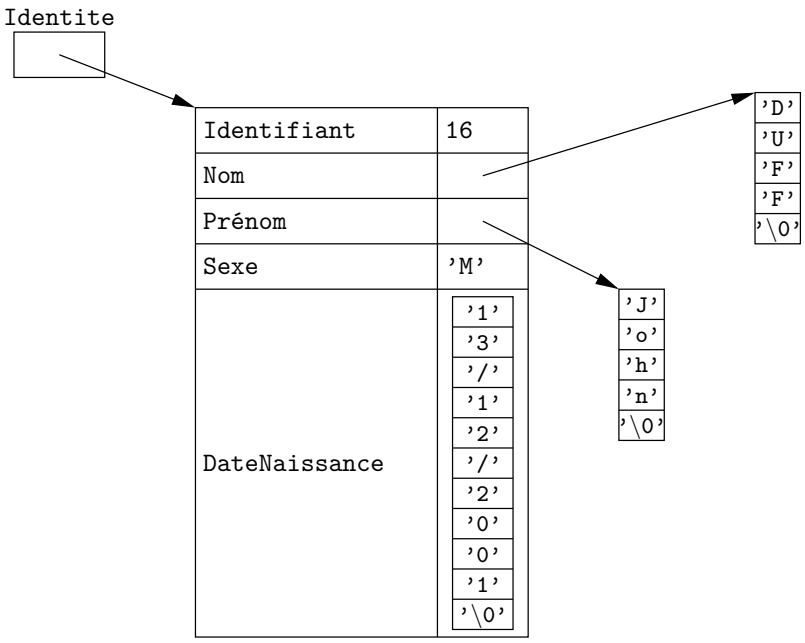

FIGURE 1 – *Représentation mémoire de l'identité de John DUFF, né le 13/12/2001, d'identifiant 16.*

Le module `identite` devra comporter les fonctions suivantes :

- `tIdentite IdentiteCreer(int Id, char *Nom, char *Prenom, char Sexe, char DateNais[])` : crée une identité à partir des informations passées en paramètres, c'est-à-dire son identifiant, son nom, son prénom, son sexe et sa date de naissance. La structure ainsi que les chaînes de caractères correspondant aux champs `Nom` et `Prenom` doivent être allouées dynamiquement. Ces deux chaînes de caractères doivent être des copies des deux chaînes passées en paramètres de la fonction. L'espace mémoire réservé avant de faire la copie doit être le plus petit possible, c'est-à-dire correspondre exactement aux longueurs des deux chaînes passées en paramètres de la fonction. On suppose que la date de naissance passée en paramètre de la fonction est bien une chaîne de 10 caractères dans le format prévu. Cette fonction doit retourner l'identité créée ou `NULL` en cas de problème d'allocation mémoire.

- `int IdentiteIdentifiant(tIdentite Identite)` : retourne l'identifiant de la personne décrite par `Identite`.
- `char *IdentiteNom(tIdentite Identite)` : retourne l'adresse du premier caractère du nom de la personne décrite par `Identite`.
- `char *IdentitePrenom(tIdentite Identite)` : retourne l'adresse du premier caractère du prénom de la personne décrite par `Identite`.
- `char IdentiteSexe(tIdentite Identite)` : retourne le sexe (`'F'` ou `'M'`) de la personne décrite par `Identite`.
- `char *IdentiteDateNaissance(tIdentite Identite)` : retourne l'adresse du premier caractère de la date de naissance de la personne décrite par `Identite`.
- `void IdentiteAfficher(tIdentite Identite)` : affiche à l'écran les informations données par `Identite` sous la forme d'une ligne ayant le modèle suivant (␣ représente un caractère espace) :
  `[`*identifiant*`]`␣*nom*␣*prénom*`,`␣*sexe*`,`␣*date de naissance*
  Avec l'exemple donné ci-dessus, cette fonction doit afficher la ligne :
  `[16]␣DUFF␣John,␣M,␣13/12/2001`
- `void IdentiteLiberer(tIdentite Identite)` : libère tout l'espace mémoire (les deux chaînes et la structure) occupé par l'identité représentée par `Identite`.
- `tIdentite IdentiteLiref(FILE *f)` : lit dans le fichier de texte d'identificateur `f` les informations constituant l'identité d'une personne et retourne l'identité créée. Cette fonction suppose que l'identité est décrite dans le fichier selon le format dans lequel chaque information est sur une ligne constituée d'au plus 80 caractères, les lignes étant placées dans l'ordre suivant :
  identifiant
  nom
  prénom
  sexe
  date de naissance
  Cette fonction doit retourner `NULL` s'il n'y a plus d'informations à lire dans le fichier, c'est-à-dire si une ligne vide (constituée uniquement du caractère `\n`) est lue ou si la fin du fichier est atteinte.
  Le fichier `personne.ind` contient les informations correspondant à l'exemple ci-dessus écrites dans ce format.

Remarque : dans tous les modules, il est possible d'écrire des fonctions « auxiliaires » qui ne seront pas publiques ; elles devront donc être de classe statique.

2.2 Test du module `identite`

2.2.1 Compilation séparée

Créez le fichier source `testidentite.c` contenant la fonction `main` afin de tester les différentes fonctions du module `identite`.

Effectuez la compilation séparée grâce à l'utilitaire `make`. Pour cela, écrivez un fichier de description des dépendances `Makefile1` et lancez les compilations en tapant :

```
make -f Makefile1
```

2.2.2 Aide à la mise au point

Afin de vérifier que vos programmes n'ont pas de « fuites » mémoire, c'est-à-dire gèrent correctement l'allocation dynamique et la libération des zones en mémoire, un outil d'analyse dynamique peut vous

aider. Sous Linux, vous pouvez utiliser Valgrind [2]. Par exemple, avec l'exécutable `testidentite`, lancez l'analyse avec Valgrind en tapant [3] :

```
valgrind ./testidentite
```

et lisez le résultat pour savoir s'il a détecté des fuites (« *leaks* »).

Sous MacOS, tapez la commande :

```
leaks --atExit -- ./testidentite
```

et vérifiez le nombre de « *leaks* » à la fin de l'affichage produit.

Il existe également des outils (*code sanitizers* [4]) qui détectent, au moment de l'exécution du programme, les erreurs d'accès à la mémoire comme le débordement d'un tableau. Sur les machines des salles de TP, quand vous invoquez `gcc`, vous pouvez ajouter l'option `-fsanitize=address` qui permettra l'affichage d'informations en cas d'erreur d'accès à la mémoire lors de l'exécution d'un programme.

3 Niveau 2 : arbre généalogique

3.1 Module `genea`

Il s'agit d'écrire un module `genea`, faisant appel au module `identite`, qui permet de gérer un arbre généalogique en utilisant une liste chaînée dynamique de fiches. La structure de données privée est la suivante :

```
// Arbre généalogique
struct sArbre
{
  struct sFiche *pPremiere; // Adresse de la première fiche
  struct sFiche *pDerniere; // Adresse de la dernière fiche
};

// Fiche associée à chaque individu présent dans l'arbre
struct sFiche
{
  tIdentite Identite; // Accès aux informations de l'identité de la personne
  struct sFiche *pPere; // Adresse de la fiche du père
  struct sFiche *pMere; // Adresse de la fiche de la mère
  struct sFiche *pSuivante; // Adresse de la fiche suivante
};
```

et le type public suivant :

```
// Type permettant de manipuler un arbre généalogique
typedef struct sArbre *tArbre;
```

Exemple : la représentation mémoire d'un arbre constitué de deux personnes [5], Pierre et Catherine, qui sont frère et soeur, ainsi que de Marthe et Jean qui sont leurs parents, s'ils sont ajoutés dans l'arbre dans l'ordre Pierre-Catherine-Marthe-Jean, devra être celle qui est présentée sur la figure 2.

Le module `genea` devra comporter les fonctions suivantes :

- `tArbre ArbreCreer(void)` : crée un arbre généalogique vide et retourne l'arbre créé ou `NULL` en cas de problème d'allocation mémoire.

2. https://fr.wikipedia.org/wiki/Valgrind
3. Si des paramètres devaient être passés à l'exécutable depuis la ligne de commande, il suffirait de les placer normalement après le nom de l'exécutable.
4. https://en.wikipedia.org/wiki/Code_sanitizer
5. À partir d'ici, les exemples sont inspirés des arbres disponibles sur la page https://www.arestempslibre.fr/html/smanugen/geneal/geneal1.htm

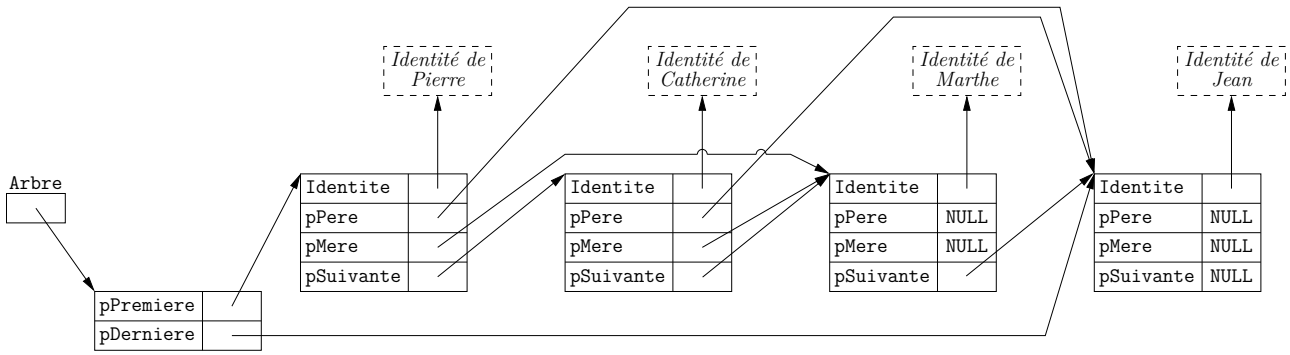

FIGURE 2 – *Représentation mémoire d'un arbre généalogique contenant quatre personnes : les boites en pointillés sont les représentations mémoire des identités des quatre personnes ; pour alléger le schéma, les détails de ces représentations, qui sont visibles pour l'exemple de la section 2.1, n'apparaissent pas ici ; les identités des personnes doivent être gérées grâce au module* `identite`.

- `void ArbreAfficher(tArbre Arbre)` : affiche à l'écran les informations contenues dans l'arbre représenté par `Arbre`. Cette fonction doit afficher, pour chaque personne présente dans l'arbre, les informations sous la forme des trois lignes suivantes :
  identité de la personne
  ⟼`Père :` *identité du père de la personne*
  ⟼`Mère :` *identité de la mère de la personne*
  où ⟼ représente le caractère tabulation (`'\t'`) et les identités sont affichées selon le format de la fonction `IdentiteAfficher` du module `identite`. Si un parent n'est pas présent dans l'arbre, c'est le mot `inconnu` qui doit être affiché à la place de son identité.
  Voici le résultat de l'affichage de l'arbre généalogique constitué des quatre personnes citées précédemment :

```
[122] DUBOURG Pierre, M, 15/12/1910
        Père : [18] DUBOURG Jean, M, 21/07/1866
        Mère : [19] COURBIN Marthe, F, 22/04/1872
[9] DUBOURG Catherine, F, 21/08/1904
        Père : [18] DUBOURG Jean, M, 21/07/1866
        Mère : [19] COURBIN Marthe, F, 22/04/1872
[19] COURBIN Marthe, F, 22/04/1872
        Père : inconnu
        Mère : inconnue
[18] DUBOURG Jean, M, 21/07/1866
        Père : inconnu
        Mère : inconnue
```

- `void ArbreAjouterPersonne(tArbre Arbre, tIdentite Identite)` : ajoute une personne d'identité `Identite` dans l'arbre `Arbre`. Cet ajout doit se faire à la fin de la liste des personnes. En cas de problème d'allocation mémoire, l'exécution ne doit pas être interrompue mais un message d'erreur doit être affiché sur la sortie standard des erreurs.
- `void ArbreLiberer(tArbre Arbre)` : libère tout l'espace mémoire (y compris les identités) occupé par l'arbre représenté par `Arbre`.
- `tArbre ArbreLirePersonnesFichier(char Fichier[])` : crée un arbre généalogique à partir d'une liste d'identités de personnes stockées dans le fichier de texte de désignation `Fichier`. Cette

fonction doit retourner l'arbre créé ou `NULL` en cas de problème. Les personnes doivent être ajoutées en fin de liste au fur et à mesure de la lecture dans le fichier. Le fichier ne contenant pas d'information sur les liens de parenté, ils doivent être initialisés à `NULL`. Le fichier `arbre4.ind` correspond à l'exemple cité ci-dessus.

- `void ArbreAjouterLienParente(tArbre Arbre, int IdEnfant, int IdParent, char Parente)` : ajoute dans l'arbre `Arbre` un lien de parenté entre l'enfant d'indentifiant `IdEnfant` et le parent d'indentifiant `IdParent`. Le type de lien de parenté est décrit par `Parente` qui peut valoir `'M'` pour « mère » ou `'P'` pour « père ». Si un identifiant n'est pas trouvé dans l'arbre, l'exécution ne doit pas être interrompue mais un message d'erreur doit être affiché sur la sortie standard des erreurs.
- `int ArbreLireLienParentef(FILE *f, int *pIdEnfant, int *pIdParent, char *pParente)` : lit dans le fichier de texte d'identificateur `f` les informations constituant un lien de parenté et stocke l'identifiant de l'enfant à l'adresse `pIdEnfant`, l'identifiant du parent à l'adresse `pIdParent` et la parenté à l'adresse `pParente`. Cette fonction suppose que le lien de parenté est décrit dans le fichier sous la forme d'une ligne constituée de l'identifiant de l'enfant, l'identifiant du parent et le caractère `F` ou `M`, séparés par un espace. Par exemple, le fait que Marthe est la mère de Pierre sera représenté par la ligne suivante :

  ```
  122 19 M
  ```

- `tArbre ArbreLireLienParenteFichier(tArbre Arbre, char Fichier[])` : ajoute à l'arbre `Arbre` les liens de parenté décrits dans le fichier de texte de désignation `Fichier`. Chaque lien de parenté est décrit dans le fichier par une ligne au format décrit pour la fonction `ArbreLireLienParentef`. Cette fonction doit retourner l'arbre modifié ou `NULL` en cas de problème. Le fichier `arbre4.par` correspond à l'exemple cité ci-dessus.

3.2 Test du module `genea`

Créez le fichier source `testgenea.c` contenant la fonction `main` afin de tester les différentes fonctions du module `genea`. Effectuez la compilation séparée grâce à l'utilitaire `make`. Pour cela, écrivez un fichier de description des dépendances `Makefile2` et lancez les compilations en tapant :

```
make -f Makefile2
```

Quand vous aurez mis au point les différentes fonctions du module, modifiez le fichier `testgenea.c` afin de produire un exécutable dont l'appel suivant (n'oubliez pas de vérifier le nombre d'arguments reçus) :

`./testgenea` *fichier-personnes fichier-liens-parente*

crée l'arbre généalogique décrit par les deux fichiers passés en arguments, l'affiche à l'écran et le libère.

Vous pouvez utiliser les données fournies en tapant :

```
./testgenea arbre4.ind arbre4.par
```

pour afficher l'arbre donné en exemple ci-dessus.

Vous pouvez également effectuer des tests avec des arbres plus complexes.

4 Niveau 3 : ajout de fonctionnalités au module `genea`

4.1 Fonctions supplémentaires

L'objectif du niveau 3 est d'ajouter au module `genea` les fonctions suivantes :

- `void ArbreEcrireGV(tArbre Arbre, char Fichier[])` : écrit dans le fichier de texte de désignation `Fichier` l'arbre `Arbre` au format DOT [6] afin de pouvoir le visualiser grâce à l'outil de visua-

6. https://graphviz.org/doc/info/lang.html

lisation de graphe Graphviz[7]. Nous n'utiliserons pas toutes les fonctionnalités du langage DOT. Il suffit d'examiner le contenu du fichier `arbre4.dot` que cette fonction doit créer pour l'arbre donné en exemple au niveau 2. Son contenu doit être le suivant :

```
1  digraph {
2    rankdir="BT";
3
4    node [shape=box,color=blue,fontname="Arial",fontsize=10];
5    122 [label="DUBOURG\nPierre\n15/12/1910"];
6    18 [label="DUBOURG\nJean\n21/07/1866"];
7
8    node [color=green];
9    9 [label="DUBOURG\nCatherine\n21/08/1904"];
10   19 [label="COURBIN\nMarthe\n22/04/1872"];
11
12   edge [dir=none];
13   122 -> 18;
14   122 -> 19;
15   9 -> 18;
16   9 -> 19;
17 }
```

Dans ce fichier :

- la ligne 1 indique qu'il s'agit d'un graphe orienté ;
- la ligne 2 demande que le graphe soit représenté du bas vers le haut ;
- la ligne 4 décrit le style des nœuds du graphe qui vont correspondre aux hommes ;
- les lignes 5 et 6 décrivent les nœuds du graphe représentant les hommes (le premier mot sur la ligne est l'identifiant du nœud qui va ensuite être utilisé pour décrire les arcs) ;
- la ligne 8 modifie le style des nœuds du graphe pour ceux qui correspondent aux femmes ;
- les lignes 9 et 10 décrivent les nœuds du graphe représentant les femmes ;
- la ligne 12 décrit le style des arcs du graphe (ici, pas de flèche) ;
- les lignes de 13 à 16 décrivent les arcs du graphe représentant les liens de parenté ;
- la ligne 17 termine le graphe.

Pour produire un fichier au format PDF de visualisation de l'arbre généalogique donné précédemment en exemple, sur les machines des salles de TP, il suffit de taper la commande suivante :
`cat arbre4.dot | dot -Tpdf > arbre4.pdf`
Le résultat est donné par la figure 3.

- `void ArbreAfficherAscendants(tArbre Arbre, int Identifiant)` : affiche à l'écran l'arbre généalogique ascendant (constitué de tous les ascendants) de la personne d'identifiant `Identifiant` contenu dans l'arbre `Arbre`. Si l'identifiant de la personne n'est pas présent dans l'arbre, l'exécution ne doit pas être interrompue mais un message d'erreur doit être affiché sur la sortie standard des erreurs. Par exemple, l'affichage de l'arbre ascendant de Jean-Louis CHARLOT (identifiant 4), extrait de l'arbre stocké dans les fichiers `arbre10.ind` et `arbre10.par`, doit être :

```
[4] CHARLOT Jean-Louis, M, 09/05/1945
        Père : [8] CHARLOT Jacques, M, 08/04/1900
                Mère : [17] LAFONT Marie, F, 13/10/1869
        Mère : [9] DUBOURG Catherine, F, 21/08/1904
```

7. https://graphviz.org/

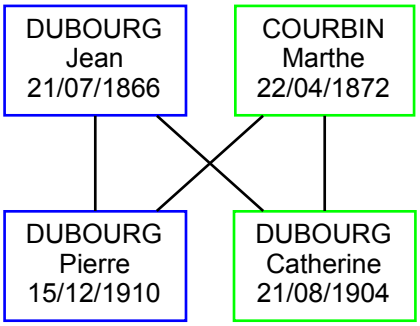

FIGURE 3 – *Visualisation de l'arbre généalogique contenant quatre personnes.*

```
Père : [18] DUBOURG Jean, M, 21/07/1866
Mère : [19] COURBIN Marthe, F, 22/04/1872
```

Les mots `Père` et `Mère` doivent être précédés d'un nombre de tabulations (`'\t'`) égal au niveau dans l'arbre (la génération). Il est conseillé de faire appel à une fonction récursive pour réaliser cet affichage. La fonction `ArbreAfficherAscendants` pourra être constituée de deux étapes :

– rechercher la personne à partir de son identifiant ;
– si elle est présente dans l'arbre, afficher ses ascendants avec le niveau 0.

La fonction récursive consistant à afficher les ascendants d'une personne avec un certain niveau pourra être constituée de trois étapes :

– afficher l'identité de la personne ;
– si la personne a un père dans l'arbre, afficher le nombre de tabulations correspondant au niveau et afficher les ascendants du père en augmentant le niveau ;
– si la personne a une mère dans l'arbre, afficher le nombre de tabulations correspondant au niveau et afficher les ascendants de la mère en augmentant le niveau.

- `void ArbreEcrireAscendantsGV(tArbre Arbre, int Identifiant, char Fichier[])` : écrit au format DOT dans le fichier de texte de désignation `Fichier` l'arbre généalogique ascendant de la personne d'identifiant `Identifiant` contenu dans l'arbre `Arbre`. Il est conseillé de faire appel à une fonction récursive pour réaliser cette écriture. Le contenu du fichier `arbre10.dot` au format DOT de l'arbre généalogique ascendant de Jean-Louis CHARLOT, extrait de l'arbre stocké dans les fichiers `arbre10.ind` et `arbre10.par`, peut être :

```
digraph {
  rankdir="BT";

  node [shape=box,fontname="Arial",fontsize=10];

  edge [dir=none];

  node [color=blue];
  4 [label="CHARLOT\nJean-Louis\n09/05/1945"];

  node [color=blue];
  8 [label="CHARLOT\nJacques\n08/04/1900"];
  4 -> 8;

  node [color=green];
```

```
  17 [label="LAFONT\nMarie\n13/10/1869"];
  8 -> 17;

  node [color=green];
  9 [label="DUBOURG\nCatherine\n21/08/1904"];
  4 -> 9;

  node [color=blue];
  18 [label="DUBOURG\nJean\n21/07/1866"];
  9 -> 18;

  node [color=green];
  19 [label="COURBIN\nMarthe\n22/04/1872"];
  9 -> 19;
}
```

La visualisation de ce résultat obtenu grâce à la command `dot` de Graphviz est présentée sur la figure 4.

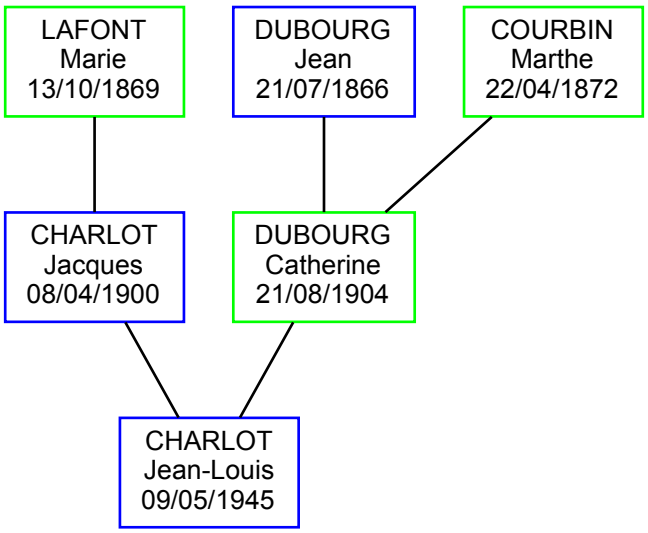

FIGURE 4 – *Visualisation de l'arbre généalogique ascendant de Jean-Louis CHARLOT.*

4.2 Test des nouvelles fonctions

4.2.1 Visualisation d'un arbre généalogique

Créez le fichier source `visuarbre.c` contenant une fonction `main` afin de tester la fonction `ArbreEcrireGV`. Écrivez le fichier de description des dépendances associé `Makefile3` pour effectuer la compilation séparée et produire un exécutable dont l'appel suivant (n'oubliez pas de vérifier le nombre d'arguments reçus) :

`./visuarbre` *fichier-personnes fichier-liens-parente fichier-dot*

écrit au format DOT dans le fichier *fichier-dot* l'arbre généalogique décrit par les fichiers *fichier-personnes* et *fichier-liens-parente*.

Par exemple, les deux commandes :

```
./visuarbre arbre10.ind arbre10.par arbre10.dot
```

```
cat arbre10.dot | dot -Tpdf > arbre10.pdf
```
doivent produire le fichier PDF dont la visualisation est présentée sur la figure 5.

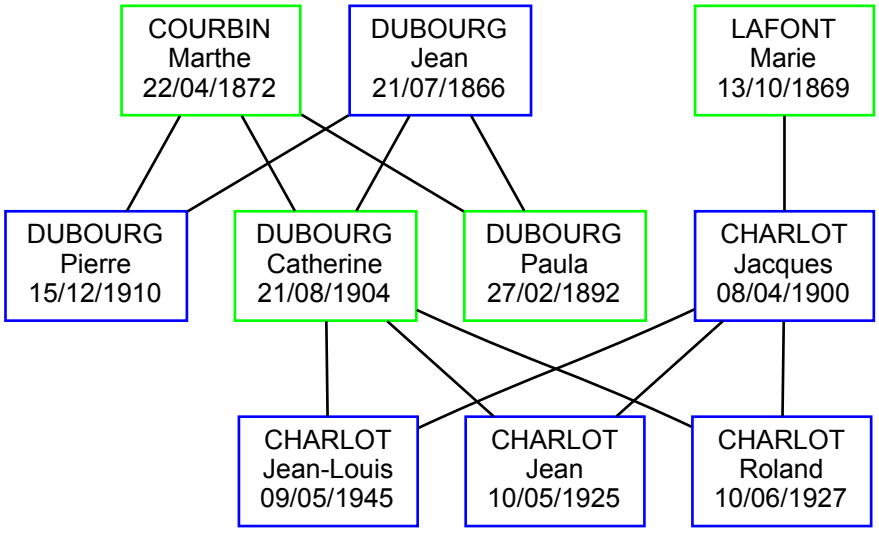

FIGURE 5 – *Visualisation de l'arbre généalogique contenant dix personnes.*

4.2.2 Visualisation d'un arbre généalogique ascendant

Créez le fichier source `visuarbreasc.c` contenant une fonction `main` afin de tester les fonctions `ArbreAfficherAscendants` et `ArbreEcrireAscendantsGV`. Écrivez le fichier de description des dépendances associé `Makefile4` pour effectuer la compilation séparée et produire un exécutable dont l'appel suivant (n'oubliez pas de vérifier le nombre d'arguments reçus) :
`./visuarbreasc` *fichier-personnes fichier-liens-parente identifiant fichier-dot*
affiche à l'écran l'arbre généalogique ascendant de *identifiant* extrait de l'arbre généalogique décrit par les fichiers *fichier-personnes* et *fichier-liens-parente* et écrit au format DOT dans le fichier *fichier-dot* cet arbre généalogique ascendant.