ISEN école d'ingénieurs

Projet de fin d'année
Un site de généalogie

Introduction

Vous allez implémenter un outil capable de générer un ensemble de fichier HTML à partir de données formatées dans un fichier texte.

L'outil de génération doit impérativement être codé en C. Une fois généré, l'ensemble des fichiers HTML ne doit plus être modifié. Le site restera statique.

Le projet contient un ensemble d'exigences minimales à respecter pour débuter (partie 3). Une fois ces exigences implémentées, vous aurez la possibilité de rajouter des fonctionnalités supplémentaires à votre cahier des charges, à définir avec votre groupe (partie 4).

Le projet se divise en deux phases. Dans une première phase d'analyse, vous devrez rédiger en groupe un cahier des charges qui rassemble l'ensemble des fonctionnalités que vous mettrez en œuvre : celles qui sont dans les parties 1 à 3 puis celles que vous voulez ajouter en partie 4. La description des tâches, l'organisation et la répartition de celles-ci font partie intégrante de la note finale. Dans un deuxième temps, implémentez les fonctionnalités et testez-les.

Commencez par vous répartir les tâches pour réussir la partie 3. Décrivez votre planning dans le cahier des charges. Puis organisez-vous pour la partie 4, et complétez le cahier des charges. Adaptez votre cahier des charges pour ne pas être débordés en fin de projet.

1. Architecture minimale

Le projet se décompose à minima en trois parties

- Le noyau : Un programme C qui génère automatiquement un site
- Les scripts : Un ensemble de scripts LINUX permettant d'utiliser simplement le noyau C
- La sortie : Un ensemble de fichiers permettant d'afficher les résultats sous forme de site (local)

Le noyau comprend :

- Un ensemble de fichiers .c et .h,
- Les outils permettant de compiler le projet (makefile / projet Clion)
- Un ou plusieurs fichiers textes au format CSV contenant les attributs des personnes de la généalogie.

Les scripts comprennent :

Un dossier contenant un ou plusieurs scripts permettant de générer simplement le site à partir du noyau.

La sortie comprend :

- Les fichiers HTML générés par le noyau C,
- Des fichiers de style, du code javascript,
- Eventuellement des fichiers templates permettant de faciliter la génération du site

Dans le rendu final, les fichiers HTML générés par le noyau ne sont pas à rendre. Tout le reste doit être rendu dans une archive ZIP qui contiendra un fichier README.md permettant de compiler et d'exécuter le projet.

2. Description du projet

L'objectif est de générer un ensemble de fichiers HTML à partir de données contenues dans un simple fichier texte. Il s'agit de générer un (ou plusieurs) arbre généalogique à partir d'un fichier texte contenant des données sur des personnes. Les fichiers HTML devront être mis en forme (css, javascript).

Description du fichier d'entrée

Le fichier texte sera au format CSV (Comma Separated Values ou données séparées par des virgules). Chaque ligne va contenir les attributs d'une personne. Cette ligne commencera obligatoirement par un nombre entier : l'identifiant unique de chaque personne. Elle sera obligatoirement suivie de deux autres nombres entiers : le premier sera l'identifiant du père de la personne, le deuxième celui de la mère de la personne. Suivront ensuite sur cette même ligne des attributs comme le nom de famille, le prénom, la date de naissance, la région de naissance.

Exemple :

```
1,2,3,Hatton,Andrew,9/2/1801,Yorkshire
2,0,0,Hatton,Steve,25/7/1775,Yorkshire
3,0,0,Vockins,Mary,11/10/1777,West Yorkshire
```

Dans cet exemple Andrew Hatton avec l'identifiant 1 est le fils de 2 : Steve Hatton et de 3 : Mary Vockins, et il est né le 9 février 1801 dans la région Yorkshire.

Notez que lorsqu'un parent est inconnu, on lui attribue automatiquement le numéro 0. Ce numéro 0 correspond à un inconnu que l'on insérera toujours en première ligne du fichier CSV de la manière suivante :

```
0,0,0,-,-,-,-
```

Les parents de l'inconnu sont eux-mêmes inconnus, *sa date de naissance est la plus petite possible*.

Chaque identifiant de personne sera garanti comme unique dans ce fichier.

Le programme C

Le programme à écrire en C doit se décomposer en plusieurs modules. Vous serez guidés dans la suite de ce document pour implémenter la base de chacune de ces modules.

- Le premier module « structures » consiste à structurer l'information en C et à y ajouter plusieurs fonctions pour en simplifier l'utilisation
- Le deuxième module « lecture » consiste à ouvrir et lire le fichier CSV et à remplir les structures avec les données du fichier.
- Le troisième module « avancé » consiste en des fonctions capables d'effectuer des traitements de haut niveau sur les données (par exemple retrouver tous les frères et sœurs d'une personne)

- Le quatrième module «export» regroupe l'ensemble des fonctions qui permettent d'exporter les données sous forme de fichiers HTML
- Le dernier module « menu » propose une interface utilisateur pour répondre à des questions sur la généalogie d'une personne. *Exemple : quel est mon plus lointain ancêtre connu ?*

Les scripts

Nous proposons de regrouper des scripts linux dans un dossier du projet nommé scripts. Nous vous laissons libres d'y ajouter des scripts bash pour lancer votre noyau avec des paramètres.

Pour rappel, pour prendre en compte des paramètres dans un programme C il faut déclarer la fonction main de la façon suivante :

```
int main(int argc, char *argv[])
```

argc désignant le nombre de paramètres et **argv** les paramètres eux-mêmes

Les scripts devront être commentés.

Remarque : cette partie n'est pas obligatoire si vos menus sont dans le noyau C

La sortie

Nous proposons de regrouper les fichiers générés par le noyau dans un dossier export qui sera contenu dans votre projet.

Le dossier export contiendra aussi les fichiers css et javascript pour la mise en page.

A minima, les fichiers HTML devront être constitués de la façon suivante :

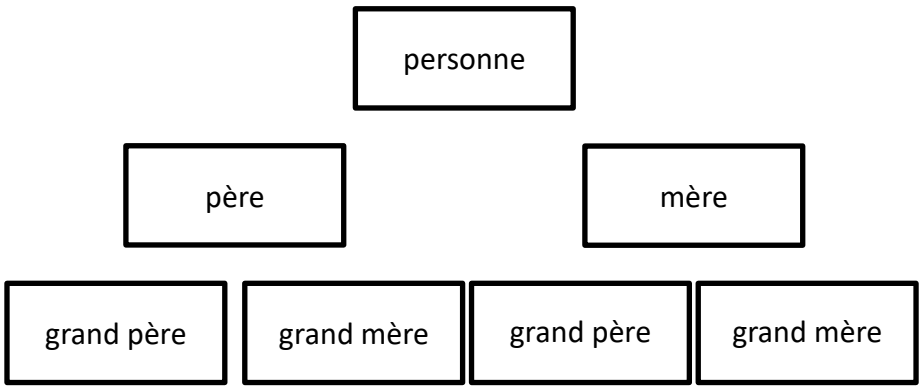
Un fichier HTML par personne contenant :

- Le nom prénom de la personne
- Un tableau contenant un arbre généalogique sur deux générations
- Les noms des personnes contenues dans le tableau devront être encapsulés dans des liens permettant ainsi de naviguer entre différentes fiches de personnes.
- Des informations supplémentaires sur cette personne (date de naissance)

Rien ne doit être ajouté manuellement dans les fichiers HTML générés. C'est au noyau C de tout gérer correctement (entête, lien vers les styles, informations sur les personnes …).

Le style comme l'organisation de l'affichage des données est laissé libre. A vous d'exprimer votre créativité !

Quoiqu'il en soit, l'arbre généalogique prendra l'allure suivante : la personne à considérer est la racine de notre arbre, et ses parents seront pour nous les fils gauche fils droit (~ étrange ~).



Organisation du projet

En regroupant les informations données précédemment, l'arborescence du projet doit être identique à celle-ci :

```
projet_genealogie
      export (dossier contenant la sortie)
      ressources (dossier contenant les csv)
      scripts (dossier script linux)
      fichier1.c
      fichier1.h
      …
      main.c
      makefile ou arborescence CLion classique)
      readme.md
```

Il est important de noter que le rendu de votre projet devra respecter cette organisation. Le dossier export ne devra pas contenir de fichiers générés lors du rendu, mais il est indispensable d'y ajouter vos feuilles de style et le javascript.

3. Démarrer le projet

Il est fortement conseillé de suivre les étapes suivantes pour bien démarrer le projet. Il vous sera possible par la suite de modifier ou de compléter votre projet en fonction du cahier des charges que vous vous serez fixés (partie 4).

Module structures de données

Les données de ce projet sont structurées autour de la notion de personne

Structure personne (obligatoire)

Créer un module de fichiers person.h/.c y insérer la structure suivante :

```
typedef struct Person{
    int id;
    int father_id;
    int mother_id;
    char lastname[20];
    char firstname[20];
    int birthday; int birthmonth; int birthyear;
    char[30] birthzipcode;
```

```
    struct Person * p_father;
    struct Person * p_mother;
}person;
```

Les trois premiers champs sont les identifiants obligatoires. Les quatre champs suivants correspondent au nom de famille, prénom, année de naissance et région de naissance.

Les deux derniers pointeurs seront utilisés plus tard pour relier chaque personne à ses parents (ou à l'inconnu).

Initialisation d'une personne (optionnel)

Définir une fonction **initPerson** qui alloue en mémoire une personne et remplit les champs correspondants.

```
person * initPerson(int id, int father_id, int mother_id, char *lastname,
char *firstname, int birthday, int birthmonth int birthyear, char
*birthzipcode...);
```

Les deux derniers pointeurs resteront NULL pour l'instant.

Population

Créer deux fichiers population.h/.c

Dans ce fichier, nous allons définir une structure permettant d'organiser la mise en mémoire de toutes les personnes de la population (c'est-à-dire celles qui seront importées depuis le fichier CSV)

Vous avez la possibilité d'utiliser :

1. un tableau de pointeurs de personne
2. une table de hachage contenant des pointeurs de personne

Avec un tableau

Le **tableau** est plus simple à mettre en œuvre mais procure moins de souplesse. Dans ce cas, le pointeur de chaque personne devra être stocké dans la case dont l'indice correspond à l'identifiant de cette personne. Le tableau de pointeurs pourra être alloué dynamiquement, pour s'adapter à la taille du fichier à lire.

Crée une fonction de libération de la mémoire qui libère chaque allocation de personne puis le tableau entier.

Avec une table de hachage

Dans le cas d'une **table de hachage**, nous conseillons de choisir une clef égale à l'identifiant de la personne à stocker.

Ecrire une fonction d'insertion d'une nouvelle personne, une fonction de recherche à partir de l'identifiant d'une personne, ainsi qu'une fonction permettant de libérer toute la mémoire allouée : pour chaque personne puis pour la table de hachage.

Voici une proposition de structure pour la table de hachage :

```
typedef struct DataItem {
    person *data;
    int key;
} dataItem;

typedef struct HashTable{
```

```
    int size;
    dataItem** hashArray;
}hashTable;
```

Implémenter une fonction pour le hashcode

```
int hashCode(hashTable t, int key);
```

Proposition de signature pour la fonction d'insertion :

```
void insert(hashTable t, int key, person *data );
```

Lier la population par parenté

Dans les deux cas (tableau et table de hachage)

Ecrire une fonction **`linkPopulation`** qui va initialiser la valeur de chaque pointeur **`p_father`** et **`p_mother`** de chaque personne vers l'adresse mémoire de la bonne personne. Plus aucune valeur de pointeur ne doit rester nulle puisque par défaut l'inconnu est le père ou la mère de chaque personne.

Signature avec une HashTable :

```
void linkPopulation(hashTable t);
```

Nous rappelons que l'identifiant de chaque personne est unique. L'inconnu pointera vers lui-même.

Note : il est possible d'implémenter cette fonction après le module lecture.

Module lecture

Ce module permet de lire un fichier CSV et de remplir une population avec des personnes.

Créer deux fichiers filemanager.h/.c

Implémenter une fonction read_csv

```
hashTable read_csv(int row, int col, const char *filename, const char*
delimiter)
```

row et col représentent respectivement le nombre de lignes et de colonnes dans le fichier (dans notre fichier CSV de base il y a 7 colonnes). **`filename`** est le chemin vers le fichier à lire ; délimiter est le caractère utilisé pour séparer chaque colonne (une virgule, un point-virgule, un espace…).

Pour lire une ligne avec des **`delimiters`**, nous proposons d'utiliser la fonction **`strtok`** de **`string.h`**. Se documenter sur cette fonction avant l'implémentation de la fonction.

Note : Ne pas oublier la gestion de l'ouverture fermeture de fichier, l'allocation de mémoire.

Module avancé

Créer deux fichiers advanced.h/.c

Dans le module avancé vous allez créer des fonctions qui permettent d'organiser et de récupérer des données dans la population. Nous proposons d'implémenter deux fonctions. Il sera possible d'en rajouter (en partie 4).

***Attention** : dans cette partie avancée, une réflexion doit être menée sur la complexité des algorithmes implémentés.*

Fratrie

Une fratrie regroupe l'ensemble des frères et sœurs d'une même famille

Implémenter une fonction qui retourne sous forme de tableau l'ensemble des frères et sœurs d'un même individu. On considère dans cet exercice que les frères et sœurs doivent avoir le même père et la même mère.

```
person ** brotherhood(hashTable t, person *p)
```

Ancêtres

Cette fonctionnalité va permettre de stocker les ancêtres d'une personne sur deux générations dans un tableau. Ce tableau sera ensuite réutilisé dans les fonctionnalités d'export HTML. Dans le cas de base, nous nous limiterons à deux générations. Rien n'interdit d'étendre cette fonction au-delà de deux générations.

Un peu de mathématiques.

Pour commencer, il faut remarquer que dans un arbre binaire, le nombre d'éléments double à chaque génération : 1 individu, 2 parents, 4 grands parents etc. La somme de tous ces individus est égale à 7 soit 8 – 1.

Montrer par récurrence que

$$\forall n > 1, \sum_{i=0}^{n-1} 2^i = 2^n - 1$$

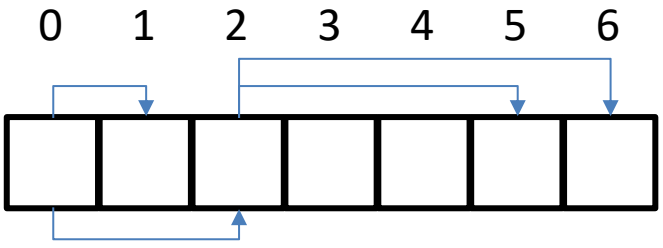
La démonstration sera à inclure dans le rendu intermédiaire.

Pour stocker n générations, il faudra donc 2^{n+1} -1 cases dans le tableau.

Organisation du tableau

On considère qu'au départ, la personne dont on souhaite afficher les ancêtres sera stockée dans la case d'indice 0. Le père sera stocké dans la case 1, la mère dans la case 2. On remarque ensuite qu'une formule toute simple permet de stocker le père et la mère de n'importe quel élément du tableau. Considérons une personne stockée à l'indice n du tableau. Son père sera de ce fait placé à l'indice 2*n+1 et sa mère à l'indice 2*n+2. Dans le cas de deux générations, le grand père maternel sera stocké à l'indice 5 du tableau tandis que la grand-mère maternelle sera stockée à l'indice 6. La case 6 est bien la dernière case de notre tableau à 7 cases.

Le principe est illustré sur la figure suivante :



Le point intéressant avec cette formule est qu'elle nous offre la possibilité détendre le concept à autant de générations que l'on souhaite.

Note : on parle souvent de parents dans ce projet, alors que dans la définition d'un arbre, on utiliserait les termes fils gauche fils droit. C'est un détail important. Ne pas confondre le vocabulaire utilisé en informatique (arbre binaire) et les objets que l'on manipule (une généalogie).

Implémentation

Finalement, implémenter une fonction **ancestorsPersons** qui renvoie un tableau de pointeurs vers les personnes ancêtres de la personne p. La personne p se situera dans la case 0, les parents se trouveront aux indices 2n+1 et 2n+2 de chaque personne n.

```
person ** ancestorsPersons(hashTable t, person *p)
```

Ne pas oublier l'allocation de mémoire, tester votre fonction

Cas de l'inconnu : si les parents d'un individu sont inconnus, votre algorithme devrait mécaniquement remplir les espaces manquant par un pointeur vers la personne inconnue.

Module menu

Ce module est relativement libre. Vous implémenterez un menu qui devra permettre de proposer à l'utilisateur des fonctionnalités internes au noyau.

Remarque : vous pouvez également intégrer vos menus en bash dans la partie script, à vous de choisir le plus pertinent.

Nous conseillons d'implémenter le module menu lors de la partie 4.

Module export HTML

Créer deux fichiers htmlexport.h/.c

L'export en HTML va essentiellement consister à encapsuler des données issues de population et des personnes dans des balises html sous forme de chaîne de caractères. Pour cela, nous vous conseillons :

- De séparer vos fonctions en plusieurs fonctions simples
- De tester ces fonctions régulièrement
- D'utiliser la fonction sprintf (voir plus bas)
- D'imiter le système de buffer de sprintf pour vos propres fonctions

Rappels sur sprintf

Avant de commencer, rappels sur sprintf

```
int sprintf( char * buffer, const char *format, ... );
```

La fonction **sprintf** prend en référence un pointeur vers une chaine de caractère, le **buffer**, qui sera la chaîne modifiée. Le deuxième paramètre, format, est la chaine de caractère que l'on souhaite insérer dans le buffer. Il s'agit du même type de format que celui utilisé par printf : vous pouvez insérer des %d pour inclure des entiers, des %s pour inclure des chaînes de caractère etc. Les derniers paramètres sont justement les nombres, les chaînes de caractères que vous souhaitez inclure.

L'élément clef de cette fonction, c'est son retour de type **int**. Il renvoie le nombre de caractères écrits dans buffer pendant l'appel (il peut aussi renvoyer des codes d'erreurs, voir la doc officielle). C'est ce nombre de caractères qui va vous permettre d'enchaîner facilement plusieurs appels à **sprintf**.

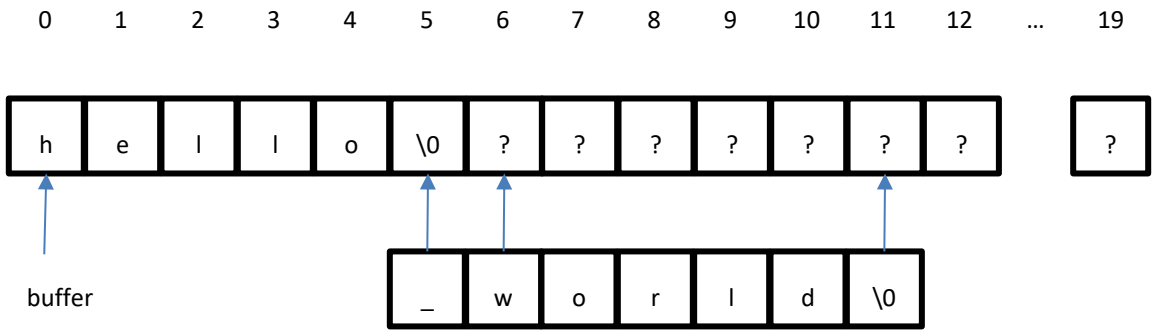
Admettons que l'on veuille écrire « hello world » en deux fois dans un buffer de 20 caractères. Voici un exemple de code:

```
char buffer[20] ;
int nb_characters = 0; // count the number of printed characters
int index = nb_characters; // index indicates the position where it prints
in the buffer
nb_characters += sprintf(buffer+index, "hello");
index = nb_characters; // update current index
nb_characters += sprintf(buffer+index, " world");

printf("%d caracteres ecrits : %s\n", nb_characters, buffer);
```

À la fin de l'exécution, vous verrez apparaître le nombre exact de caractères écrits dans le buffer : 11

Le douzième caractère (celui de la case 11 puisqu'un on commence à 0) sera un \0 pour finir la chaîne. Que s'est-t-il passé en l'écriture de hello et celle de world ? le nombre de caractères renvoyé par **sprintf** est 5. Nous mettons donc à jour l'index à 5 pour se rappeler qu'au prochain appel il faut commencer 5 cases plus loin.



Dans l'exemple ci-dessus, on voit qu'il est très simple de réutiliser la valeur 5 retournée par le premier **sprintf** : c'est aussi la case où est placée le premier \0 qu'il faut supprimer pour continuer la chaîne de caractères (le _ représente un espace).

Se rappeler également que dans le cadre des tableaux et des pointeurs, **buffer** représente l'adresse de **buffer[0]** donc **buffer + 5** représente cette même adresse décalée de 5 cases. Notez que le nombre 20 qui est la taille de départ du buffer n'intervient nulle part dans nos calculs et c'est tant mieux. Il suffit de prendre un buffer assez grand.

L'espace mémoire (que ce soit sur la pile ou sur le tas) peut être fixé (ou alloué) une seule fois.

Vous allez également utiliser le même type de signature pour vos fonctions d'export HTML : un buffer sera passé en paramètre, puis le nombre de caractères sera retourné. Attention, le buffer à passer en paramètre devra déjà être alloué (sur la pile avec buffer[100] ou sur le tas avec un malloc) avant l'appel de fonction.

Titre de la fiche

Implémenter une fonction **titreHTMLPerson** avec la signature suivante :

```
int titreHTMLPerson(char *buffer, person *p)
```

La fonction prend un buffer préalloué en paramètre ainsi qu'un pointeur vers une personne. La fonction doit retourner le nombre exact de caractères écrits.

Voici un exemple de test pour cette fonction

```
person jeanDupont;
sprintf(jeanDupont.firstname, "Jean");
sprintf(jeanDupont.lastname, "Dupont");
//AFFICHAGE d'un element HTML pour 3
char buffer[255];
int m = titreHTMLPerson(buffer, &jeanDupont);
printf("%s %d characters written\n\n", buffer, m);
```

et la sortie attendue :

```
<h2> Dupont, Jean</h2>
 23 characters written
```

Prenez le temps de bien comprendre le mécanisme de ces fonctions, ils vous feront gagner du temps pour la suite.

Entête HTML

Concernant l'entête du fichier HTML, vous pouvez utiliser plusieurs techniques différentes.

- L'une d'entre elle consiste à écrire une chaine de caractère en C puis à écrire cette chaine dans un fichier (implémenter au moins deux fonctions)
- La deuxième consiste à écrire un fichier entête que vous pourrez concaténer à d'autres fichiers contenant des morceaux de HTML.

Il existe d'autres techniques

Faire un choix technique et expliquez clairement dans votre rapport comment votre projet fonctionne.

Ecriture des données dans un fichier

Chaque personne devra avoir son propre fichier HTML, nous conseillons d'utiliser le nom suivant pour chaque fichier : **`[id_person]-fiche.html`**. Pour la personne 2, le fichier se nommera donc **`2-fiche.html`**.

a) Implémenter une fonction **`fichePath`** qui écrit le nom de la fiche dans un **`buffer`**.

```
int fichePath(char * buffer, person *p)
```

Cette fonction doit retourner le nombre de caractères écrits.

b) Implémenter une fonction **`exportPersonToHTML`**

```
void exportPersonToHTML(const hashTable t, person *p, char *path)
```

Cette fonction prend la population, une personne, et le chemin vers le fichier à écrire en paramètre. Elle doit en outre exporter toutes les données voulues au format HTML dans un fichier texte donc l'extension sera .html (**`[id_person]-fiche.html`**).

Au départ, on ne mettra que le titre de la fiche dans le fichier. Vous enrichirez cette fonction au fur et à mesure du projet. Faites des tests simples pour vous assurez que vous avez bien un fichier qui contient de la syntaxe html en sortie dans un premier temps.

Ecrire l'arbre en HTML

Nous proposons dans cet exemple d'écrire un arbre généalogique sur 2 générations dans un tableau html. La première ligne du tableau contient le nom de la personne, la ligne en dessous est divisée en

deux cases et contient le nom de ses parents et la troisième ligne est séparée en 4 et contient le nom de ses grands-parents. Quand une personne est inconnue, la case est vide.

Le résultat obtenu est le suivant.

```
<table>
<tbody>
<tr>  <td colspan="4">ben solo</tr>
<tr>  <td colspan="2">han solo</td>
      <td colspan="2">leia organa</td>
</tr>
<tr>  <td colspan="1"> </td>
      <td colspan="1"> </td>
      <td colspan="1">darth vader</td>
      <td colspan="1">padme amidala</td>
</tr>
</tbody>
</table>
```

Note : pensez à insérer des \n pour aller à ligne pour que le code html soit lisible (par vous)

Implémenter une fonction **printAncestorsToHTML** qui écrit dans un buffer un tableau HTML comparable à celui donné ci-dessus.

```
int printAncestorsToHTML(char *buffer, const hashTable t, person *p);
```

Pour implémenter cette fonction, vous aurez besoin de la fonction **ancestorsPersons** du module avancé. Le premier élément du tableau renvoyé par **ancestorsPersons** doit être placé sur la première ligne. Notez l'organisation des colspans pour organiser votre algorithme.

Ce résultat pourra être enrichi par la suite dans votre projet.

Un lien hypertexte dans chaque case devra permettre de se rendre vers une autre fiche :

```
<td colspan="2"><a href="5-fiche.html">han solo</a></td>
```

Le style du tableau peut être modifié

Si vos algorithmes sont bien écrits, on doit pouvoir ajouter une troisième génération, une quatrième (ce point sera examiné pours les meilleurs projets)

Insérez cette fonction dans la fonction **exportPersonToHTML** après avoir fait une première série de tests.

Une fois la partie 3 – Démarrer votre projet est terminée : Sauvegardez et archivez votre projet. Cela vous permettra :

D'avoir un projet minimal qui fonctionne à présenter

De pouvoir repartir en arrière

4. Suite du projet

Ne commencez cette partie que lorsque la partie 3 est implémentée correctement.

Ici pas d'instruction ciblée. Vous devez trouver des fonctionnalités à améliorer, à rajouter. Bien penser à être en adéquation avec votre cahier des charges (pas plus, pas moins).

Chaque nouvelle fonctionnalité doit être commentée. Son utilité doit être justifié (simplement) en concertation avec son groupe, dans le cahier des charges. Des

fichiers .h et .c peuvent être ajoutés : votre code doit être bien organisé, commenté, compréhensible.

Exemple : « il est difficile de distinguer les cases du tableau de la généalogie. Les cases du tableau sont donc colorées une fois sur deux en bleu et en rouge. Pour mettre en place cette fonctionnalité, une fonction colorier est implémentée dans le code (fichier exportHTML.h). Elle permet de … »

A contrario : on ne doit pas rajouter de code à la va-vite sans concertation avec son groupe ou sans justification.