David PROSPERIN
Riana CHAUVET RAHERIMALALA

13/01/2026

Compte-rendu SMP-TP 5

1. Introduction

L'objectif de ce TP est de nous familiariser avec les arbres binaires. Pour cela, nous allons prendre l'exemple des arbres généalogiques. Dans ce TP, nous allons créer un type `personne` qui nous permettra de créer par la suite les arbres dont nous aurons besoin.

2. Travail réalisé

Question 1 : Construction d'un type `Personne`

Dans cette question, on souhaite créer une structure de données `Personne` qui contiendra le nom, le prénom, la date de naissance et les informations relatives au conjoint de cette personne. On veut aussi créer une fonction `creerPersonne` qui renverra un pointeur vers un élément de `Personne`.
Voici le code définissant la structure :

```
struct Personne
{
   string nom;
   string prenom;
   int naissance;
   int sexe;
   Personne* conjoint;
};
```

La fonction `creerPersonne` prend en paramètre les champs décrits par la structure `Personne`.
Voici les spécifications de cette fonction :

Fonction: creerPersonne(string nom,string prenom,int naissance,int sexe,Personne* conjoint) Paramètres d'entrée : chaînes de caractères nom et prenom, un entier pour la date de naissance et le sexe (1 si homme, 2 si femme), un pointeur vers une autre personne pour le conjoint Paramètres d'entrée/sortie : pointeur sur une Personne Résultat : Personne*

Voici le code de la fonction `creerPersonne` :

```
Personne* creerPersonne(string nom,string prenom,int
naissance,int sexe,Personne* conjoint){
   Personne* nv    = new Personne;
   nv->nom         = nom;
   nv-> prenom     = prenom;
   nv->naissance   = naissance;
   nv->sexe        = sexe;
   nv->conjoint    = conjoint;
   return nv;
}
```

Pour vérifier le bon fonctionnement de cette fonction, nous avons réalisé plusieurs tests :

```
Personne* david = creerPersonne("PROSPERIN","David",2004,1,nullptr);
Personne* riana = creerPersonne("CHAUVET
RAHERIMALALA","Riana",2006,2,nullptr);
```

Ces tests et d'autres nous serviront par ailleurs dans la suite de ce TP.

Question 2 : Mariage

Dans cette question, il s'agit de modifier les champs `conjoint` des deux personnes passées en paramètre afin de référencer le fait que celles-ci entretiennent une relation. Voici les spécifications de la fonction :

Fonction : mariage(Personne* p1, Personne* p2)
Paramètres d'entrée : des pointeurs vers des élément de type Personne p1 et p2
Paramètres d'entrée/sortie : pointeur sur une Personne
Résultat : Modifie le champs conjoint de p1 passée en paramètre en mettant p2 comme conjoint

Voici le code de notre fonction :

```
void mariage(Personne* p1,Personne* p2){
   p1-> conjoint = p2;
}
```

Pour tester notre fonction, nous avons réalisé le test suivant :

```
Personne* tom = creerPersonne("LOMBE","Tom",2005,1,nullptr);
mariage(riana,tom);
```

L'affichage que nous réaliserons à la question 3 va nous permettre de vérifier le bon déroulé de la fonction `mariage`.

Question 3 : Affichage simple

Dans cette question, on souhaite afficher les informations relatives à une personne. Pour être plus efficace dans un premier temps, nous n'avons pas prêté attention à l'accord masculin/ féminin pour les verbes et les noms communs.

Voici les spécifications de notre fonction :

Fonction : affichage(Personne* p)
Paramètres d'entrée : un pointeur vers un élément de Personne p
Paramètres d'entrée/sortie : rien
Résultat : Affiche les informations relatives à une personne.

Pour ce faire, nous avons écrit le code suivant :

```
void affichage (Personne* p){
   if (p->sexe == 1){
       cout << "Monsieur ";
   }
   else{
       cout << "Madame ";
   }
   cout << p-> prenom << " " << p-> nom << endl;

   if (p->sexe == 1)
   {
       cout << "Il";
   } else {
       cout << "Elle";
   }

   cout << " est née en "<< p->naissance << endl;
```

```
    if (p->conjoint != nullptr)
    {
        if (p->sexe == 1)
        {
            cout << "Il";
        } else if (p->sexe == 2) {
            cout << "Elle";
        }
        cout << " est l'épouse de "<< p->conjoint->prenom << " "<<
p->conjoint->nom << endl;
    }
}
```

Pour tester notre fonction, nous avons affiché les données liées à riana et david.

```
affichage(david);
affichage(riana);
```

Voici ce qu'affiche la console.

```
Monsieur David PROSPERIN
Il est née en 2004
Question 2
Madame Riana CHAUVET RAHERIMALALA
Elle est née en 2006
Elle est l'épouse de Tom LOMBE
```

Grâce à ce test, nous avons pu valider notre fonction `affichage` ainsi que notre fonction `mariage`.

Question 4 : Relations de parenté

Dans cette question, il s'agit d'ajouter les champs représentant les parents de la personne. On modifie la structure de données, la fonction `creerPersonne` et la fonction `affichage`.
Voici les différentes lignes ajoutées.

- Dans la structure `Personne`

```
Personne* pere;
Personne* mere;
```

- Dans la fonction `creerPersonne`

```
Personne* creerPersonne(string nom,string prenom,int
naissance,int sexe,Personne* conjoint){
   Personne* nv    = new Personne;
   nv->nom         = nom;
   nv-> prenom     = prenom;
   nv->naissance   = naissance;
   nv->sexe        = sexe;
   nv->conjoint    = conjoint;
   nv->pere        = nullptr;
   nv->mere        = nullptr;
   return nv;
}
```

- Dans la fonction `affichage`

```
if (p->pere == nullptr)
   {
       cout << "   De père INCONNU" << endl;
   } else {
       cout << "   De père " << p->pere->prenom << " " <<
p->pere->nom << endl;
   }
if (p->mere == nullptr)
   {
       cout << "   De mère INCONNUE" << endl;
   } else {
       cout << "   De mère " << p->mere->prenom << " " <<
p->mere->nom << endl;
   }
```

Pour tester nos modifications, nous avons de nouveau affiché david et riana.

```
Monsieur David PROSPERIN
Il est née en 2004
   De père INCONNU
   De mère INCONNUE
Question 2
Madame Riana CHAUVET RAHERIMALALA
Elle est née en 2006
Elle est l'épouse de Tom LOMBE
   De père INCONNU
   De mère INCONNUE
```

Question 5 : Frère et soeur

Dans cette question, on souhaite écrire une fonction qui vérifie si deux personnes sont frères ou sœurs.

Dans un premier temps, nous allons vérifier que les deux personnes sur qui on va appliquer la fonction `sontFrereSoeur` ne sont pas une seule et même personne. Pour cela, on va commencer par écrire la fonction `memePersonne` dont les spécifications sont décrites ci-dessous :

Fonction : memePersonne(Personne* p1,Personne* p2) Paramètres d'entrée : deux pointeurs vers des éléments de Personne Paramètres de sortie : un booléen, vrai s'il s'agit de la même personne, faux sinon

Voici le code de cette fonction :

```
bool memePersonne(Personne* p1,Personne* p2){
   if(p1-> nom == p2->nom && p1->prenom == p2->prenom &&
p1->naissance == p2->naissance && p1->sexe == p2-> sexe){
        return true;
   }
   return false;
}
```

Pour tester notre fonction, nous avons réalisé les tests suivants :

```
memePersonne(riana,riana);
memePersonne(riana,david);
```

Ces tests ont renvoyé les résultats suivants :

```
Question 5
riana et riana sont les mêmes personnes?
true
riana et david sont les mêmes personnes?
false
```

Ces résultats correspondent à ceux attendus.

Maintenant, nous réalisons la fonction `sontFrereSoeur`. Cette fonction vérifiera au préalable que les personnes passées en paramètre ne sont pas les mêmes.
Voici les spécifications de cette fonction :

Fonction : sontFrereSoeur(Personne* p1, Personne* p2) Paramètres d'entrée : deux pointeurs vers des éléments de Personne Paramètres de sortie : un booléen, vrai si les personne sont frère et soeur, faux sinon.

Pour vérifier le bon fonctionnement de notre fonction, nous avons utilisé le jeu de test suivant :

```
Personne* david =
creerPersonne("PROSPERIN","David",2004,1,nullptr);
Personne* riana =
creerPersonne("CHAUVET-RAHERIMALALA","Riana",2006,2,nullptr);
Personne* angelique = creerPersonne("PAROIS", "Angélique", 2005,
2,nullptr);
riana->pere = david;
angelique->pere = david;
```

Nous avons alors réalisé les tests suivant :

```
sontFrereSoeur(riana,angelique);
sontFrereSoeur(riana,david);
```

Ce test nous a renvoyé le résultat suivant :

```
riana et angelique sont soeurs?
true
riana et david sont frère et soeur?
false
```

Ce test nous montre le bon fonctionnement de notre fonction.

Question 6 : Ancêtre

David PROSPERIN
Riana CHAUVET RAHERIMALALA

13/01/2026

Dans cette question, on souhaite déterminer si une personne b est l'ancêtre d'une personne a. Pour ce faire, nous allons programmer une fonction récursive qui va rechercher b dans les branches du père et de la mère de a.
Voici les spécifications de la fonction :

Fonction : estAncetre(Personne* a, Personne* b) Paramètres d'entrée : deux pointeurs vers des éléments de Personne Paramètres de sortie : un booléen, vrai si b est un ancêtre de a, faux sinon.

Voici le code de notre fonction :

```
bool estAncetre (Personne* a, Personne* b)
{
   //Initialisation de flag pour tester par branche
   bool flagEstAncetreMere = false;
   bool flagEstAncetrePere = false;

   //Test si les personnes existent
   if (a == nullptr || b == nullptr)
   {
       return false;
   }
   //Teste si la personne est orpheline
   else if (a->pere == nullptr && a->mere == nullptr)
   {
       return false;
   }
   //Teste si la mère de a est la personne recherchée
   else if (a->pere == nullptr && memePersonne(a->mere,b))
   {
       return true;
   }
   //Teste si le père de a est la personne recherchée
   else if (memePersonne(a->pere,b) && a->mere == nullptr)
   {
       return true;
   }
   else{
       flagEstAncetreMere = estAncetre(a->mere, b);
       flagEstAncetrePere = estAncetre(a->pere, b);
       return (flagEstAncetreMere || flagEstAncetrePere);
   }
```

```
}
```

Pour vérifier le bon fonctionnement, nous avons réalisé différents tests.
Nous avons utilisé le jeu de données suivant :

```
Personne* david =
creerPersonne("PROSPERIN","David",2004,1,nullptr);
Personne* riana =
creerPersonne("CHAUVET-RAHERIMALALA","Riana",2006,2,nullptr);
Personne* marceline = creerPersonne("GLIPARD", "Marceline", 1985,
2,nullptr);
riana->pere = david;
```

```
cout << "David ancêtre de Riana ?" << endl;
cout << std::boolalpha << estAncetre(riana, david) << endl;
cout << "Riana ancêtre de David ?" << endl;
cout << std::boolalpha << estAncetre(david, riana) << endl;
cout << "Riana ancêtre de Marceline ?" << endl;
cout << std::boolalpha << estAncetre(marceline, riana) << endl;
```

Nous avons alors obtenu les résultats suivants :

```
David ancêtre de Riana ?
true
Riana ancêtre de David ?
false
Riana ancêtre de Marceline ?
false
```

Ces résultats montrent que notre fonction respecte bien son cahier des charges.

Question 7 : Générations

Dans cette question, on va chercher à savoir de combien de générations est composé l'arbre généalogique d'une personne, autrement dit, nous allons déterminer la profondeur de l'arbre.
Voici les spécifications de la fonction associée :

Fonction : nbGenerations(Personne* p) Paramètres d'entrée : un pointeur vers un élément de Personne Paramètres de sortie : profondeur de l'arbre de racine *p

Voici le code de notre fonction :

```
int nbGenerations (Personne* p){
    if (p == nullptr){
        return 0;
    }
    else{
        return 1 +
max(nbGenerations(p->pere),nbGenerations(p->mere));
    }
}
```

Pour tester notre fonction, nous avons utiliser les personnes suivantes :

```
Personne* riana =
creerPersonne("CHAUVET-RAHERIMALALA","Riana",2006,2,nullptr);
Personne* marceline = creerPersonne("GLIPARD", "Marceline", 1985,
2,nullptr);
riana->pere = david;
```

Et nous obtenons les résultats suivants :

```
Pour Riana, il y a 2 générations.
Pour Marceline, il y a 1 générations.
```

Ces résultats sont cohérents avec nos attentes.

Question 8 : Taille de l'arbre

Dans cette question, on souhaite écrire une fonction donnant le nombre de membres de l'arbre généalogique d'une personne. Cela revient à déterminer le nombre de noeud de l'arbre.
Voici les spécifications de notre code :

Fonction : nbMembres(Personne* p)
Paramètres d'entrée : un pointeur vers un élément de Personne
Paramètres de sortie : un entier correspondant au nombre de personnes contenues dans l'arbre généalogique

Voici le code de cette fonction :

```
int nbMembres (Personne* p){
    if(p == nullptr){
        return 0;
```

```
    }
    else{
        return 1 + nbMembres(p->pere) + nbMembres(p->mere);
    }
}
```

Nous avons réalisé les tests suivants :

```
cout << "Pour Riana, il y a " << nbMembres(riana) << "
personnes."<< endl;
cout << "Pour Marceline, il y a " << nbMembres(marceline) << "
personnes."<< endl;
```

Nous obtenons les résultats suivants :

```
Pour Riana, il y a 2 personnes.
Pour Marceline, il y a 1 personnes.
```

Ceux-ci correspondent à nos attentes.

Question 9 : Mariage possible

L'objectif de cette fonction est de vérifier si un mariage entre deux personnes est possible. Pour cela, nous devons respecter des règles précises :
— a ne doit pas être un ancêtre de b
— b ne doit pas être un ancêtre de a
— a et b ne doivent pas être frère et/ou sœur

Voici les spécifications de la fonction respectant ce cahier des charges :

Fonction : mariagePossible(Personne* a, Personne* b)
Paramètres d'entrée : deux pointeurs vers des éléments de Personne
Paramètres de sortie : renvoie un booléen, vrai si le mariage est possible et faux sinon

Voici le code de cette fonction :

```
bool mariagePossible(Personne* a, Personne* b)
{
    if (estAncetre(a, b))
    {
        return false;
    } else if (estAncetre(b, a))
    {
```

```
        return false;
    } else if (sontFrereSoeur(b, a))
    {
        return false;
    }
    return true;
}
```

Pour tester notre fonction, nous avons réalisé les tests suivants.

```
cout << "Mariage possible" << endl;
cout << std::boolalpha << mariagePossible(riana, tom) << endl;
cout << "Mariage pas possible" << endl;
cout << std::boolalpha << mariagePossible(riana, angelique) <<
endl;
```

Dans le premier cas, le mariage est possible car riana et tom ne sont pas frère et soeur et ne partagent pas de lien de parenté (l'un n'est pas l'ancêtre de l'autre).
Dans le deuxième cas, riana et angelique ne peuvent pas se marier car elles ont le même père (david).
Voici le résultat des deux tests :

```
Mariage possible
true
Mariage pas possible
false
```

Question 10 : Affichage de l'arbre généalogique

La fonction affichageArbre permet d'afficher l'arbre généalogique complet d'une personne donnée, en remontant récursivement toutes les générations (parents, grands-parents, etc.). Elle constitue ainsi une généralisation de l'affichage de la première génération de parents.

La fonction repose sur un principe récursif : chaque personne affichée appelle à son tour l'affichage de son père et de sa mère, tant que ceux-ci existent.

Tout d'abord, la fonction vérifie que le pointeur passé en paramètre n'est pas nul. Cela évite tout accès mémoire invalide.

Deux cas sont ensuite distingués :

- Cas de base (personne sans parents connus) :

Si la personne ne possède ni père ni mère, elle correspond à la génération la plus ancienne de l'arbre.

La fonction affiche alors uniquement son identité, en précédant son nom par « Monsieur » ou « Madame » selon son sexe.

- Cas récursif (personne ayant au moins un parent) :

La personne courante est affichée avec une indentation afin de visualiser sa position dans l'arbre généalogique.
La fonction appelle ensuite récursivement affichageArbre sur le père, puis sur la mère, ce qui permet d'afficher les générations précédentes de manière hiérarchique.

L'utilisation de décalages (espaces) dans l'affichage permet de mettre en forme visuellement l'arbre, rendant la lecture plus intuitive.

Ainsi, cette fonction parcourt récursivement l'arbre généalogique et affiche l'ensemble des ancêtres d'une personne, génération après génération, jusqu'aux origines connues.

Voici le code de la fonction :

```
void affichageArbre(Personne* personne){
   if (personne != nullptr){
       if (personne->mere == nullptr && personne->pere ==
nullptr){
               if (personne->sexe == 1){
                   cout << "Monsieur ";
               }
               else{
                   cout << "Madame ";
               }
               cout << personne-> prenom << " " << personne-> nom
<< endl;
               cout << " De parents INCONNUS" << endl;
       }
       else{
           if (personne->sexe == 1){
                   cout << "Monsieur ";
           }
           else{
               cout << "Madame ";
           }
           cout << personne-> prenom << " " << personne-> nom <<
endl;
           cout << " De parents (père)" << endl;
           affichageArbre(personne->pere);

           cout << " De parents (mère)" << endl;
           affichageArbre(personne->mere);
```

```
            cout << "   "<< endl;
        }
    }
}
```

Voici le code de test de la fonction :

```
riana->mere = marceline;
marceline->pere = francis;
francis->mere = francine;
francine->pere = marcel;
affichageArbre(riana);
```

Ce code renvoie :

```
Madame Riana CHAUVET-RAHERIMALALA
 De parents (père)
Monsieur David PROSPERIN
 De parents INCONNUS
 De parents (mère)
Madame Marceline GLIPARD
 De parents (père)
Monsieur Francis GLIPARD
 De parents (père)
 De parents (mère)
Madame Francine GLIPARD
 De parents (père)
Monsieur Marcel IPARD
 De parents INCONNUS
 De parents (mère)
```

L'affichage n'est pas le meilleur mais nous ne sommes pas parvenu à allier bel affichage et récursivité.

Question 11 : Gestion des fichiers

L'objectif de cet exercice est de permettre la sauvegarde puis la relecture d'un arbre généalogique dans un fichier texte, en respectant le principe suivant :
chaque personne est identifiée par un identifiant unique (id), et les relations familiales (père, mère, conjoint) sont enregistrées sous forme de références par identifiant, et non par des pointeurs mémoire.

La solution mise en œuvre repose sur une décomposition du problème en plusieurs fonctions, chacune ayant un rôle précis.

La sauvegarde et la relecture se font en deux grandes étapes :

1. Sauvegarde
    - Parcours de l'arbre généalogique

- Transformation de l'arbre en une liste linéaire de personnes
- Écriture de chaque personne dans un fichier texte selon un format défini

2. Relecture
   - Lecture du fichier ligne par ligne
   - Création de toutes les personnes sans liaisons familiales
   - Reconstruction de l'arbre en remplaçant les identifiants par des pointeurs vers les bonnes personnes

Chaque personne est sauvegardée sur une ligne, avec les champs suivants séparés par des espaces :

<id> <prenom> <nom> <naissance> <sexe> <id_pere> <id_mere> <id_conjoint>

- id : identifiant unique de la personne
- prenom, nom : chaînes de caractères
- naissance : année de naissance
- sexe : codé par un entier
- id_pere, id_mere, id_conjoint :
  - contiennent l'identifiant de la personne correspondante
  - valent -1 si la relation n'existe pas

vectorisationArbre

Spécification de la fonction :

Fonction : vectorisationArbre(vector<Personne>& ps, Personne* p) Paramètres d'entrée : un pointeur vers un élément de Personne Paramètres de sortie : un vecteur de Personne, contenant l'ensemble des personnes de l'arbre généalogique

Cette fonction permet de parcourir récursivement l'arbre généalogique à partir d'une personne donnée et de stocker toutes les personnes rencontrées dans un vecteur.

Elle transforme ainsi une structure arborescente en une structure linéaire, plus simple à sauvegarder dans un fichier.

```
void vectorisationArbre(vector<Personne>& ps, Personne *p)

{

    if (p != nullptr) {

        ps.push_back(*p);
```

```
        vectorisationArbre(ps, p->mere);

        vectorisationArbre(ps, p->pere);

        vectorisationArbre(ps, p->conjoint);

    }

}
```

Voici le bloc de code qui test la fonction vectorisationArbre :

```
vector<Personne> listeRiana;
    vectorisationArbre(listeRiana, riana);

    assert(memePersonne(&listeRiana[0], riana) &&
           memePersonne(&listeRiana[1], david) &&
           memePersonne(&listeRiana[2] , tom));
```

serialisePersonne

Spécification :

Fonction : serialisePersonne(const Personne* p)
Paramètres d'entrée : un pointeur constant vers un élément de Personne
Retourne : une chaîne de caractères correspondant à la sérialisation de la personne

Cette fonction convertit une personne en une chaîne de caractères respectant le format de sauvegarde défini.

Les pointeurs pere, mere et conjoint ne sont pas écrits directement :
ils sont remplacés par les identifiants uniques correspondants, ou par -1 si la relation est absente.

```
string serialisePersonne(const Personne *p)
{
    ostringstream oss;



    oss << p->id << " " << p->prenom << " " << p->nom << " " << p->naissance << " " <<
p->sexe << " ";
    if (p->pere == nullptr) {
        oss << "-1";
    } else {
        oss << p->pere->id;
    }
    oss << " ";
    if (p->mere == nullptr) {
```

```
        oss << "-1";
    } else {
        oss << p->mere->id;
    }
    oss << " ";
    if (p->conjoint == nullptr) {
        oss << "-1";
    } else {
        oss << p->conjoint->id;
    }


    return oss.str();
}
```

Voici le bloc de code qui test cette fonction :

```
assert(serialisePersonne(david) == "0 David PROSPERIN 2004 1 -1 -1 -1");
    assert(serialisePersonne(riana) == "1 Riana CHAUVET-RAHERIMALALA 2006 2 0 4 2");
    assert(serialisePersonne(tom) == "2 Tom LOMBE 2005 1 -1 -1 -1");
    assert(serialisePersonne(angelique) == "3 Angélique PAROIS 2005 2 0 -1 -1");
```

sauvegarderArbre

Spécification :

Fonction : sauvegarderArbre(string nom_fichier, Personne* p)
Paramètres d'entrée :

- une chaîne de caractères correspondant au nom du fichier
- un pointeur vers la tête d'un arbre généalogique (Personne)

Retourne : rien

Cette fonction est responsable de la sauvegarde complète de l'arbre :

1. Elle appelle vectorisationArbre pour obtenir la liste de toutes les personnes.
2. Elle parcourt cette liste et écrit chaque personne dans le fichier à l'aide de serialisePersonne.

Ainsi, l'intégralité de l'arbre est sauvegardée de manière cohérente et indépendante de la mémoire.

```
int sauvegarderArbre(string nom_fichier, Personne *p)
{
    ofstream fluxFichier(nom_fichier);
    vector<Personne> listeDeArbre;
    vectorisationArbre(listeDeArbre, p);

    if (fluxFichier)
    {
```

```
        for(const Personne& personne : listeDeArbre)
            fluxFichier << serialisePersonne(&personne) << endl;
    } else {
        cout << "Impossible d'ouvrir le fichier : " << nom_fichier << endl;
    }
}
```

Voici le bloc de code qui test cette fonction :

```
sauvegarderArbre("./arbreRiana.txt", riana);

    string contenuArbreRiana = lireFichier("./arbreRiana.txt");
    ostringstream chaineAttendu;
    chaineAttendu << "1 Riana CHAUVET-RAHERIMALALA 2006 2 0 4 2\n";
    chaineAttendu << "0 David PROSPERIN 2004 1 -1 -1 -1\n";
    chaineAttendu << "4 Marceline GLIPARD 1985 2 5 -1 -1\n";
    chaineAttendu << "5 Francis GLIPARD 1930 1 -1 6 -1\n";
    chaineAttendu << "6 Francine GLIPARD 1930 2 7 -1 -1\n";
    chaineAttendu << "7 Marcel IPARD 1930 1 -1 -1 -1\n";
    chaineAttendu << "2 Tom LOMBE 2005 1 -1 -1 -1\n";

    assert(contenuArbreRiana == chaineAttendu.str());
```

deserialisePersonne

Spécification :

Fonction : deserialisePersonne(const string& p)
Paramètres d'entrée :
- une chaîne de caractères contenant les informations sérialisées d'une personne

Retourne :
- un pointeur vers un élément de Personne reconstruit à partir de la chaîne

Cette fonction réalise l'opération inverse de la sérialisation pour une ligne donnée du fichier.

Elle :
- crée dynamiquement une nouvelle personne
- lit les informations de base
- stocke temporairement les identifiants du père, de la mère et du conjoint

À ce stade, les pointeurs familiaux ne sont pas encore reconstruits.

```
Personne* deserialisePersonne(const string& p)
{
    Personne* nouvPersone = new Personne();
    istringstream iss(p);
```

```
    iss >> nouvPersone->id;
    iss >> nouvPersone->prenom;
    iss >> nouvPersone->nom;
    iss >> nouvPersone->naissance;
    iss >> nouvPersone->sexe;

    iss >> nouvPersone->pereId;
    iss >> nouvPersone->mereId;
    iss >> nouvPersone->conjointId;

    return nouvPersone;
}
```

Voici le bloc de code qui test cette fonction :

```
string davidSerialise      = "0 David PROSPERIN 2004 1 -1 -1 -1";
    string rianaSerialise      = "1 Riana CHAUVET-RAHERIMALALA 2006 2 0 -1 2";

    Personne* davidDeserialise   = deserialisePersonne(davidSerialise);
    Personne* rianaDeserialise   = deserialisePersonne(rianaSerialise);

    assert(
        davidDeserialise->id        == 0          &&
        davidDeserialise->prenom    == "David"    &&
        davidDeserialise->nom       == "PROSPERIN" &&
        davidDeserialise->naissance == 2004       &&
        davidDeserialise->sexe      == 1          &&
        davidDeserialise->pereId    == -1         &&
        davidDeserialise->mereId    == -1         &&
        davidDeserialise->conjointId== -1
    );

    assert(
        rianaDeserialise->id        == 1                  &&
        rianaDeserialise->prenom    == "Riana"            &&
        rianaDeserialise->nom       == "CHAUVET-RAHERIMALALA" &&
        rianaDeserialise->naissance == 2006               &&
        rianaDeserialise->sexe      == 2                  &&
        rianaDeserialise->pereId    == 0                  &&
        rianaDeserialise->mereId    == -1                 &&
        rianaDeserialise->conjointId== 2
    );
```

chercherPersonneParId

Spécification :

Fonction : chercherPersonneParId(const vector<Personne*>& ps, const int id)

Paramètres d'entrée :

- un vecteur de pointeurs vers des éléments de Personne
- un entier correspondant à l'identifiant recherché

Paramètres d'entrée/sortie :

- un pointeur vers l'élément de Personne correspondant à l'identifiant, ou nullptr s'il n'existe pas

Cette fonction parcourt un vecteur de personnes et retourne celle dont l'identifiant correspond à l'id recherché.

Elle est utilisée lors de la reconstruction de l'arbre pour transformer les identifiants stockés en pointeurs vers les bonnes personnes.

```
Personne* chercherPersonneParId(const vector<Personne*>& ps, const int id)
{
    for (Personne* p : ps) {
        if (p != nullptr && p->id == id) {
            return p;
        }
    }
    return nullptr;
}
```

Voici le bloc de code qui test cette fonction :

```
vector<Personne*> listeRiana2 {david, riana, tom};

    // La personne est dans le vector
    Personne* davidPtr = chercherPersonneParId(listeRiana2, 0);

    assert(davidPtr == listeRiana2[0] && "david est en tête de liste");

    // La personne n'est pas dans le vector
    Personne* marcelinePtr = chercherPersonneParId(listeRiana2, 4);

    assert(marcelinePtr == nullptr && "marceline n'est pas présente dans le vector listeRiana2");
```

chargerArbre

Spécification :

Fonction : chargerArbre(const string& nomFichier)

Paramètres d'entrée :

- une chaîne de caractères correspondant au nom du fichier à charger

Paramètres d'entrée/sortie :

- un pointeur vers la tête de l'arbre généalogique reconstruit, ou nullptr en cas d'échec

Cette fonction permet la relecture complète de l'arbre généalogique :

1. Lecture du fichier ligne par ligne et création de toutes les personnes à l'aide de deserialisePersonne
2. Stockage de ces personnes dans un vecteur
3. Reconstruction des liens familiaux (père, mère, conjoint) en remplaçant les identifiants par des pointeurs grâce à chercherPersonneParId
4. Retour de la tête de l'arbre (première personne du fichier)

Cette méthode garantit que toutes les personnes existent avant la création des relations, comme exigé par l'énoncé.

```
Personne* chargerArbre(const string& nomFichier)
{
   ifstream fichier(nomFichier);
   string ligne;

   vector<Personne*> listePersonnes;

   if (!fichier) {
      cout << "Chargement impossible du fichier : " << nomFichier << endl;
      return nullptr;
   }

   while (getline(fichier, ligne)) {
      Personne* p = deserialisePersonne(ligne);
      listePersonnes.push_back(p);
   }

   for (Personne* p : listePersonnes) {
      p->pere     = chercherPersonneParId(listePersonnes, p->pereId);
      p->mere     = chercherPersonneParId(listePersonnes, p->mereId);
      p->conjoint = chercherPersonneParId(listePersonnes, p->conjointId);
   }

   return listePersonnes.empty() ? nullptr : listePersonnes[0];
}
```

Voici le bloc de code qui test cette fonction :

```
Personne* tete = chargerArbre("./arbreRiana.txt");

   assert(
      tete->prenom    == "Riana"               &&
      tete->nom       == "CHAUVET-RAHERIMALALA" &&
      tete->naissance == 2006                  &&
      tete->sexe      == 2                     &&

      tete->pere->prenom    == "David"         &&
      tete->pere->nom       == "PROSPERIN"     &&
      tete->pere->sexe      == 1               &&
      tete->pere->naissance == 2004            &&

      tete->conjoint->prenom    == "Tom"       &&
      tete->conjoint->nom       == "LOMBE"     &&
      tete->conjoint->sexe      == 1           &&
      tete->conjoint->naissance == 2005
      );
```