Structures de Données et Algorithmes II - Projet

Yannick LE CAM - Kevin ELLWANGER

1 Bases :

Le projet consiste en la création d'un arbre généalogique d'individus. L'arbre en question se repose sur ces structures suivantes :

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

#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;
```

On peut voir le typedef de la structure date, qui est 3 unsigned short correspondant au jour, mois et année. Dans le but de son implémentation le besoin de créer une fonction création date est survenu 2 types de cette fonction ont été réalisés.

Le premier date creationDateVide(); ne prenant aucun paramètre et retournant une date vide, c'est-à-dire tous les attributs initialisés à 0, en effet pour un individu toujours en vie mais ayant besoin d'une date deces ils nous fallait trouver une alternative.
Puis le second date creationDate(unsigned short jour, unsigned short mois, unsigned short annee); prenant en paramètre 3 unsigned short pour créer tout simplement une date avec.

Constructeurs :

Tout d'abord la construction de l'arbre se reposant principalement sur deux fonctions que sont les constructeurs : void genealogieInit(Genealogie *g) et
ident adj(Genelogie *g, char *s, ident p, ident m, date n, date d);

Leur importance est capitale car c'est sur la façon de leur fonctionnement que découlent celui de toutes les autres fonctions.

Première erreur de ce côté, pour la création de genealogieInit. Nous étions d'abord partie vers l'allocation dynamique d'un type Genealogie. Ceci a été vite corrigé lorsqu'on a remarqué que le sujet montré dans les exemples l'utilisation de l'adresse de g et non pas un pointeur de type Genealogie donc il nous a simplement fallu retirer la ligne de l'allocation dynamique car elle n'avait pas lieu d'être. Ce qui nous donne le changement comme suit :

void genealogieInit(Genealogie* g)
{
g=(Genealogie*)malloc(sizeof(Genealogie));
g->tab=(Individu*)malloc(sizeof(Individu)*
SIZE_TAB_INIT);
g->id_cur=1;
g->taille_max_tab=SIZE_TAB_INIT;
return;
}
	→	void genealogieInit(Genealogie* g) {
		
		g->tab=(Individu*)malloc(sizeof(Individu)*
		SIZE_TAB_INIT);
		g->id_cur=1;
		g->taille_max_tab=SIZE_TAB_INIT;
		return;
		}

Avant toutes choses, le parti pris à été de démarrer l'ident à 1 en raison du fait que le 0 est pour les personnes "n'existant pas".

La fonction adj est une très grosse partie de ce projet, du fait que le tableau soit en allocation dynamique il faut passer par un realloc de plus penser à tous les cas et au fait de ne manquer aucune affectation / Réaffectation. Exemple dans le cas ou l'on affecte un enfant il faut alors mettre à jour la variable fils aîné des parents pour qu'elle coïncide.

Deux préconditions pour la fonctions adj :

Précondition 1 Impossible d'ajouter un Individu avec un nom vide.
Précondition 2 On ne peut pas ajouter une personne "identique" (avec le même nom est la même date de naissance. Rapport à la question 5)

Un regret malheureusement remarqué trop tard a été celui pour la fonction ajout cadet. En effet l'or de l'adjonction il nous faut vérifier la place de la personne ajouté dans sa fratrie pour ça nous avons besoin d'une fonction ajoutCadet qui prend en paramètre deux ident x et y et qui par récursivité parcours la fratrie pour arrivé au cadet actuel. Le regret ici fut de l'avoir implémenté en utilisant des pointeurs d'individu pour parcourir l'arbre ce faisant on est obligé de passer par un get créé pour l'occasion alors quand gardant l'ident pour parcourir l'arbre il aurait été plus simple ou du moins, moins fastidieux.

Pour la fonction get elle prend un ident en paramètre pour et retourne un pointeur d'individu correspondant à l'ident donné. Elle possède deux précondition, il faut que l'ident soit différent de 0 car comme dit plus tôt le partie pris à été de prendre 0 pour les individus qui "n'existent pas".

La complication ayant été remarquée que trop tard d'autre fonction on déjà été implémenté avec cette version et pour corriger cela il aurait donc fallu revoir plusieurs fonctions.

Une autre fonction créée pour l'adjonction à l'arbre fut plusVieux prenant en paramètre l'arbre g, et deux ident x et y. Comme elle le laisse sous-entendre la fonction renvoie donc un booléen si x est plus vieux que y. Elle nous permet donc de comparer les dates et donc de classer les différents individus dans l'ordre.

2 Lien de parenté :

Ici rien de bien compliqué dans le cas des deux fonctions :

```
bool freres_soeurs(Genealogie *g, ident x, ident y)
bool cousins(Genealogie *g, ident x, ident y)
```

Le fait d'avoir x et y une simple comparaison suffit. Pour freres_et_soeurs comme il n'y a pas de famille recomposée il suffit simplement de comparer si un parent est le même pour les deux soit : ind_X->mere==ind_Y->mere (ind_X et ind_Y étant respectivement individu x et l'individu y récupéré dans l'arbre généalogique g avec une fonction get(g, x) qui retourne un individu i. La fonction get a été préalablement défini elle permet de récupérer un individu i dans l'arbre g avec l'ident x.

3 Affichages :

En premier lieu pour une lecture plus clarifiée, une fonction `void affichage_current(Genealogie *g)` a été implémentée en vue d'afficher l'ensemble de l'arbre de l'ident 1 à celui courrant avec un simple parcours à l'aide d'une boucle for.

Affichage frères et sœurs:

Pour l'affichage des frères et sœurs il consiste simplement en le fait de partir de l'aîné et de parcourir la fratrie via les cadets successifs de même pour celui d'affichage enfant ou il suffisait de partir d'un parent et de parcourir la descendance via l'aîné et ses cadets successifs.

Affichage cousins:

Le plus compliqué était l'affichage de cousins. La difficulté résidait dans le fait de devoir remonter chaque parent pour ensuite parcourir toutes leurs fratrie en partant de l'aîné de cette dernière, il faut alors continuer le parcours de la fratrie ce qui nous donne pour parcourir l'ensemble :

- Individu x
  - Père
    - Grand-père (père du père)/Grand-mère(mère du père)
      - fils/fille aîné
        - cadet (tant que différent du Père)*
      - fils/fille aîné (les cousins)

Et cela pour la mère également et * pour chaque fils et fille du grand-père/de la grand-mère. La démultiplication des branches de recherches alourdit l'algorithme en raison du fait que pour ne rater aucun cas l'on doit tous les vérifier.

Affichage oncles :

L'affichage suivant concernant les oncles est similaire au précédent. En effet pour afficher les cousins nous devions déjà parcourir les oncles et tantes en reprenant un procédé similaire mais en s'arrêtant une étape avant, soit avant la descente vers les fils/filles, L'on retrouve le parcours souhaité vers les oncles.

Les affichages sont tous compatibles avec le fait que l'individu puisse ne pas avoir de parents néanmoins, dans ce cas l'affichage oncles et celui des frères et sœurs ne pourra afficher que les cadets à partir de l'individu donné en paramètre.

Qui plus est dans le cas ou il n'y a ni grand-mère ou grand-père l'on ne pourra parcourir l'arbre qu'à partir du parent donné ce faisant impossible d'être sûr que le parent donné soit l'aîné de sa fratrie. Ceci nous faisant parcourir uniquement les frères et sœurs plus jeunes que le parent perçu.

4 Parcours de l'arbre généalogique :

Le parcours de l'arbre à demander plusieurs fonctions annexes en vue de simplifier le code, le rendre plus lisible et plus ordonné.

```
bool ancetre(Genealogie *g, ident x, ident y);
bool ancetreCommun(Genealogie *g, ident x, ident y);
```

Ces deux fonctions ci-dessus on vu une version récursive d'elle être implémenté. En effet le parcours de l'arbre pour les ancêtres demande à ce que le remonte chaque branche une à une pour, dans la première fonction, vérifier dans les ancêtres de y si un individu correspond à x et dans la seconde trouver dans les ancêtres de x et y si l'un est en commun. D'où l'implémentation récursive car chaque parcours devra être répété pour autant d'ancêtre qu'il y a.

Les trois fonctions suivantes :

```
ident plus_ancien(Genealogie *g, ident x);
ident affiche_parente(Genealogie *g, ident x);
ident affiche_descendance(Genealogie *g, ident x);
```

La fonction plus_ancien fait appel à une fonction créer pour comparer deux individu avec deux ident donnée et renvoie vrai si x est plusVieux que y. Comme plus_ancien est une succession de comparaison il faut donc faire appel à plusVieux.

Les fonctions affiche_parente et affiche_descendance font toutes deux appels à des fonctions annexes du même style :

```
ident affiche_ind_hauteur(Genealogie* g,ident x,int  hauteur)
ident affiche_ind_hauteur_Descendance(Genealogie* g,ident x,int hauteur)
```

Le premier affiche la parenté et le second la descendance avec le même mécanisme l'un remonte et l'autre descend. Dans les deux cas les fonctions demande de retourner un type ident hors on ne sait aucunement pourquoi ou quel ident retourner parmi l'ensemble de ceux trouvé dans la descendance ou de la parenté de x donc par défaut nous retournons l'ident donné en paramètre.

5 Fusion d'arbres généalogiques :

Remarque : dans le cadre d'une fusion de deux arbres, il est logique que les deux arbres aient au moins un ancêtre commun, car dans le cas contraire, ceci se rapprocherait plus d'un recueil d'arbre généalogique plutôt que d'un arbre généalogique unique. Néanmoins, notre implémentation n'interdit pas la fusion de deux arbres totalement distincts. Ce faisant, le fait de ne fusionner que des arbres ayant au minimum un individu en commun est une précondition implicite.

Précondition 1 Les arbres à fusionner ne doivent pas être vides.
Précondition 2 Aucune date de naissance ne doit être null/égale à 0.

La première précondition est évidente, mais pour la seconde elle existe car lorsque l'on parcourt les deux arbres en vue de les fusionner. Les dates de naissance des individus des deux arbres sont mises à zéro pour s'assurer que l'individu a déjà été traité et donc pour ainsi éviter de le traiter une seconde fois.

Deux fonctions ont été créées pour la fusion :

```
void genealogieFusion(Genealogie * res, Genealogie *a1, Genealogie *a2);
void genealogieFusion_aux(Genealogie *res, Genealogie *g, ident a_ajouter);
```

Le premier fait le travail de sélection de l'ident à insérer. Il parcourt les arbres du plus vieux pour arriver aux plus jeunes. A ce moment nous utilisons à nouveau la fonction plusVieux mais décidons de la modifier en vue qu'elle n'utilise plus un pointeur sur un Individu mais l'ident. Ceci étant elle nous a amené à devoir revoir l'ensemble des fonctions utilisant la fonction plusVieux et les adapter au nouveau format de la fonction.

Le second genealogieFusion_aux est utilisé pour l'insertion de l'ident, trouvé par la première fonction de l'arbre g mis en paramètre, dans l'arbre res. L'arbre res qui est l'arbre résultant de la fonction des deux arbres donnés.

Une troisième précondition survient à ce moment, une précondition pour laquelle nous ne trouvons aucun moyen de la traiter en interne. Cette précondition est le fait que l'arbre res doit être initialisé avant. Dans le cas contraire, si nous devions initialiser l'arbre res dans la/les fonction(s) fusion(s) ce serait alors une variable local qui serait détruit à la fin de la fonction. Une solution aurait pu être utilisée pour traiter ce cas, ce serait de faire en

sorte que la fonction fusion renvoie un type Genealogie il nous suffirait donc de retourner l'arbre résultat pour le conserver. Mais en effet la fonction fusion demandée ne renvoie rien (void) de ce fait cette implémentation pour traiter ce cas n'est donc pas possible.

Encore un dernier point, notre fonction fusion met à zéro l'ensemble des dates de naissance de l'arbre a et b placés en paramètres, pour comme dit précédemment s'assurer qu'un individu est déjà traité. C'est un choix arbitraire d'implémentation. Avec celle-ci les arbres a et b deviennent individuellement inexploitables mais comme leurs fusions contiennent toutes les données des deux arbres nous avons donc décidé de le garder ainsi car rien ne stipulé de devoir conserver les deux arbres fusionnés intacte. Ce choix a été fait en raison du fait qu'il était plus facile à implémenter de la sorte.