GENET Martin
L2 Informatique CMI (Image)

PROJET DE STRUCTURE DE DONNEES ET ALGORITHME S3

SOMMAIRE

1. Ensembles

Pour implémenter les ensembles non-triés et triés, j'ai choisi de les séparer en deux couples de fichier header et source pour pouvoir simplement utilisé l'une ou l'autre des implémentation dans les tests par la suite. Les fichiers header « ensemble.h » et « ensembletrie.h » sont toutefois casiment identiques.

1.1. Structures

Pour implémenter les ensembles finis, j'ai commencé par définir un type *Entier* comme un simple *typedef int Entier;* afin de pouvoir changer simplement ce type dans tout le reste du projet. De même, j'ai défini un type *Bool* comme une énumération *{FALSE, TRUE}* que je ré-utilise tout au long du projet.

La structure s_e*nsemble* à proprement parler contient un tableau dynamique d'éléments de type *Entier* ainsi que deux entiers *taille_max* et *nbr_elmt*, respectivement le nombre d'entiers que peut contenir au maximum le tableau et le nombre d'emplacement actuellement utilisés. Le type *Ensemble* correspond à un pointeur sur une structure *s_ensemble* afin d'alléger le passage en argument pour les fonctions.

```
typedef struct s_ensemble
{
	Entier *tab;
	int taille_max;
	int nbr_elmt;
} SEnsemble, *Ensemble;
```

1.2. Opérations sur les ensembles

- *Ensemble ensembleNouv();*

Cette fonction initialise une structure *s_ensemble* en allouant l'espace mémoire nécessaire et renvoie cette adresse. Le champ *taille_max* est initialisé à une valeur prédéfinie *TAILLE_MAX* (1024 par exemple) et le champ *nbr_elmt* est initialisé à 0. Le champ *tab* est alloué avec la taille du type *Entier* et le nombre *taille_max*.

- *void delEnsemble(Ensemble e);*

Cette fonction permet de libérer les espaces mémoires alloués dynamiquement pour le type *Ensemble*, à savoir le champ *tab* de la structure *s_ensemble* et le pointeur sur cette structure. On vérifiera d'abord que le pointeur fourni en argument n'est pas *NULL*, autrement dit qu'il correspond bien à un ensemble initialisé correctement. On fera toujours cette vérification pour les fonction prenant un pointeur de type *Ensemble* en argument.

- *int rech(Entier x, Ensemble e, int rang);*

J'ai choisi, avant d'implémenter la fonction testant l'appartenance d'un élément à un ensemble, de me mettre à disposition une fonction auxiliaire recherchant l'indice d'un élément dans un ensemble, me disant que ce serait sûrement très utile par la suite. Cette fonction récursive retourne donc -1 si l'élément *x* donné en argument n'est pas trouvé dans l'ensemble *e* au terme de la récursion (ou si *e* n'est pas initialisé) et *rang* sinon (l'appel initial étant *rech(x, e, 0)*).

Pour un ensemble non-trié la fonction consiste dans un simple parcours du tableau avec un appel récursif où *rang* est incrémenté à chaque fois jusqu'à ce que l'on rencontre l'élément *x* (on retourne alors *rang*) ou que *rang* soit supérieur au nombre d'éléments de l'ensemble (on retourne alors -1).

Pour un ensemble trié (par ordre croissant), l'optimisation consiste à tester non seulement si *rang* est supérieur au nombre d'éléments mais aussi si l'élément à l'indice *rang* est supérieur à *x*, auquel cas il est inutile de continuer la récursion et on retourne -1.

- *Bool element(Entier x, Ensemble e);*

Pour tester si un élément appartient ou non à un ensemble, il suffit de tester le retour de l'appel *rech(x, e, 0)* : si la fonction renvoie -1, on renvoie *FALSE*, sinon on renvoie *TRUE*.

- *int cardinal(Ensemble e);*

Pour connaître le cardinal d'un ensemble on se contente de consulter le champs *nbr_elmt* de la structure *s_ensemble* donnée par *e*. On considère que l'appel de la fonction *cardinal* sur un ensemble non initialisé est une erreur, donc si le pointeur est *NULL* on retourne -1.

- *Ensemble adj(Entier x, Ensemble e);*

Pour adjoindre l'élément *x* à l'ensemble *e* la fonction d'adjonction commence par tester le cas d'erreur d'un ensemble non-initialisé pour lequel on retourne *NULL*, puis par tester si l'élément appartient déjà à l'ensemble, auquel cas on retourne *e*. J'avais choisi au départ de retourner *NULL* dans ce cas, mais je trouvais alors la tentative d'ajout d'un élément déjà présent dans l'ensemble trop punitive.

Ensuite un test vérifie que la taille maximum de l'ensemble n'est pas atteinte, sinon on réalloue le tableau dynamique et on met à jour la nouvelle taille maximum avec le champs *taille_max*.

Dans le cas d'un ensemble non-trié, on se contente ensuite d'ajouter *x* à la fin du tableau de l'ensemble et d'incrémenter le champs *nbr_elmt*, après quoi on retourne *e* dont la structure a été modifiée.

Dans le cas d'un ensemble trié, on effectue un parcours du tableau pour trouver l'emplacement permettant de conserver l'ordre de tri, puis on décale tous les éléments du tableau d'un cran vers la fin du tableau pour insérer *x* à la bonne position. Après cette opération, on incrémente *nbr_elmt* et on retourne *e*.

- *Ensemble sup(Entier x, Ensemble e);*

De manière similaire à l'adjonction l'opération de suppression d'un élément débute par le test du cas d'erreur d'un ensemble non-initialisé conduisant au retour de *NULL* en casde succès puis le test d'appartenance de *x* à *e*, qui conduit à retourner *e* en cas d'échec.

Si *x* appartient bien à l'ensemble, on se place à la position donnée par *rech(x, e, 0)* dans le tableau puis on effectue un décallage de tous les éléments du tableau d'un cran vers cette position tout en veillant à décrémenter le champs *nbr_elmt*. Cette opération ne modifiant pas l'ordre de tri du tableau, elle est la même pour les ensembles triés et non-triés.

- *Entier min(Ensemble e) ;* et *Entier max(Ensemble e);*

Dans le cas des ensembles triés, la fonction de minimum/maximum consiste simplement dans le test de la validité de l'ensemble (un ensemble vide est considéré comme un cas d'erreur entrainant l'arrêt du programme) puis le retour du premier (resp. dernier) élément du tableau.

Les fonctions *min* et *max* utilisent une fonction auxiliaire récursive (*__min_rec* et *__max_rec*) dans le cas des ensembles non-triés. Après avoir vérifié que le pointeur sur ensemble fourni en argument n'est pas *NULL* et que l'ensemble n'est pas vide, on initialise un minimum (resp. maximum) avec la valeur du premier élément du tableau, puis on appelle la fonction auxiliaire correspondante et on retourne le minimum/maximum ainsi calculé.

- *Entier __min_rec(Ensemble e, int rang, Entier min);* et *Entier __max_rec(Ensemble e, int rang, Entier max);*

Ces fonctions testent si le rang donné en argument dépasse le nombre d'éléments du tableau, auquel cas on retourne l'entier *min/max* donné en argument. Ensuite, on teste si l'élément d'indice *rang* est inférieur/supérieur à *min/max*. Si c'est le cas, *min/max* prends sa valeur. Ensuite on rappelle la fonction récursive avec *rang+1*. Les arguments pour l'appel initial sont donc l'ensemble *e*, le rang 1 et l'élément d'indice 0 du tableau.

- *void afficheEnsemble(Ensemble e);*

Pour des raisons évidentes de confort j'ai mis en place une fonction d'affichage parcourant les éléments valides du tableau de l'ensemble fourni en argument. Si l'ensemble est vide, un message spécifique l'indique.

- *Ensemble copie(Ensemble e);*

Pour les fonctions d'union et d'intersection que je présente plus loin, j'ai eu besoin d'une fonction créant un nouvel ensemble initialisé avec les valeurs d'un autre ensemble donné en argument. Il n'y a pas grand-chose de plus à expliquer pour cette fonction mais elle devait être mentionnée.

- *Ensemble unionEnsemble(Ensemble e, Ensemble f);*

La fonction *unionEnsemble* utilise, dans le cas des ensembles non-triés comme triés, une fonction auxiliaire récursive *Ensemble __unionEnsemble_rec*, dont les arguments et le fonctionnement changent selon que l'on manipule des ensemble triés ou non.. Après avoir vérifié les préconditions (*e* et *f* initialisés), on initialise un nouvel ensemble pour l'union de *e* et *f* qui est après rempli par l'appel de *__unionEsnemble_rec*, puis renvoyé.

Cas non-trié : *Ensemble __unionEnsemble_rec(Ensemble e, Ensemble f, int rang, Ensemble u);*

L'appel initial de la fonction est *__unionEnsemble_rec(e, f, 0, u)* où *u* est initialisé avec les éléments de *e* (via la fonction *copie*). Si le rang est inférieur au cardinal de *f*, on extrait l'élément *x* à cet indice dans *f* puis on vérifie qu'il n'appartienne pas *déjà* à *e*. Si cette condition est vérifiée on l'adjoint à *u* et on rappelle la fonction avec *rang+1*. Si le rang est supérieur ou égal au cardinal de *f* alors on renvoit *u*.

Cas trié : *Ensemble __unionEnsemble_rec(Ensemble e, int ei, Ensemble f, int fi, Ensemble u);*

L'appel initial de la fonction est *__unionEnsemble_rec(e, 0, f, 0, u)* où *u* est initialisé comme un ensemble vide. Dans cette version optimisée pour les ensembles triés on faitévoluer deux rangs *ei* et *fi* en parralèlle. Si les deux rangs sont suppérieurs ou égaux au cardinal de leur ensemble respectif, c-à-d si tous les éléments ont été traités, la récursion se termine et on renvoie u.

Sinon, si *ei* le rang pour l'ensemble *e* est supérieur ou égal au cardinal de *e* (si tous les éléments de *e* ont été traités) OU si l'élément de *e* à l'indice *ei* est supérieur à l'élément de *f* à l'indice *fi*, alors on peut ajouter ce dernier à l'ensemble *u* sans se poser plus de question, après quoi on rappelle la fonction avec *fi+1*.

Enfin, si les deux conditions précédentes ne sont pas remplies, on ajoute l'élément de *e* à l'indice *ei* dans *u*. Il faut ensuite savoir si l'on incrémente seulement le rang *ei* ou s'il faut incrémenter *ei* et *fi* pour l'appel récursif. On doit incrémenter les deux rangs seulement si, d'une part, *fi* est inférieur au cardinal de *f*, et d'autre part si l'élément de *e* et l'élément de *f* aux indices respectifs *ei* et *fi* sont égaux, afin de ne pas traiter l'élément en double. Sinon on incrémente seulement *ei*.

- *Ensemble interEnsemble(Ensemble e, Ensemble f);*

De même que pour l'union, la fonction d'intersection initialise un nouvel ensemble qui est ensuite rempli via l'appel d'une fonction récursive auxiliaire *__interEnsemble_rec* avant d'être renvoyé.

Cas non-trié : *Ensemble __interEnsemble_rec(Ensemble e, Ensemble f, int rang, Ensemble i);*

Le but est ici de re-construite l'ensemble *f* sans les éléments qui n'appartiennent pas à l'ensemble *e*. L'appel initial est *__interEnsemble_rec(e, f, 0, i)* où *i* est un ensemble vide. Le terme de la récursion est atteint lorsque le rang donné en argument est supérieur ou égal au cardinal de *f*, auquel cas on ronvoie *i*.

Sinon, on extrait l'élémentde *f* à l'indice *rang* et, s'il appartient à *e*, on l'ajoute à l'ensemble *i*. Puis on rappelle la fonction avec *rang+1*. Ainsi, les éléments de *f* qui n'appartiennent pas à *e* sont ignorés.

Cas trié : *Ensemble __interEnsemble_rec(Ensemble e, int ei, Ensemble f, int fi, Ensemble i);*

Dans cette fonction récursive, le terme de la récursion est atteint si l'un ou l'autre des rangs *ei* et *fi* est supérieur ou égal au cardinal de son ensemble respectif, auquel cas on renvoie *i*.

Sinon, si l'élément de *e* et l'élément de *f* aux rangs donnés sont égaux alors on ajoute l'élément de *e* en question à *i*, avant de rappeler récursivement la fonction avec *ei+1* (on garde le même rang pour *f* afin de le traiter à l'appel suivant).

Lorsque les éléments de *e* et de *f* ne sont pas égaux, on rappelle immédiatement la fonction en incrémentant celui des rangs pour lequel l'élément est le plus petit, afin de conserver l'ordre croissant dans le traitement.

- *Bool egEnsemble(Ensemble e, Ensemble f);*

Cette fonction teste l'égalité de deux ensembles en utilisant une fonction auxiliaire récursive dont le fonctionnement diffère selon que les ensembles sont des ensembles triés ou non. Cette opération n'était pas demandée mais elle s'est avérée nécessaire pour tester les préconditions lors de la composition d'application. On considère que comparer des ensembles non-initialisés doit renvoyer *FALSE*, et bien sûr, avant d'appeler la fonction auxiliaire, on commence par vérifier que les ensembles ont la même taille. Dans le cas contraire on revoie *FALSE* immédiatement.

Cas non-trié : *Bool __egEnsemble_rec(Ensemble e, Ensemble f);*

Dans le contexte des ensembles non-triés, la fonction d'égalité auxiliaire est destructive : elle vide, au fur et à mesure des appels récursifs, les deux ensembles *e* et *f*. L'appel initial dans la fonction principale se fait donc avec des copies que l'on veille à détruire ensuite pour des raisons évidentes d'espace mémoire.

A chaque appel récursif, on extrait le premier élément de *e* que l'on supprime de *e* et de *f*, puis on rappelle la fonction avec les deux ensembles. Si au terme de la récursion les deux ensembles sont vides en même temps, alors on retourne *TRUE*, sinon alors un seul des deux ensembles est vide et donc les ensembles sont différents, alors on retourne *FALSE*.

Cas trié : *Bool __egEnsemble_rec(Ensemble e, Ensemble f, int rang);*

Contrairement au cas non-trié, cette fonction ne modifie pas les ensembles donnés en argument. L'appel initial est donc *__egEnsemble_rec(e, f, 0)*. Je rappelle ici que *e* et *f* ont la même taille.

A chaque appel récursif, si l'élément de *e* et l'élément de *f* à l'indice *rang* sont différent, on retourne *FALSE*. Sinon, on rappelle la fonction avex *rang+1*.

Si tous les éléments ont été traités, soit si le rang est supérieur ou égal au cardinal de *e* (ou *f*), alors on retourne *TRUE*.

2. Applications

2.1. Structures

Pour implémanter les applications j'ai défini une structure *s_application* contenant deux champs *depart* et *arrivee* de type *Ensemble* (pointeur sur une structure *s_ensemble*) correspondant aux ensembles respectivement de départ et d'arrivée de l'application, ainsi qu'un tableau dynamique d'entiers *relation*. Ce tableau est initialisé avec la taille de l'ensemble de départ, et chaque entrée correspond à un indice dans l'ensemble de départ. Pour chaque entrée on peut sotcker un *int* (et non un *Entier*) correspondant à l'indice de l'élément image dans l'ensemble d'arrivée (s'il y en a une, sinon, c'est -1).

```
typedef struct s_application
{
        Ensemble depart, arrivee;
        int *relation;
} SApplication, *Application;
```

Pour la fonction *antecedent* détaillée plus loin, j'ai également défini une structure de liste chaînée d'*Entier* très basique pour laquelle j'ai implémantée une fonction de création, de destruction (libération de l'espace mémoire) et d'adjonction en tête de manière assez sommaire, n'ayant pas besoin de plus. La structure *s_liste* contient donc les champs évident *x* de type *Entier* et *next* un pointeur sur *struct s_liste*.

```
typedef struct s_liste
{
        Entier x;
        struct s_liste *next;
} SListe, *Liste;
```

2.2 Opérations sur les applications

- *Application applicationNouv(Ensemble depart, Ensemble arrivee);*

Pour créer et initialiser une nouvelle application, on vérifie que les arguments fournis sont valides, puis on alloue l'espace mémoire nécessaire pour un pointeur de type *Application*. Ensuite, on affecte aux champs *depart* et *arrivee* de la structure pointée une copie de l'ensemble correspondant donné en argument. On alloue également l'espace mémoire nécessaire pour le champs *relation*, à savoir la taille de l'ensemble de départ fois la taille d'un entier *int*. On initialise toutes les valeurs du tableau à -1. Le pointeur ainsi initialisé est ensuite renvoyé.

- *void delApplication(Application A);*

Cette fonction permet de libérer tout l'espace mémoire nécessaire pour une application, à savoir les pointeurs *Ensemble* des champs *depart* et *arrivee*, le tableau *relation* et le pointeur *A* lui même.

- *Application fonction(Application A, Entier x, Entier y);*

Cette fonction permet de mettre à jour le tableau des relations de l'application pointée par *A* en créant un lien antécédent/image entre *x* et *y*. Si *x* n'appartient pas à l'ensemble de départ ou si *y* n'appartient pas à l'ensemble d'arrivée, on arrête la fonction en renvoyant *A*.

Sinon, grâce à la fonction *rech*, on récupère l'indice *ind_x* de *x* dans *A->depart* et l'indice *ind_y* de *y* dans *A->arrivee*. On vérifie alors la valeur sotckée dans *A->relation[ind_x]* : si une relation est déjà définie, on s'interdit de la modifier, on revoie alors *A* accompagné d'un message d'erreur. Sinon, on y écrit la valeur de *ind_y*, puis en renvoie *A* dont la structure a été mise à jour.

- *Entier *image(Application A, Entier x);*

Cette fonction permet de récuperer l'unique image de *x* par l'application *A*, si elle existe. Le type de retour initial était *Entier* mais pour gérer la vérification des préconditions j'ai opté pour un pointeur afin de pouvoir renvoyer la valeur *NULL* en cas d'erreur. Les cas d'erreur sont les suivants : *A* vaut *NULL*, *x* n'appartient pas à l'ensemble de départ de l'application, il n'y a pas de relation pour cet élément. Si les 2 premiers cas d'erreur ne sont pas rencontrés, alors, avec l'indice de *x* dans l'ensemble de départ de l'application (obtenu avec *rech*) on récupère l'indice de l'image dans le tableau *relation*. Si on récupère -1, alors *x* n'a pas d'image, on retourne *NULL*. Sinon, on renvoie le un pointeur sur l'élément à cet indice dans l'ensemble d'arrivée.

- *Liste antecedent(Application A, Entier y);*

Pour récupérer les antécédents de *y*, on commence par vérifier la validitié de *A* et l'appartenance de *y* a l'ensemble d'arrivée. Ensuite, on récupère l'indice de *y* et on appelle la fonction auxiliaire récursive *Liste __antecedent_rec(Application A, int ind_y, int rang, Liste l)*. On retourne la liste renvoyée par cette fonction.

- *Liste __antecedent_rec(Application A, int ind_y, int rang, Liste l);*

L'appel initial est *__antecedent_rec(A, ind_y, 0, l)*, où i*nd_y* est l'indice de *y* dans l'ensemble d'arrivée de l'application, et *l* est une liste vide. A chaque appel récursif, si *rang* est supérieur ou égal au cardinal de l'ensemble de départ (donc du tableau des relations), alors on retourne la liste *l*. Sinon, si la valeur stockée dans *relation* à l'indice *rang* correspond à *ind_y*, alors on récupère l'élément à l'indice *rang* dans l'ensemble de départ et on l'ajoute en tête de la liste *l*. Ensuite, on rappelle la fonction avec *rang+1*.

- *Application composition(Application F, Application G);*

Pour faire la composition *FoG*, *F* et *G* étant des applications correctement initialisées, il faut commencer par vérifier la pré-condition que l'ensemble d'arrivée de *G* est le même que l'ensemble de départ de *F*, d'où la fonction *egEnsemble* évoquée précédemment. Une fois cette vérification effectuée, cette fonction initialise une nouvelle application ayant pour ensemble de départ celui de *G* et pour ensemble d'arrivée celui de *F*. Pour la construction des relations image/antécédent de la nouvelle application, j'utilise ici une fonction auxiliaire récursive détaillée ci-après, puis le pointeur sur l'application est renvoyé.

- *Application __composition_rec(Application F, Application G, int rang, Application FoG);*

Cette fonction effectue un parcours récursif de l'ensemble de départ de *FoG*, l'appel initial étant *__composition_rec(F, G, 0, FoG)* où *FoG* est une application initialisée avec l'ensemble de départ de *G* et l'ensemble d'arrivée de *F*.

A chaque appel, si le rang est inférieur au cardinal de l'ensemble de départ de *FoG*, on récupère l'image par l'application *G* de l'élément d'indice *rang* dans l'ensemble de départ de *FoG* via un pointeur renvoyé par la fonction *image*. Si cette image existe, on récupère son image par l'application *F* de la même manière. Si cette seconde image est définie alors on met à jour le tableau des relation de *FoG* à l'indice *rang* avec l'indice de cette image dans l'ensemble d'arrivée de *FoG* (que l'on récupère avec la fonction *rech*). Sinon on ingore simplement cet indice dans la table des relations, qui reste à la valeur -1 signifiant qu'il n'y a pas d'image pour cet élément de l'ensemble de départ.

Lorsque le rang est supérieur ou égal au cardinal de *FoG* alors on peut renvoyer l'application à jour *FoG*.

…

Une série de tests est fournie ainsi qu'un Makefile pour tout compiler. Les résultats des tests sont également répertoriés dans un fichier *Resultats.txt*.