L2 Algorithmique des arbres - Session 1

Durée 2 heures.

Notes de cours, de TD et de TP autorisées.

- Les différentes parties (5 exercices) sont indépendantes et peuvent être traitées dans n'importe quel ordre.
- Chaque fonction sera écrite en `C`. On veillera à minimiser la complexité des fonctions.
- On pourra écrire des fonctions annexes, si l'utilisation de paramètres supplémentaires est nécessaire.
- Le Q.C.M. (12 questions) sera noté sur 4 points.

▸ **Exercice 1. Arbres binaires.** (5 points)
On utilise la structure :

```
typedef struct noeud{
    int val;
    struct noeud *fg, *fd;
} Noeud, *Arbre;
```

1. Écrire une fonction `int hauteur(Arbre a)` qui renvoie la hauteur de l'arbre `a`.
2. Écrire une fonction `int copie_arbre(Arbre * dest, Arbre src)` qui réalise une copie de l'arbre binaire `src` dans l'arbre `*dest`. La fonction renverra 0 en cas d'échec, et 1 en cas de succès.

   On pourra supposer écrit les fonctions `Noeud * alloue_noeud(int val)` *qui alloue en mémoire un nœud contenant la valeur* `val` *et* `void libere(Arbre a)` *qui libère l'espace mémoire alloué lors de la construction ou la manipulation de l'arbre* `a`.
3. On dit qu'un arbre binaire d'entiers est un arbre somme lorsqu'en chaque nœud interne de l'arbre, on trouve comme etiquette la somme des étiquettes du sous-arbre gauche et du sous-arbre droite de ce nœud.

   Écrire une fonction `int est_somme(Arbre a)` qui renvoie 1 si l'arbre `a` est un arbre somme.
4. Écrire une fonction `void etiquette_en_abr(Arbre a, int * min)` qui réétiquette les nœuds de l'arbre binaire a pour qu'il devienne un arbre binaire de recherche.

   A l'issue de l'appel à la fonction `etiquette_en_abr`, la forme de l'arbre n'aura pas changée. Les étiquettes de l'arbre passé en paramètre seront des entiers consécutifs, supérieurs ou égaux à `*min`.

   On supposera que `*min` *définit correctement un entier.*

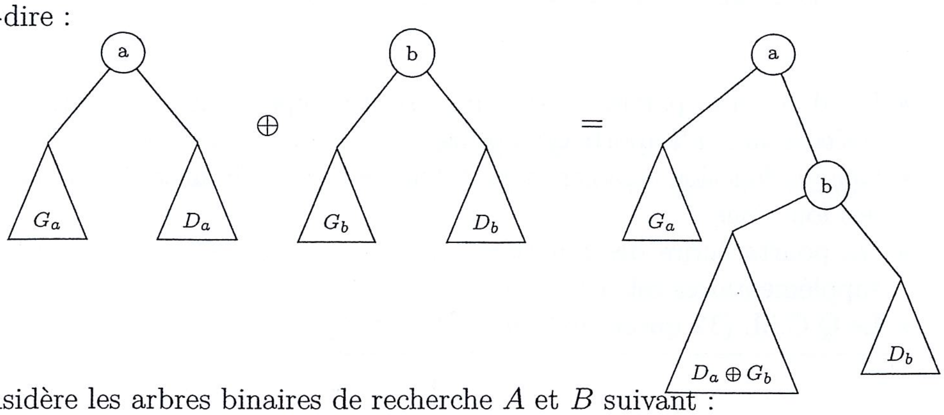
▸ **Exercice 2. Fusion d'arbres binaires de recherche** (4 points)
On rappelle que Λ désigne dans tout l'exercice l'arbre vide.
On définit l'opération $\oplus$ agissant sur les arbres binaires comme suit :

- Pour tout arbre binaire A, $A \oplus \Lambda = \Lambda \oplus A = A$;

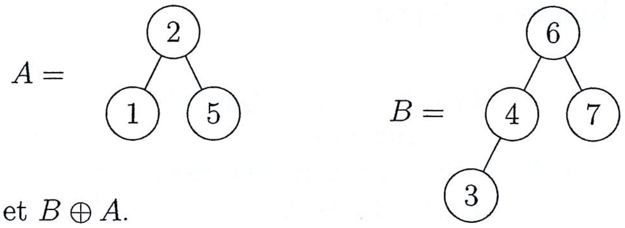
- Pour tous arbres binaires non vide $A = (a, G_a, D_a)$ et $B = (b, G_b, D_b)$:

$$A \oplus B = \big(a, G_a, (b, D_a \oplus G_b, D_b)\big)$$

c'est-à-dire :



1. On considère les arbres binaires de recherche A et B suivant :



   (a) Dessiner $A \oplus B$ et $B \oplus A$.

   (b) Est-ce que l'on a construit des arbres binaires de recherche ? *(La réponse **devra** être justifiée.)*

2. Dans cette question, on suppose désormais que A et B sont deux arbres binaires de recherche sans doublons tels que les étiquettes de A sont strictement plus petites que celle de B.

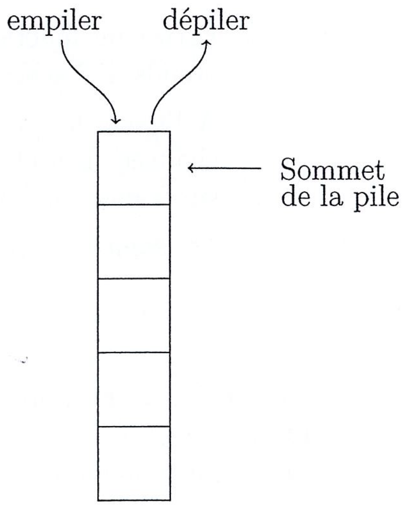
   (a) Démontrer que $A \oplus B$ est un arbre binaire sans doublons.

   (b) En déduire que $A \oplus B$ est un arbre binaire de recherche.

3. Écrire la fonction `void fusion(Arbre * A, Arbre B)` qui modifie l'arbre `*A` en `*A` $\oplus$ `B`.

▸ Exercice 3. File de priorités (5 points)

On rappelle dans cet exercice qu'une pile est une structure de données basée sur le principe *Last In First Out*, comme une pile d'assiettes. Le type `Pile` étant défini, on peut entre autre l'utiliser pour réaliser les opérations suivantes :



— `int est_vide(Pile p)`, qui teste si la pile `p` est vide ;

— `int consulter(Pile p)`, qui renvoie la valeur au sommet de la pile `p` ;

— `int depiler(Pile p, int * sortant)`, qui retire de la pile `p` son sommet et le stocke dans `*sortant` lorsque `p` est non vide, la fonction renvoie 1 si l'opération a pu être réalisée, 0 sinon (*i.e.* lorsque la pile `p` était vide) ;

— `void liberer(Pile * p)`, qui libère l'espace alloué pour la pile `*p`.

Dans cet exercice, on se donne un tableau T de m piles d'entiers tel que :

- aucune pile de T n'est initialement vide ;
- il y a en tout N entiers répartis dans les m piles ;
- les sommets des piles vont décroissants :

$$\text{Sommet(T[m - 1])} \leq \text{Sommet(T[m - 2])} \leq \cdots \leq \text{Sommet(T[0])};$$

- dans chaque pile, les entiers sont ordonnés du plus grand (au sommet) au plus petit.

On souhaite réaliser l'interclassement des éléments des m piles de manière à obtenir un nouveau tableau contenant les N éléments rangés dans l'ordre croissant.

1. Rappeler ce qu'est une file de priorité, les opérations que l'on souhaite réaliser dessus Expliquer trois implémentations possibles, en précisant leurs avantages / défauts.
2. Pourquoi le tableau T peut-il être vu comme un tas max dont les éléments sont des piles et les priorités associées sont les entiers au sommet des piles.
3. Ainsi, le tableau T s'interprète initialement comme une file de priorité.
   (a) Comment fonctionne l'opération de suppression dans cette file de priorité T ?

   A l'issue de l'opération, la structure de tas max devra être rétablie dans tous les cas.

   Un ou plusieurs exemples pourront accompagner le fonctionnement de la suppression.

   (b) Quelle est la complexité de l'opération ?
4. On considère la structure suivante :

```
typedef struct {
    Pile * T;
    int N;
    int m;
} Tas;
```

   Dans toute cette question, seule les fonctions de manipulation des piles (`est_vide`, `consulter`, `depiler` et `liberer`) pourront agir sur des objets de type abstrait `Pile`.

   (a) Écrire la fonction `int Fils(Tas * tas, int i)` qui renvois l'indice de l'enfant de la pile d'indice `i` dans le tableau `T` associé au tas `*tas` ayant le plus haut sommet.

   La fonction renverra -1 en cas d'absence d'enfants.

   (b) Dans cette question, on suppose que `*tas` contient un tableau `T` non vide et que tous ses éléments vérifient bien la condition de tas binaire, sauf éventuellement la racine.

   Écrire la fonction `void descente(Tas * tas)` qui fait descendre la racine du tas à la place où elle devrait se trouver dans le tas `*tas` pour rétablir parfaitement la structure de tas.

   (On pourra supposer que la fonction `echange(Pile * un, Pile * deux)` *inversant le contenu des variables* `*un` *et* `*deux` *est déjà disponible.)*

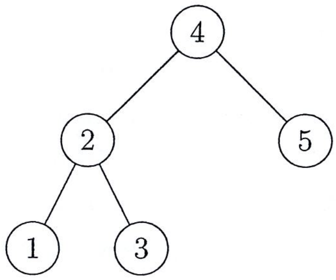
   (c) Écrire la fonction `int suppr(Tas *tas, int * sortant)` qui supprime la valeur à la racine du tas `*tas`. Cette valeur sera stockée dans la variable `*sortant` en cas de succès, la fonction renverra alors 1. Dans le cas où la suppression n'a pas été possible, elle renverra 0.

   Rappel : *On pensera à libérer tout espace mémoire devenu inutile.*

5. Donner un algorithme en $\mathcal{O}(N \ln N)$ comparaisons pour résoudre le problème posé : réaliser l'interclassement des éléments des m piles de manières à obtenir un nouveau tableau contenant les N éléments rangés dans l'ordre croissant.

   La réponse sera justifiée !

▸ **Exercice 4. Insertions et suppressions dans un AVL** *(4,75 points)*
Dans tout cet exercice, nous souhaitons, après insertion d'une série de nombres, aboutir à l'arbre A suivant :



Dans tout cet exercice, lors d'une suppression dans un arbre binaire de recherche, on remontera la valeur minimale de sous-arbre de droite lorsqu'il y aura le choix.

Remarque importante : Chaque réponse de cet exercice devra être justifiée !

1. Donner tous les ordres possibles d'insertion des entiers 1, 2, 3, 4 et 5 dans un ABR initialement vide permettant d'obtenir A.
2. Parmi ces ordres d'insertions, lesquels produisent le même arbre, en partant d'un AVL initialement vide, tout en maintenant une structure d'AVL à chaque insertion ?
3. (a) Donner toutes les formes possible d'AVL à 1, 2 et 3 nœuds

   La réponse devra être justifiée précisément.

   (b) i. Donner toutes les formes possibles d'AVL à 4 nœuds.

   ii. Parmi celles-ci, décrire précisément lesquels permettent d'aboutir à l'arbre A en effectuant un seul rééquilibrage lors de l'insertion d'un cinquième nœud ; préciser la rotation alors effectuée s'il y en a une.

   (c) i. Donner tous les ordres d'insertion des entiers de 1 à 5 permettant d'aboutir à l'arbre A en effectuant un seul rééquilibrage par rotation gauche.

   ii. Donner tous les ordres d'insertion des entiers de 1 à 5 permettant d'aboutir à l'arbre A en effectuant exactement deux rééquilibrages, d'abord une rotation droite, puis une rotation gauche-droite.

   iii. *(Question bonus, + 1 point)*

   Pouvez-vous anticiper combien d'ordre différents d'insertion des entiers 1 à 5 permettent d'obtenir l'arbre A, peu importe le nombre de rotations à effectuer et leur ordre ?