Exercices de révisions MP

Exercice 1 (Arbres binaires de recherche) :

Dans cet exercice, on autorise les doublons dans un arbre binaire de recherche et pour le cas d'égalité on choisira le sous-arbre gauche. On ne cherchera pas à équilibrer les arbres.

1. Rappeler la définition d'un arbre binaire de recherche.
2. Insérer successivement et une à une dans un arbre binaire de recherche initialement vide toutes les lettres du mot *bacddabdbae* en utilisant l'ordre alphabétique sur les lettres. Quelle est la hauteur de l'arbre ainsi obtenu ?
3. Montrer que le parcours en profondeur infixe d'un arbre binaire de recherche de lettres est un mot dont les lettres sont rangées dans l'ordre croissant. On pourra procéder par induction structurelle.
4. Proposer un algorithme qui permet de compter le nombre d'occurrences d'une lettre dans un arbre binaire de recherche de lettres. Quelle est sa complexité ?
5. On souhaite supprimer une occurrence d'une lettre donnée dans un arbre binaire de recherche de lettres. Expliquer le principe de l'algorithme permettant de résoudre ce problème et le mettre en oeuvre sur l'arbre obtenu à la question 2 en supprimant successivement une occurrence des lettres e, b, b, c et d. Quelle est sa complexité ?

Exercice 2 (Arbres AVL) :

Question de cours :

- Construire l'arbre binaire de recherche obtenu en y insérant dans cet ordre les entiers 5, -2, 0, 4, 1, 8, 6.
- Dans un arbre binaire de recherche à n noeuds, quelle est la complexité de la recherche et de l'ajout d'un élément en pire cas ? Justifier rapidement

Pour améliorer la complexité des opérations précédentes, on aimerait limiter la hauteur de l'arbre. Une manière de faire est de construire des arbres équilibrés, comme les arbres rouge-noir. On propose ici une famille particulière d'arbre qu'on appelle les arbres AVL.

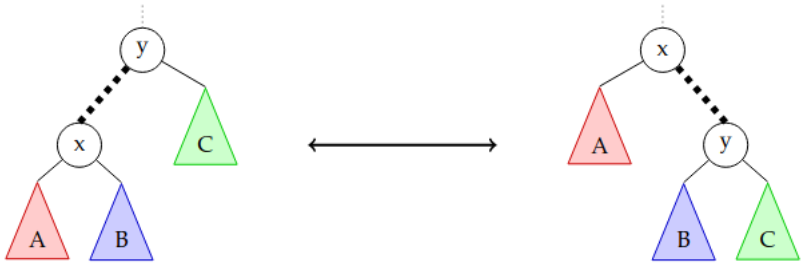
Soit φ une fonction qui à tout arbre binaire de recherche associe l'ensemble des étiquettes apparaissant sur ses noeuds. On définit les arbres **AVL** de manière inductive :

- l'arbre vide $\perp$ est un arbre AVL.
- si g et d sont des arbres et x une étiquette tels que :
  - g et d sont des arbre AVL
  - $|h(g) - h(d)| \leq 1$ (l'arbre est équilibré)
  - $\max(\varphi(g)) < x \leq \min(\varphi(d))$ (on prendra la convention que $\min(\emptyset) = +\infty$ et $\max(\emptyset) = -\infty$)

  Alors $N(g, x, d)$ est un arbre AVL.

1. Donner un exemple d'arbre AVL de hauteur 2, et d'un arbre de hauteur 3 qui n'est pas un arbre AVL.
2. Soit N_h le nombre minimum de noeuds dans un arbre AVL de hauteur h. Montrer que $(N_h)_{h\geq -1}$ vérifie une relation de récurrence simple.
3. On pose $F_{h+1} = N_h + 1$, pour tout $h \geq -1$. En s'appuyant sur la suite $(F_h)_{h\in\mathbb{N}}$, justifier que la hauteur h d'un arbre AVL à n noeuds vérifie $h = O(\log(n))$.

   On rappelle les opérations de *rotations* sur des arbres, sans les redéfinir formellement. Le principe est rappelé sur le schéma ci-dessous ; on parle de *rotation droite* lorsqu'on transforme l'arbre de gauche en celui de droite, et de *rotation gauche* dans l'autre sens.



4. Dans l'exemple de la question de cours, l'arbre initial est vide, c'est donc un AVL. Relever à quel ajout l'arbre cesse d'être un AVL, et proposer une méthode pour corriger ce problème. Appliquer la même méthode pour ajouter les éléments qui suivent.
5. Soit t un arbre AVL dans lequel on insère un élément, comme dans un arbre binaire de recherche classique ; justifier qu'après l'insertion, on a besoin d'au plus deux rotations pour obtenir un nouvel arbre AVL contenant les mêmes éléments.
6. Quelle est la complexité de l'algorithme précédent ? On l'exprimera en fonction de données pertinentes.
7. On fournit en OCaml le type `abr`, représentant des arbres binaires de recherches sur des entiers, défini comme suit : `type abr = V | N of abr*int*abr`.
   Implémenter une fonction `inserer_avl : abr -> int -> abr` qui ajoute un entier dans un arbre supposé être un AVL, et utiliser le procédé précédent pour renvoyer l'AVL obtenu en sortie. On prendra soin de définir les fonctions auxiliaires nécessaires.

Exercice 3 (Minima locaux dans des arbres) :

Dans cet exercice, on considère des arbres binaires étiquetés par des entiers relatifs deux à deux distincts. Un nœud est un minimum local d'un arbre si son étiquette est plus petite que celle de son éventuel père et celles de ses éventuels fils. Considérons par exemple l'étiquetage (b) de l'arbre binaire non étiqueté (a).

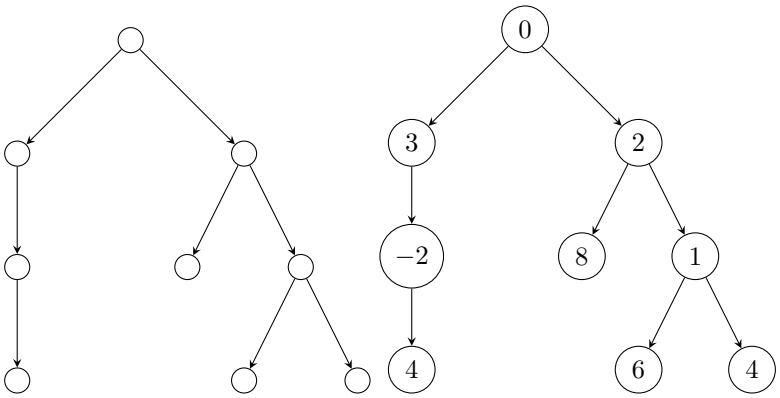


Figure 1: Arbre non-étiqueté (a) à gauche, et arbre étiqueté (b) à droite

1. Déterminer le ou les minima locaux de l'arbre (b).
2. Donner une définition inductive permettant de définir les arbres binaires ainsi que la définition de la hauteur d'un arbre. Quelle est la hauteur de l'arbre (b) ?
3. Montrer que tout arbre non vide possède un minimum local.
4. Proposer un algorithme permettant de trouver un minimum local d'un arbre non vide et déterminer sa complexité.

   On considère un arbre binaire non étiqueté que l'on souhaite étiqueter par des entiers relatifs distincts deux à deux de manière à maximiser le nombre de minima locaux de cet arbre.

5. Proposer sans justifier un étiquetage de l'arbre (a) qui maximise le nombre de minima locaux.
6. Proposer un algorithme qui, étant donné un arbre binaire non étiqueté en entrée, permet de calculer le nombre maximal de minima locaux qu'il est possible d'obtenir pour cet arbre. Déterminer la complexité de votre algorithme.
7. Montrer que, pour un arbre de taille $n \in \mathbb{N}$, le nombre maximal de minima locaux est majoré par $\left\lfloor \frac{2n+1}{3} \right\rfloor$.
   On pourra remarquer que les nœuds non minimaux couvrent l'ensemble des arêtes de l'arbre.

Exercice 4 (Flux de données) :

Question de cours :

- Construire le tas-min obtenu en y insérant dans cet ordre les entiers 5, -2, 0, 4, 1, 8, 6.
- Dans un tas binaire à n noeuds, quelle est la complexité de l'ajout et de la suppression d'un élément en pire cas ? Justifier rapidement.

On s'intéresse dans la suite à un flux de données, c'est-à-dire à un séquence infinie de données reçu par un code qui s'exécute en continu ; un tel code serait de la forme suivante, dans laquelle on ne spécifie pas la méthode par laquelle les données sont reçues.

Algorithme 1 : A

```
(étape 1 : à compléter)
while True do
    Recevoir x
    (étape 2 : à compléter)
```

On suppose que les données qu'on reçoit appartiennent toutes à un ensemble totalement ordonné (par exemple $\mathbb{Z}$). Dans la suite, on voudrait pouvoir connaître, à chaque instant, la valeur du k-ème plus grand élément parmi toutes les données qui ont déjà été reçues. On souhaite donc compléter ce code, en déclarant des structures de données adaptées avant la boucle, et en complétant le code de la boucle, de sorte que chaque fois que l'on reçoit une nouvelle donnée x, on mette à jour le k-ème plus grand élément connu si nécessaire.

1. Dans le cas où $k = 1$, proposer une solution simple pour compléter le pseudo-code de l'algorithme A ci-dessus.
2. Comment adapterait-on cette approche pour $k = 2$? Et pour un k quelconque (on pourra donner k en argument à l'algorithme A) ? Dans les deux cas, compléter le pseudo-code de l'algorithme A.
3. Quelle est la complexité temporelle d'une réalisation de l'étape 2 ? Quelle est la complexité spatiale de l'algorithme A ?
4. Proposer une structure de données adaptée qui améliore la complexité de cette approche naïve, et compléter en conséquence le pseudo-code de A
5. Justifier qu'on obtient bien une meilleur complexité temporelle pour une réalisation de l'étape 2.
6. Quelle est la complexité spatiale de cette nouvelle approche ?

   On souhaite maintenant conserver, à chaque instant, plusieurs éléments consécutifs parmi les données qui ont déjà été reçues. Dans la suite, on se donne deux entiers k et q, et on supposera qu'on cherche à conserver, le k-ème plus grand, le $(k+1)$-ème plus grand, etc. jusqu'au $(k+q-1)$-ème plus grand.
7. Proposer un pseudo-code pour l'algorithme A permettant de résoudre ce problème (on suppose qu'on lui donne k et q en argument), et discuter de la complexité temporelle de l'étape 2, et de la complexité spatiale de l'algorithme.

On suppose dans les deux questions suivantes que le flux est fini ; dans l'algorithme A, on remplace donc le `true` dans la condition du `while` par un test, dont on ne donne pas le détail, qui vaut `true` pendant un certain nombre d'itérations, puis `false` au bout d'un nombre arbitraire d'itérations.

8. Comment cette méthode se compare-t-elle à une méthode naïve dans le cas où les données reçuessont toutes connues à l'avance au lieu d'être reçues par un flux (par exemple, disponibles dans un tableau de taille n) ?
9. Quelle amélioration de la méthode précédente peut-on apporter si on connaît par avance la taille du flux (donc le nombre d'itérations qui seront exécutées dans l'algorithme A) pour diminuer la complexité spatiale ?

Exercice 5 (Multi-fusions) :

Question de cours :

- Construire le tas-min obtenu en y insérant dans cet ordre les entiers 5, -2, 0, 4, 1, 8, 6.
- Dans un tas binaire à n noeuds, quelle est la complexité de l'ajout et de la suppression d'un élément en pire cas ? Justifier rapidement.

On souhaite savoir comment fusionner efficacement k listes triées (par exemple dans l'ordre croissant).

1. Proposer une méthode pour le cas $k = 2$. On pourra écrire du pseudo-code, ou décrire le principe de la méthode.
2. Justifier la correction et la complexité de cette méthode.
3. Proposer une implémentation en OCaml de cette méthode.
4. Comment peut-on l'adapter pour $k > 2$? Quelle est la complexité qui en résulte ?
5. Quelle structure de données pourrait être utilisée pour résoudre le problème consistant à trouver le plus petit élément en tête de ces k listes ?
6. Proposer un algorithme A en pseudo-code utilisant l'idée de la question précédente. Cet algorithme recevra en argument un tableau t de taille k, tel que chaque case $t[k]$ contient une des listes triées en entrée.
7. Quelle est la complexité spatiale et temporelle de cette méthode ?

On suppose que les listes de données à traiter ne sont plus données sous forme de listes, mais sous forme de flux. Un flux f est un objet dans lequel on peut récupérer des données, à travers une fonction Recevoir, qui renvoie :

- soit la prochaine donnée x envoyée par le flux.
- soit la constant EOF si le flux est terminé et n'envoie plus de données.

Dans la suite, on dispose de k flux différent, qu'on suppose tous finis : ils envoient tous un nombre fini de données, mais ce nombre n'est pas connu à l'avance, et peut être différent pour chaque flux.

8. On suppose que chaque flux nous envoie des entiers dans l'ordre croissant. Peut-on adapter la méthode précédente ?
9. Proposez une implémentation OCaml, on pourra supposer disposer des flux en entrée sous la forme d'un tableau d'objets de type `stream_type` et d'une fonction `stream_pop: stream_type -> int` qui extrait le premier élément d'un flux. Votre programme affichera la liste des entiers dans l'ordre croissant. Vous pourrez supposer avoir accès à des fonctions `heap_create`, `heap_insert`, `heap_extract` et `heap_size` pour la gestion du tas, ainsi qu'à la constante `eof` modélisant `EOF` définie au dessus.

Exercice 6 :

Les deux questions sont indépendantes.

1. Prouvez que $\vdash A \to (A \to B) \to B$
2. Prouvez que $\vdash ((A \vee B) \to C) \to (A \to C) \wedge (B \to C)$

Exercice 7 :

1. Montrer que le séquent $\vdash \neg A \to (A \to \bot)$ est dérivable/constructible.
2. Montrer que le séquent $\vdash (A \to \bot) \to \neg A$ est dérivable/construcible.
3. Montrer que le séquent $\vdash (\neg A \to (A \to \bot)) \wedge ((A \to \bot) \to \neg A)$ est dérivable.
4. On considère la formule $P = ((A \to B) \to A) \to A$ appelée *loi de Pierce*. Montrer que $\models P$, c'est-à-dire que P est une tautologie.
5. Montrer que le séquent $\vdash P$ est constructible/dérivable.