
Cahier de TD de SDD
: structures arborescentes

ARBRES (Basiques)

Exercice 3.1 – Rechercher un élément dans un ABR[[1]](#footnote-1) 🟅

Concevoir un algorithme itératif pour rechercher un élément dans un ABR.

Exercice 3.2 – Ajouter un élément dans un ABR 🟅

Concevoir un algorithme itératif pour ajouter un élément dans un ABR.

Exercice 3.3 – Analyser un AB[[2]](#footnote-2), connaître le vocabulaire 🟅🟅

Concevoir des algorithmes pour évaluer (compter ou mesurer) sur un AB :

1. le nombre d’éléments,
2. le nombre de feuilles,
3. le nombre de points doubles,
4. le nombre de nœuds monoparentaux.

Problème 3.4 – Un peu de réflexion algorithmique 🟅🟅(🟅)

1. Concevoir un algorithme qui retourne le nombre de nœuds d’un arbre situés à une profondeur donnée[[3]](#footnote-3).
2. En vous appuyant sur l’algorithme précédent, concevoir un algorithme itératif qui retourne le niveau de profondeur d’un arbre qui contient le plus grand nombre de nœuds.
Evaluer la complexité algorithmique de cet algorithme.
3. 🟅🟅🟅 Concevoir un algorithme qui fait la même chose que le précédent de manière plus performante[[4]](#footnote-4).

Exercices supplémentaires pour les plus courageux

Exercice 3.5 – Manipulations structurales simples sur un AB 🟅🟅

1. Concevoir un algorithme qui transpose[[5]](#footnote-5) toutes les nœuds feuilles d'un arbre avec leur frère.
2. Concevoir un algorithme pour transformer un AB en son symétrique par rapport à l'axe vertical (pour chaque niveau de profondeur, l’ordre des nœuds est inversé entre la droite et la gauche).

Exercice 3.­6 – 5.3 adapté pour retourner des listes d’éléments 🟅🟅🟅

Adapter les exercices 5.3.1 à 5.3.3 pour concevoir des algorithmes pour retourner sous forme de liste simplement chaînée selon un parcours en profondeur main gauche ou un parcours en largeur de gauche à droite :

1. la liste des éléments,
2. la liste de feuilles,
3. la liste des points doubles,
4. la liste des nœuds monoparentaux.

d’un AB.

Exercice 3.7 – Extractions de listes moins triviales 🟅🟅🟅🟅

Concevoir un algorithme pour extraire sous forme de LSC, à partir d’un AB :

1. Les branches extérieures gauche et droite,
2. Les feuilles,
3. Une coupe transversale de profondeur donnée (de gauche à droite),
4. Une branche située entre la racine et un nœud quelconque.

**Note** – vous pouvez concevoir les LSC à extraire comme des LSC contenant les adresses des nœuds, ou bien celles des éléments, ou bien encore, une copie des éléments de l’arbre.

Problème 3.8 – Adressage logique dans un AB 🟅🟅🟅🟅

L’adresse logique relative d’un sous-arbre d’un arbre, exprimée en binaire, est constituée d’autant de bits que de nœuds sur la branche qui relie la racine de l’arbre au sous-arbre. Le premier bit est nécessairement 1 (adresse de la racine), les suivants sont 0 ou 1 respectivement pour indiquer un prochain pas vers la gauche ou vers la droite : par exemple 101 est l’adresse du fils droit du fils gauche de la racine.

1. Concevoir un algorithme qui retourne l’adresse logique du sous-arbre d’un arbre.
2. Quel est l’intérêt de disposer d’une telle adresse ?
Indice : pourquoi un ABR est-il plus intéressant qu’un AB standard.
3. Proposer des fonctionnalités performantes qui exploitent ce système d’adressage.

ARBRES (Moyens)

Exercice 3.9 – ABR : opération utilitaire élémentaire pour préparer le 6.2 🟅🟅

1. Concevoir un algorithme itératif pour récupérer l’adresse[[6]](#footnote-6) du nœud qui contient le plus petit élément d’un ABR. Même question avec le plus grand élément.
2. Concevoir un algorithme itératif pour échanger un élément d’un ABR avec le plus grand élément de son sous-arbre gauche. Même question avec le plus petit élément de son sous-arbre droit.
3. Que peut-on dire de l’arbre après une telle opération ?

Problème 3.10 – Supprimer un élément d’un ABR 🟅🟅🟅

Concevoir un algorithme pour supprimer un élément dans un ABR.

Envisager deux possibilités dont l’une déséquilibre l’arbre tandis que l’autre maintient l’équilibre.

La méthode qui permet de maintenir l’équilibre peut suivre deux options : laquelle faut-il préférer, pourquoi ?

**Indice** – traiter d’abord le cas de la feuille, puis celui du nœud monoparental. Réfléchir ensuite à un moyen, en vous appuyant sur l’exercice précédent, de ramener le cas du point double au cas du nœud monoparental.

Exercice 3.11 – Parcours d’un AB 🟅🟅

Concevoir un algorithme itératif puis récursif de parcours pré-ordre d’un arbre à l’aide d’une pile auxiliaire.

Que doit-on modifier dans ces deux algorithmes, pour un parcours infixe et post-ordre ?

Pour libérer la mémoire allouée pour un arbre, quel ordre de parcours faut-il choisir ? Pourquoi ?

Exercice 3.12 – Dupliquer et supprimer un AB 🟅🟅

Concevoir un algorithme pour :

1. Dupliquer un AB.
2. Supprimer un AB.

Exercice 3.13 – Création d’AB qui représente un tas 🟅🟅

Concevoir un algorithme pour créer un arbre binaire qui correspond au mappage d’initialisation du tri par tas (heapsort) à partir d'une liste simplement chaînée d'entiers.

Exercices supplémentaires pour les plus courageux

Problème 3.14 – Pour calmer le brouhaha du groupe B 🟅🟅🟅🟅

Concevoir un algorithme qui réalise le décalage circulaire d’un nombre de pas donné (dans le sens des aiguilles d’une montre) de l’ensemble des éléments contenus dans les nœuds qui forment le contour d’un arbre (branches la plus à gauche et la plus à droite, feuilles).



Exemple : décalage circulaire d’un pas de 2

Problème 3.15 – Générer l’arbre de Syracuse d’ordre n 🟅🟅🟅🟅

Une suite de Syracuse de terme initial un entier *N* est définie par :



Il semblerait que quel que soit *N*, la suite *sN* converge vers 1, i.e. :



Il vous est demandé de réaliser une fonction récursive qui construit un arbre de Syracuse de racine un entier donné *e*, de profondeur donnée *q*, dont les branches sont toutes les suites de Syracuse qui convergent vers *e* en *q* étapes.

Pour cela, il suffit de procéder à une inversion de la suite.

Partons de *e*, le terme de convergence visé, et racine de l’arbre : *e* peut être le successeur direct de 2*e*, et éventuellement de (*e* – 1)/3 si cet entier existe et s’il est impair. Disons que *e* a nécessairement un fils droit 2*e*, et éventuellement un fils gauche (*e* – 1)/3. En appliquant récursivement ce procédé, on peut donc construire un arbre binaire de profondeur quelconque dont chaque branche est l’une des suites de Syracuse qui converge vers *e*.

Écrire une fonction qui retourne le prédécesseur entier impair d’un entier donné s’il existe, et 0 sinon.

Écrire une fonction récursive qui construit l’arbre de Syracuse à partir d’une feuille donnée contenant la valeur *e* et développé jusqu’à la profondeur *p*. Si un fils gauche (*e* – 1)/3 vaut 0 ou 1 à une étape quelconque de la récursion (i.e. *e* vaut 1 ou 4), ce fils gauche n’est pas créé.

ARBRES (Avancés)

Exercice 3.16 – Mesurer la hauteur d’un AB 🟅🟅

Concevoir un algorithme pour mesure la hauteur d’un AB.

Exercice 3.17 – Vérifier qu’un AB est le sous AB d’un autre 🟅🟅🟅

Concevoir un algorithme pour vérifier qu’un AB est le sous-arbre d’un autre.

Problème 3.18 – Vérifier qu’un AB est un ABR 🟅🟅🟅(🟅)

Concevoir un algorithme pour reconnaître si un AB donné est un ABR ou non.

1. Commencer par concevoir les algorithmes pour retourner plus petit et plus grand éléments d’un AB.
2. Utiliser ces algorithmes pour effectuer la vérification de la condition d’ordre locale entre la racine et ses deux sous-arbres (*cf*. définition de l’ABR).
3. Vérifier que l’arbre et chacun de ses sous-arbres vérifie cette condition *cf*. définition de l’ABR).
4. 🟅🟅🟅🟅 Trouver un moyen astucieux de faire la même chose avec une meilleure performance.
Comparer les performances respectives.

Exercices supplémentaires pour les plus courageux

Problème 3.19 – Transformer un ensemble de LSC en forêt 🟅🟅🟅🟅🟅🟅🟅

1. Etant donné un ensemble de LSC deux à deux confluentes[[7]](#footnote-7), concevoir un algorithme qui transforme cet ensemble de listes en un AB dont les feuilles sont les ex têtes de liste, les parents, les ex successeurs, la racine, l’ex queue de liste nécessairement unique et commune à toutes les listes.

Pour précision :

* On travaille sans copie, c'est-à-dire que l’on réutilise les nœuds de la LSC conçue comme une LDC dont on n’a pas réalisé le chaînage arrière (cf. TP 1), on les détache de la LSC, et on considère dès lors ses deux pointeurs succ et prev respectivement comme sag et sad.
* Si un nœud n’a qu’un sous-arbre (il n’est pas un point de confluence), ce sous-arbre est placé de préférence à gauche.
* Les listes sont à traiter dans l’ordre où elles sont données, c'est-à-dire l’ordre des adresses des têtes de listes, données sous la forme d’un tableau.
1. Par généralisation, concevoir un algorithme qui transforme n’importe quel ensemble de LSC, confluentes ou non, à terminaison circulaire ou non en une forêt.

Note – après conception et réalisation de votre algorithme, vous pouvez en tester le bon fonctionnement à l’aide, par exemple de l’arbre de Syracuse (Problème supplémentaire

Problèmes 3.20 – Dénombrement d’ABR et de classes d’équivalence d’AB (mod. ABR) 🟅🟅🟅🟅🟅🟅🟅🟅

Soit un ensemble de *n* éléments strictement ordonnés.

1. 🟅🟅🟅🟅🟅 Combien existe-t-il ABR 2 à 2 distincts qui contiennent ces *n*, et seulement ces *n* éléments ?
**Conseil** – Etayez votre réponse d’une preuve rigoureuse (la récurrence peut être utile).
**Coup de pouce** – Rappel : 
2. 🟅🟅🟅🟅🟅🟅🟅🟅 Pour chacun de ces ABR, combien d’AB 2 à 2 distincts sont-ils équivalents par la relation « peuvent être transformés en l’ABR par une succession de transpositions de nœuds frères ».
3. Concevoir un algorithme qui transforme un AB en ABR par une succession de transpositions de nœuds frères.
4. Concevoir un algorithme qui produit tous les ABR à partir d’un ensemble de *n* éléments, un autre qui produit directement le mieux équilibré d’entre eux.

ARBRES (Equilibrage d’ABR : AVL et misc.)

Exercice 3.21 – Mesure d’équilibre 🟅🟅

Concevoir un algorithme pour vérifier qu’un arbre est :

1. Parfaitement équilibré.
2. Partiellement équilibré.

Exercice 3.22 – Ajout dans un AVL : exécution d’un cas 🟅🟅

Exécuter l’algorithme d’ajout d’éléments dans un AVL pour les éléments 6, 4, 7, 1, 5, 3, 8, 9, 10, 2, pris dans cet ordre.

Exercice 3.23 – Ajout dans un AVL : conception de l’algorithme 🟅🟅🟅

Concevoir un algorithme récursif d’ajout d’un élément dans un AVL comme fils gauche ou fils droit d’un nœud feuille ou simple donné.

1. Vérifier que l’opération est licite, i.e. que le nœud donné est bien un nœud de l’arbre, et que l’ajout est possible, c'est-à-dire que la place de l’ajout n’est pas déjà occupée.
2. Si l’opération est licite, ajouter effectivement l’élément, et recalculer la « balance » de chacun des nœuds de l’AVL.
3. Modifier l’algorithme précédent en effectuant l’opération de rééquilibrage nécessaire au moment où un déséquilibre est constaté, avant de propager plus avant vers le sommet, la mise à jour de la balance de chacun des nœuds.

Exercice 3.24 – Ajout et équilibrage parfait 🟅🟅

1. Que faut-il modifier dans l’algorithme précédent pour que l’arbre soit maintenu parfaitement équilibré ?
2. Exécuter à nouveau l’exemple du 8.1.

Exercices supplémentaires pour les plus courageux

Exercice 3.25 – Retrait dans un AVL : exécution d’un cas 🟅🟅🟅

Exécuter l’algorithme de retrait d’éléments dans l’AVL obtenu à l’issue de 8.1 pour les éléments 6, 4, 7, 8, 9, 10, 2, 1, 5, 3, pris dans cet ordre.

Exercice 3.26 – Equilibrer un ABR 🟅🟅🟅🟅

Etant donné un ABR quelconque, et en vous inspirant des algorithmes précédents, concevoir un algorithme pour :

1. Transformer cet ABR en AVL (s’il ne l’est pas déjà).
2. Transformer cet ABR en arbre parfaitement équilibré.
1. ABR : Arbre Binaire de Recherche (EN – BST : Binary Search Tree) : ordre horizontal (*cf*. cours). [↑](#footnote-ref-1)
2. AB : Arbre Binaire (EN – BT : Binary Tree) : il ne répond à aucune structure d’ordre interne particulière. [↑](#footnote-ref-2)
3. Par convention, la profondeur de la racine est zéro. [↑](#footnote-ref-3)
4. Indice : pensez à produire une structure séquentielle, puis dans un second temps, l’analyser. [↑](#footnote-ref-4)
5. Transposer : échanger les positions [↑](#footnote-ref-5)
6. En C/C++, on appelle cette adresse un « pointeur ». [↑](#footnote-ref-6)
7. Rappel : deux LSC sont confluentes ssi elles contiennent une même sous-liste. En d’autres termes, ces deux listes forment un Y, commencent distinctement, mais se terminent comme une seule et même liste. Le point de confluence est le premier maillon commun aux deux listes. [↑](#footnote-ref-7)