IUT de Montpellier **TD4 : Arbres binaires de recherche (ABR)**
R3.02 Développement efficace

Commencez par récupérer le squelette de code sur le gitlab du cours `https://gitlabinfo.iutmontp.univ-montp2.fr/r3.02-dev-efficace`. Notez que les tests fournis ne sont *pas* suffisants, ils constituent une première étape de validation. Pensez donc à les compléter, et/ou à écrire un `main` avec quelques affichages pour contrôler ce qui se passe!

Rappel

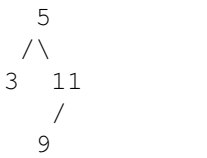
Un arbre `A` est un arbre binaire de recherche (ABR) ssi

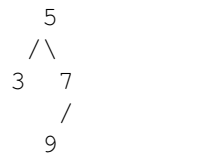
- `A` est vide, ou
- `A.filsG` et `A.filsD` sont des ABR, et
  - pour toute valeur x dans `A.filsG`, $x \leq A.val$, et
  - pour toute valeur x dans `A.filsD`, $A.val \leq x$

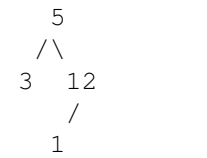
Toutes les méthodes du TD sont à écrire dans la classe `ABR`, et on supposera donc en prérequis que l'arbre courant est un ABR.

Exercices

Exercice 1. Quizz
Déterminez si les arbres ci-dessous sont des ABR.







Exercice 2. Recherche

Question 2.1.
Ecrire une méthode `boolean recherche(int x)` qui retourne vrai ssi x est dans l'arbre.

Exercice 3. Insertion

Question 3.1.
Ecrire une méthode `void insert(int x)` qui insert x dans l'arbre, et de telle sorte que l'arbre reste un ABR. Vous pouvez maintenant lancer le test fourni pour la recherche (qui utilisait `insert`)

Exercice 4. Tri
Pour cet exercice on aura besoin de la classe `Liste` du TD précédent (mais toutes les méthodes sont toujours à écrire dans la classe `ABR`).

Question 4.1.
Ecrire une méthode `Liste toListeTriee()` qui retourne dans une liste toutes les valeurs de l'arbre, triées par ordre croissant. ez utilisé s'appelle le *parcours infixe*.

Question 4.2.
Ecrire un constructeur `ABR(Liste l)` qui créer un ABR contenant tous les entiers de `l`. Pas besoin de se forcer à faire ce constructeur en récursif, vous pouvez faire une boucle pour insérer les éléments de `l` dans this.

Question 4.3.
A quoi correspond le programme suivant ?

```
Liste static prog(Liste l){
  return new ABR(l).toListeTriee();
}
```

Question 4.4.
La complexité de `toListeTriee` n'est pas optimale (elle est en $\mathcal{O}(n^2)$), car on utilise la méthode `concat`. Ecrire, sans utilise `concat`, la méthode `void toListeTrieeV2Aux(Liste l)`, qui modifie `l` pour y ajouter (en tête) tous les éléments de this, triés. Ecrire la méthode `void toListeTrieeV2()` "cliente" qui se contente d'appeler `toListeTrieeV2Aux`. La complexité de `toListeTrieeV2` est maintenant de $\mathcal{O}(n)$.

Exercice 5. Suppression

Question 5.1.
Ecrire une méthode `void suppr(int x)` qui supprime x (si il était présent) dans l'arbre, et de telle sorte que l'arbre reste un ABR. Si x n'est pas présent, alors cette méthode ne fait rien. Indication: le problème délicat sur lequel vous devez tomber est celui où x est contenu dans la racine de l'arbre, et où les deux sous arbres de sont non vides. Dans ce cas, pensez à la stratégie suivante (on aurait aussi pu faire une stratégie du même type à droite)(faites un dessin pour vous convaincre que cela fonctionne!) :

- cherchez m, le maximum du sous arbre gauche
- enlever m du sous arbre gauche
- dans la racine, remplacez x par m

Exercice 6. Vérification
Dans cet exercice on s'intéresse à la vérification qu'un arbre donné est un ABR. On prendra pour prérequis que l'arbre est un arbre bien construit (au sens des arbres habituels : soit les deux fils null, soit les deux fils non null), et la question sera de vérifier que les valeurs respectent bien les conditions d'un ABR.

Question 6.1.
Pourquoi sembe-t-il difficile d'écrire directement (sans fonctions auxiliaires) la méthode `boolean verifie()` ?

Comme souvent lorsque l'écriture directe pose problème, on peut envisager deux solutions : créer des méthodes auxiliaires, ou appliquer la technique de "recursion overloading". Dans cette technique, on résoud une version plus générale du problème de départ, soit en rajoutant des paramètres pour "donner plus de contexte à la récurrence", soit en demandant à la fonction de retourner plus d'informations que prévu. L'idée est que, bien que ce problème soit plus général, on réussira à le résoudre plus facilement puisque les appels récursifs font aussi une tâche plus générale! Notez que c'est la même technique qui nous a fait ajouter des paramètres pour la tour de Hanoï, pour compter les chemins dans une grille, pour la fractale du dragon.. Nous allons maintenant suivre ces deux solutions pour écrire deux versions différentes de l'algorithme de vérification.

Méthode 1 (ajout de fonctions auxiliaires).

Question 6.2.
Ecrire une méthode `int min()` qui retourne l'entier minimum de l'arbre (ou $+\infty$ si l'arbre est vide). Si ce n'est pas déjà fait (pour l'exercice sur la suppression), écrire aussi la fonction `int max()` qui retourne l'entier maximum de l'arbre (ou $-\infty$ si l'arbre est vide).

Question 6.3.
En utilisant `min` et `max`, écrire la fonction `boolean verifie()`.

Question 6.4.
L'inconvénient de cette méthode est la complexité. En effet, considérons un arbre "chemin" (avec toujours le fils gauche vide) ayant n valeurs : 1 à la racine, 2 à droite, 3 à droite de la droite, .., jusqu'à n. Quelle est la complexité de `verifie()` sur cet arbre ? (Cette complexité peut en fait être atteinte sur tout ABR, pas uniquement chemin).

Nous allons donc améliorer la complexité avec du recursion overloading.
Méthode 2 (recursion overloading : version 1). Pour trouver quels paramètres ajouter, on peut voir les choses ainsi : lorsque l'on veut vérifier qu'un arbre a est un ABR, on aimerait vérifier récursivement

- que a.filsG est un ABR, et que ses valeurs sont inférieures ou égales à un nouveau paramètre M (M=a.val ici)
- que a.filsD est un ABR, et que ses valeurs sont supérieures ou égales à un nouveau paramètre m (m=a.val ici)

Suivons par exemple ce qui se passe du côté gauche. Notons a'=a.filsG. On veut donc vérifier que a' est un ABR dont les valeurs sont inférieures ou égales à M. Pour cela, on va devoir (entre autre) vérifier que a'.filsD est bien un ABR dont les valeurs sont inférieures ou égales à M .. mais aussi supérieures à a'.val ! On voit donc qu'il va falloir donner deux paramètres lors de la vérification de a'.filsD (M et a'.val). Ces deux paramètres servent donc à contrôler l'intervalle dans lequel sont contenues les valeurs de l'arbre. Heureusement, la course aux paramètres s'arrête ici, et on obtient donc la spécification suivante.

Question 6.5.
Ecrire la méthode `boolean verifABR(int m, int M)` retourne vrai ssi l'arbre est un ABR, et que pour toute valeur x de l'arbre, on ait $m \leq x \leq M$. Ecrire la méthode `boolean verife()` qui appelle `verifABR`.

Méthode 2 (recursion overloading : version 2). Remarquez que l'on aurait tout aussi bien pu faire la "recursion overloading" de façon différente. On peut par exemple considérer une version 2 où l'on modifie seulement le type de retour plutôt que d'ajouter des paramètres supplémentaires :

Question 6.6.
Ecrire la méthode `int[] verifABRv2()` qui retourne null si l'arbre n'est pas un ABR, et qui sinon retourne (m, M), avec m la valeur minimale de l'arbre, et M la valeur maximale. A vous de décider quoi retourner quand l'arbre est vide.

Question 6.7.
Bonus : que pensez vous de la statégie suivante pour vérifier qu'un arbre est un ABR : on appelle `toListeTriee()`, et l'on vérifie que la liste obtenue est effectivement triée par ordre croissant.