Les arbres binaires de recherche (ABR)

Exercices obligatoires

A Code Runner

Faites les *codeRunner* sur les ABR. Suivez les niveaux : 1, 2 et puis 3.
Le document *CodeRunner _ABRTestes* donne une visualisation des arbres testés dans ce codeRunner.

B Implémentation d'un ABR d'entiers

Vous allez implémenter un arbre binaire de recherche dont chaque nœud contiendra un entier. Dans cet arbre, tout élément situé dans un sous-arbre de gauche devra être inférieur à l'élément racine de cet arbre. Tout élément situé dans un sous-arbre de droite devra lui être supérieur ou égal.

B1 Assurez-vous d'avoir bien compris l'algorithme d'insertion dans un ABR.
Voici le lien d'un programme de démo :
https://www.cs.usfca.edu/~galles/visualization/BST.html

Soyez actif.
Par exemple, commencez par dessiner *sur papier* un ABR en partant de l'arbre vide puis en faisant des ajouts (12 5 8 17 2 …) :
(Nb : ici, on ne s'intéresse pas encore à l'implémentation et aux classes *Noeud* et *Arbre*, représentez un nœud par un simple rond contenant sa valeur)
Ensuite vérifiez votre ABR en utilisant le programme de démo.

B2 Complétez la classe *ABRDEntiers*. Les « objets » contenus dans l'arbre sont des entiers.
La classe *TestABRDEntier*s permet de tester cette classe.

B3 Il existe plusieurs variantes pour l'algorithme de suppression dans un ABR.
Expliquez en français l'algorithme de suppression utilisé dans le programme de démo
https://www.cs.usfca.edu/~galles/visualization/BST.html

C Arbre binaire équilibré

C1 Ecrivez la méthode `hauteur()` de la classe *ArbreDEntiersPlus*.
Testez ces méthodes via la classe *TestHauteur.*
Cette classe teste les arbres mis en exemple dans le document de présentation *ABR.*

Exercices défis

B3 Implémentez la méthode `supprime()` de la classe *ABRDEntiers*.

Voici quelques indications :

Algorithme proposé :
Si le nœud qui contient l'entier à supprimer n'a pas de fils droit :
    Le nœud est remplacé par son fils droit (qui pourrait être vide).
    Attention c'est donc le parent du nœud à supprimer qui remplace un de ses 2 nœuds.
Sinon (le nœud qui contient l'entier à supprimer a un fils droit)
    L'entier à supprimer est remplacé par le plus petit entier retrouvé dans son fils droit et le nœud qui contient ce plus petit entier est supprimé.
    Attention, au final, ce n'est pas le nœud qui contient l'entier à supprimer qui est supprimé, cet entier est juste remplacé.

Pour suivre cet algorithme, nous vous proposons de d'abord écrire les méthodes `min()` et `supprimeMin()`. La classe de test permet de tester ces 2 méthodes séparément.

La méthode `min()` peut s'écrire de façon itérative, réfléchissez : où se trouve le plus petit entier dans un arbre?
La méthode `supprimeMin()` est plus simple que la méthode supprime(). Le nœud qui contient le plus petit entier n'a pas de fils gauche, réfléchissez !

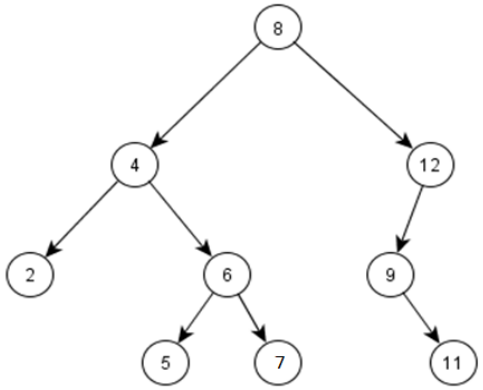
B4 ❤️ La méthode `taille()` de la classe *ABRDEntiers* a été implémentée en O(N).
Elle peut s'implémenter en O(1) en ajoutant un attribut taille.

```
public int taille(){
      return taille;
}
```

Ajoutez l'attribut `taille` et modifiez les méthodes `insere()`, `supprimeMin()` et `supprime()` afin qu'elles modifient l'attribut en cas de réussite de l'opération.
La classe *TestABRDEntiers* permet de tester cette méthode.

B5 La méthode `toArray()` de la classe *ABRDEntiers* va remplir une table avec les entiers contenus dans l'arbre. La taille logique de cette table doit correspondre à sa taille physique.
Les entiers devront y apparaître par ordre croissant :

Exemple :



2	4	5	6	7	8	9	11	12

Dans l'exemple ci-dessus, l'arbre contient 9 entiers, le sous-arbre de gauche en contient 5 et le sous-arbre de droite en contient 3.
L'attribut taille permet de trouver facilement l'indice où placer l'entier situé dans la racine de l'arbre.

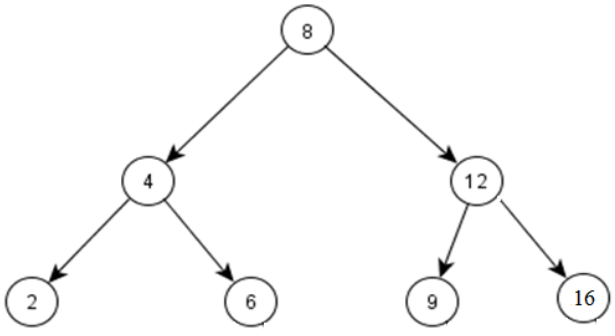
La méthode `toArray()` construit la table et appelle une méthode **`private`** à 4 paramètres : un nœud, la table, un indice de début et un indice de fin.
Ces indices permettent de délimiter la partie de la table où placer l'arbre (le sous-arbre) sur lequel la méthode est appelée.
La classe *TestABRDEntiers* permet de tester cette méthode.

Exercices supplémentaires

D Arbre binaire complet

L'arbre binaire pourrait être à certains moments équilibré.

Exemple :



MAIS on remarque qu'à ce stade, le prochain ajout rend l'arbre déséquilibré !
Il est intéressant de travailler avec un arbre binaire qui est équilibré à un niveau près.

Voici 2 caractéristiques intéressantes :

Un arbre binaire est dit **complètement rempli** s'il est de hauteur h et possède $2^{h+1} - 1$ nœuds, c'est-à-dire possède tous les nœuds possibles pour un arbre de hauteur h.

Un arbre binaire de hauteur h est dit **complet** si

- l'arbre est vide

ou

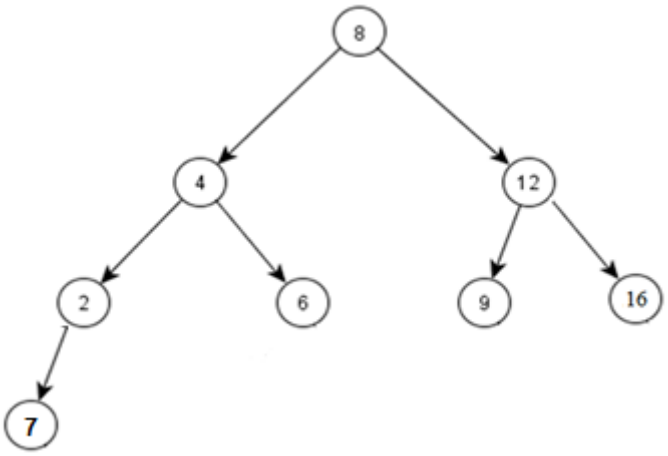
- son sous-arbre gauche est complet de hauteur h - 1 et son sous-arbre droit est complètement rempli de hauteur h - 2

ou

- son sous-arbre gauche est complètement rempli de hauteur h - 1 et son sous-arbre droit est complet de hauteur h - 1.

Ceci signifie qu'un arbre binaire est complet si toutes ses feuilles sont au même niveau ou sur deux niveaux adjacents et si toutes les feuilles situées au niveau le plus bas sont le plus à gauche possible.

Dans l'exemple ci-dessus, le seul arbre complet après ajout de 7 est :



D1 Sur papier, complétez au fur et à mesure un arbre binaire d'entiers en y ajoutant successivement les entiers de 1 à 10.
A chaque étape, l'arbre doit être complet. Il n'y a qu'une solution possible.
Vérifiez vos réponses avec celles du diaporama *D1Sol* qui se trouve sur moodle.

D2 Complétez le document *DocArbreComplet*.
Vérifiez vos réponses avec celles du document *D2Sol* qui se trouve sur moodle.

D3 Ecrivez les méthodes `estCompletementRempli()` et `estComplet()` dans la classe *ArbreDEntiersPlus*.
Testez ces méthodes via la classe *TestArbreComplet.*
Cette classe reprend les tests de l'exercice précédent.