ACT2 ARBRES BINAIRES DE RECHERCHE

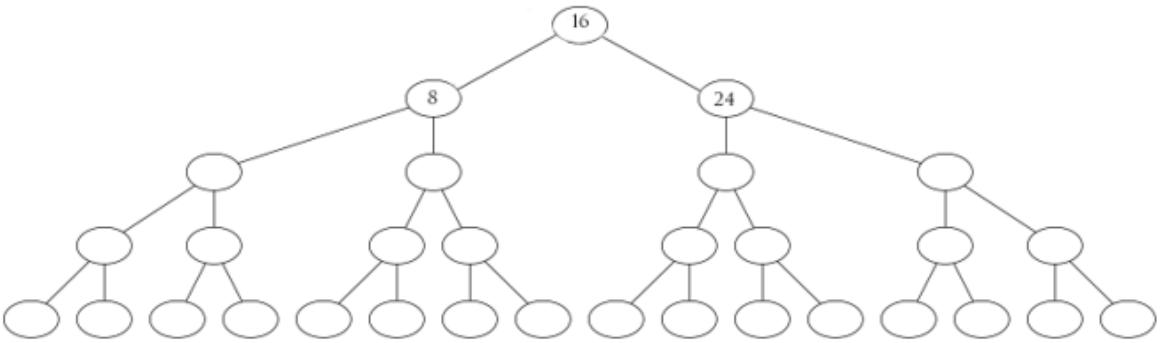
1. Définition

Choix d'un nombre le plus rapidement possible...

Pearl choisit un nombre entier compris entre 1 et 31. Bob cherche à trouver le plus rapidement possible ce nombre en posant des questions à Pearl.

1. Quelle stratégie vue en première permettra à Bob de trouver le plus rapidement possible le nombre choisi par Pearl ?

2. On peut représenter cette stratégie à l'aide d'un arbre binaire. Compléter cet arbre :



3. Combien d'étapes sont nécessaires au maximum pour trouver le nombre qu'a choisi Pearl ?

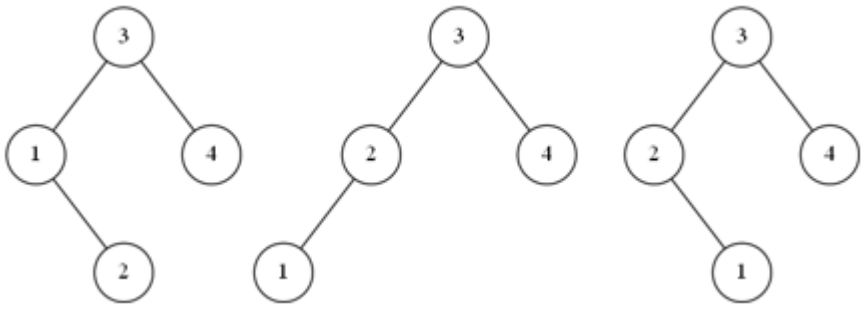
4. En vous inspirant de l'exemple précédent, construire un arbre binaire permettant de trouver rapidement une lettre dans l'alphabet.

A retenir :

Un **arbre binaire de recherche (ABR)** est un arbre binaire dont les nœuds contiennent des valeurs qui peuvent être comparées entre elles, et tel que, pour tout nœud de l'arbre, toutes les valeurs situées dans le sous-arbre gauche (resp. droit) sont plus petites (resp. plus grandes) que la valeur située dans le nœud.

On emploie également le terme **clé** à la place de valeur.

Exemples : Parmi les trois arbres suivants, lesquels sont des arbres binaires de recherche ?



Dans quel ordre sont visités les nœuds de ces arbres en parcours infixe ? Que remarque-t-on ?

Application : Un vétérinaire voudrait stocker les fiches médicales de ses patients, et, plutôt que d'utiliser un tableau ou une liste, on se propose d'utiliser un arbre binaire. La fiche contiendra différentes informations sur l'animal ; on utilisera son nom comme clé, que l'on triera selon l'ordre alphabétique croissant. Le vétérinaire reçoit sa première patiente, qui répond au nom de Gaufrette. Comme sa fiche sera le premier nœud de notre arbre, elle en devient automatiquement la racine. Puis le vétérinaire reçoit les animaux dans l'ordre suivant afin de les soigner : Charlie, Médor, Flipper, Bubulle et Augustin.

Construire l'arbre binaire de recherche associé à cette séquence.

2. Recherche dans un ABR

A retenir :

Pour chercher une valeur dans un ABR, on la compare à la valeur de la racine. Si elle est égale, on a trouvé, sinon on se dirige vers le sous-arbre gauche (si la valeur cherchée est plus petite), ou le sous-arbre droit (si la valeur est plus grande), et on recommence de manière récursive.

Cette recherche se décrit ainsi de manière récursive :

- Si l'arbre est vide, la valeur n'est pas dans l'arbre !
- Sinon, l'arbre contient au moins un nœud. On compare donc la valeur cherchée avec celle de la racine de l'arbre :
    - Si la valeur est plus petite, on continue la recherche dans le sous-arbre gauche (de manière récursive)
    - Si la valeur est plus grande, on continue la recherche dans le sous-arbre droit
    - Sinon on a trouvé la valeur.

Ecrire l'algorithme de la méthode recherche(self, arbre) qui renvoie True si la valeur est dans l'arbre, False sinon. L'implémenter dans une classe ABR, qui reprend les fonctionnalités de la classe AB.

Efficacité :

Lorsque les éléments sont répartis à peu près équitablement entre les sous-arbres, la recherche élimine environ la moitié des éléments à chaque étape. C'est le même principe que la recherche dichotomique dans un tableau **trié**.

De manière générale, le nombre d'étape ne peut pas dépasser la hauteur de l'arbre. Il est donc intéressant de construire un ABR de hauteur minimale.

3. Ajout d'un élément dans un ABR

Ajouter un élément dans un ABR repose que le même principe que la recherche d'un élément :

- Si le nouvel élément est plus petit que la valeur du nœud en cours, on va à gauche
- Si le nouvel élément est plus grand que la valeur du nœud en cours, on va à droite
- Quand on arrive à un arbre vide, on ajoute un nouveau nœud.

Ecrire l'algorithme correspondant à l'ajout d'un élément dans un ABR. L'implémenter dans la classe ABR.

4. Arbre équilibré

Comme on l'a dit plus haut, le coût de la recherche et de l'ajout dans un ABR dépend de sa structure. Il dépend de la hauteur de l'arbre. Dans le pire des cas, l'arbre est complètement linéaire et coût est alors proportionnel au nombre d'éléments : on est ramené à une recherche dans une liste triée, ce qui a peu d'intérêt !

Il est possible (mais compliqué et hors programme), de s'assurer, lors de la construction de l'arbre, que la hauteur ne sera pas trop grande. On réorganise pour cela les données au fur et à mesure, l'idée étant, pour chaque sous-arbre, de mettre environ la moitié des données dans chaque sous-arbre gauche et droit. On obtient alors une hauteur logarithmique, c'est-à-dire qu'il existe une constante C telle que $h \leq Clog_2(N)$, où h est la hauteur de l'arbre et N le nombre de données. On parle alors d'**arbre équilibré**.

Les opérations de recherche et d'ajout ont dans ce cas également une complexité logarithmique, ce qui les rend très efficaces.