Arbres de Recherche

Arbres binaires de Recherche (Binary Search Trees)

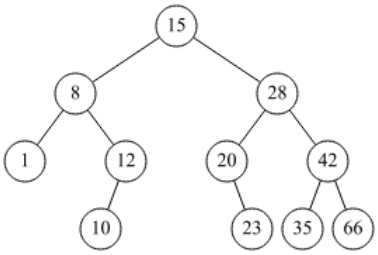

Figure 1 – Arbre binaire de recherche (ABR) `B1`

1 Les classiques

Exercice 1.1 (Recherches)

1. (a) Où se trouvent les valeurs minimum et maximum dans un arbre binaire de recherche non vide ?
   (b) Écrire les deux fonctions `minBST(`B`)` (en récursif) et `maxBST(`B`)` (en itératif), avec B un ABR non vide.
2. Écrire une fonction qui recherche une valeur x dans un arbre binaire de recherche. La fonction retournera l'arbre contenant x en racine si la recherche est positive, la valeur `None` sinon.
   Deux versions pour cette fonction : récursive et itérative !

Exercice 1.2 (Insertion en feuille)

1. En utilisant le principe de l'ajout aux feuilles, construisez, à partir d'un arbre vide, l'arbre binaire de recherche obtenu après ajouts successifs des valeurs suivantes (dans cet ordre) :

$$13, 20, 5, 1, 15, 10, 18, 25, 4, 21, 27, 7, 12$$

2. Écrire une fonction récursive qui ajoute un élément dans un arbre binaire de recherche.
   Bonus : écrire la fonction d'insertion en itératif.

Exercice 1.3 (Suppression)

Écrire une fonction récursive qui supprime un élément dans un arbre binaire de recherche.

Exercice 1.4 (Insertion en racine)

1. En utilisant le principe de l'ajout en racine, construisez, à partir d'un arbre vide, l'arbre binaire de recherche obtenu après ajouts successifs des valeurs suivantes (dans cet ordre) :

$$13, 20, 5, 1, 15, 10, 18, 25, 4, 21, 27, 7, 12$$

2. Écrire la fonction qui ajoute un élément en racine dans un *arbre binaire de recherche*.

2 Applications

Exercice 2.1 (Second minimum)

Écrire la fonction `second_min(`B`)` qui retourne la seconde plus petite valeur (2ème dans l'ordre croissant) de l'arbre binaire de recherche `B` ou la valeur `None` si elle n'existe pas. Toutes les clés de l'arbre binaire de recherche `B` sont supposées distinctes.

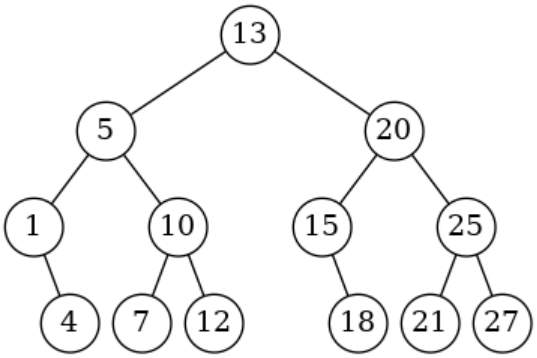

Figure 2 – B2

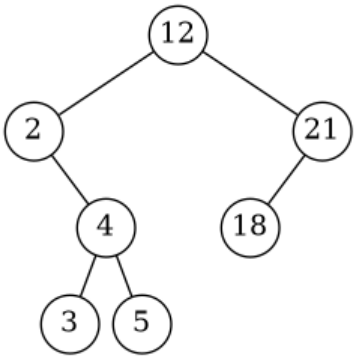

Figure 3 – B3

Exemples d'applications :

```
>>> second_min(B1)
8
>>> second_min(B2)
4
>>> second_min(B3)
3
```

Exercice 2.2 (Paire)

Écrire la fonction `both_x(`B`, `x`)` qui retourne une paire d'entiers `(inf, sup)` telle que :
— `inf` est le nombre d'éléments inférieurs à `x` dans l'arbre binaire de recherche `B`
— `sup` est le nombre d'éléments supérieurs ou égaux à `x` dans l'arbre binaire de recherche `B`

Exemples d'applications :

```
>>> both_x(B2, 20)
(9, 4)
>>> both_x(B2, 27)
(12, 1)
>>> both_x(B2, 21)
(10, 3)
>>> both_x(B3, 12)
(4, 3)
>>> both_x(B3, 23)
(7, 0)
>>> both_x(B3, 7)
(4, 3)
```

Exercice 2.3 (Plus proche ancêtre commun)

Le plus proche ancêtre* commun de deux noeuds d'un arbre binaire de recherche est le noeud le plus bas dans l'arbre binaire de recherche (le plus profond) ayant ces deux noeuds pour descendants.

* On considère qu'un noeud peut être son propre ancêtre.

Écrire la fonction `lca(`B`, `x`, `y`)` avec :
— B, un arbre binaire de recherche supposé non vide et dont toutes les clés sont uniques
— x et y, deux entiers qui correspondent aux clés de deux noeuds de B
et qui retourne la clé du plus proche ancêtre commun des noeuds contenant x et y.

On suppose de plus que l'arbre binaire de recherche B **contient** x **et** y **et que** $x \neq y$.

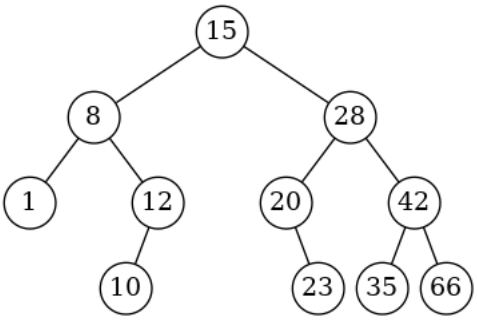

Figure 4 – Arbre binaire de recherche `B1`

Exemples d'applications avec $B1$ l'arbre binaire de recherche de la Figure 7 :

```
>>> lca(B1, 10, 15)
15
>>> lca (B1, 20, 66)
28
>>> lca(B1, 8, 10)
8
>>> lca(B1, 10, 42)
15
```

3 Bonus : `BinTreeSize`

Exercice 3.1 (Médian)

On s'intéresse à la recherche de la valeur médiane d'un arbre binaire de recherche B, c'est à dire celle qui, dans la liste des éléments en ordre croissant, se trouve à la place $(taille(B) + 1)\ DIV\ 2$.

Pour cela, on veut écrire une fonction `nthBST(`B`, `k`)` qui retourne le nœud contenant le $k^{ème}$ élément de l'ABR B (dans l'ordre des éléments croissants). Par exemple, l'appel à `nthBST(`B_1`, 3)` avec B_1 l'arbre de la figure 1 nous retournera le nœud contenant la valeur 10.

1. **Étude abstraite :**

   On ajoute à la définition abstraite des arbres binaires l'opération *taille*, définie comme suit :

   opérations
   taille : ArbreBinaire → Entier

   axiomes
   taille (arbrevide) = 0
   taille (<o, G, D>) = 1 + *taille* (G) + *taille* (D)

   Donner une définition abstraite de l'opération *kieme* (utilisant obligatoirement l'opération *taille*).

2. **Implémentation :**

   Les fonctions à écrire utilisent des arbres binaires avec la taille renseignée en chaque noeud : `BinTreeSize` (voir annexe).

   — Écrire la fonction `nthBST(`B`, `k`)` qui retourne l'arbre contenant le $k^{ème}$ élément en racine. On supposera que cet élément existe toujours : $1 \leq k \leq taille(B)$.

   — Écrire la fonction `median(`B`)` qui retourne la valeur médiane de l'arbre binaire de recherche B s'il est non vide.

Annexes : types, méthodes et fonctions autorisées

Les arbres binaires

— L'arbre vide est `None`

— L'arbre non vide est (une référence sur) un objet de la classe `BinTree` avec 3 attributs : `key`, `left`, `right`.

- `B` : classe `BinTree`
- `B.key` : contenu du nœud racine
- `B.left` : le sous-arbre gauche
- `B.right` : le sous-arbre droit

```
class BinTree:
    def __init__(self, key, left, right):
        self.key = key
        self.left = left
        self.right = right
```

Dans la classe `BinTreeSize` chaque noeud possède en plus l'attribut `size` : la taille de l'arbre dont le nœud est racine.

Fonctions et méthodes autorisées

Les fonctions `min` et `max`, mais uniquement avec deux valeurs entières !

Aucun opérateur n'est autorisé sur les listes (`+`, `*`, `==` ...).

Vos fonctions

Vous pouvez écrire des fonctions 'intermédiaires' / 'supplémentaires', dans ce cas vous devez donner leurs spécifications : on doit savoir ce qu'elles font.

Dans tous les cas, la dernière fonction écrite doit être celle qui répond à la question.