Arbres binaires de Recherche (Binary Search Trees)

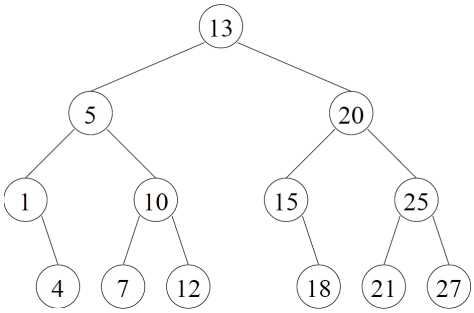

FIGURE 1 – Arbre binaire de recherche (BST)

1 Outils et mystères

Exercice 1.1 (Liste ↔ ABR)

1. Écrire une fonction qui construit une liste triée des éléments contenus d'un arbre binaire de recherche.
2. Écrire une fonction qui construit un arbre binaire de recherche *équilibré* à partir d'une liste triée.

Exercice 1.2 (Test)

Écrire une fonction qui vérifie si un arbre binaire est bien un arbre binaire de recherche.

Exercice 1.3 (Arbres et mystère – *P2 - June 2017*)

```
1             def __makeTree(n, i, cur):
2                 if i > n:
3                     return (None, cur)
4                 else:
5                     (left, cur) = __makeTree(n, 2*i, cur)
6                     key = cur+1
7                     (right, cur) = __makeTree(n, 2*i+1, key)
8                     return (binTree.BinTree(key, left, right), cur)
9
10            def makeTree(n):
11                (B, val) = __makeTree(n, 1, 0)
12                return B
```

1. La fonction `makeTree(n)` construit (et retourne) un arbre binaire. Dessiner l'arbre résultat lorsque $n = 13$.
2. On appelle `makeTree(n)` avec `n` un entier strictement positif.
   Donner deux propriétés de l'arbre retourné.

2 Les classiques

Exercice 2.1 (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`)` et `maxBST(`B`)`, 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 chaque fonction : récursive et itérative !

Exercice 2.2 (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 qui ajoute un élément dans un arbre binaire de recherche.

Exercice 2.3 (Suppression)

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

Exercice 2.4 (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*.

3 `BinTreeSize` – *P2# – Jan. 2018*

Pour les deux exercices qui suivent, on ajoute une nouvelle implémentation des arbres binaires dans laquelle chaque nœud contient la taille de l'arbre dont il est racine.
Voir td 4

Exercice 3.1 (Ajout avec mise à jour de la taille)

Écrire une fonction **récursive** qui ajoute un élément en feuille dans un arbre binaire de recherche sauf si celui-ci est déjà présent.
L'arbre est représenté par le type `BinTreeSize`, il faut donc mettre à jour, lorsque nécessaire, le champ *size* en chaque nœud de l'arbre.

Exercice 3.2 (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 1 nous retournera le nœud contenant la valeur 5.

1. **Étude abstraite :**
   On ajoute à la définition abstraite des arbres binaires (donnée en annexe) 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 abstraire 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 nœud (`BinTreeSize`).
   – 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.