Arbres binaires de recherche

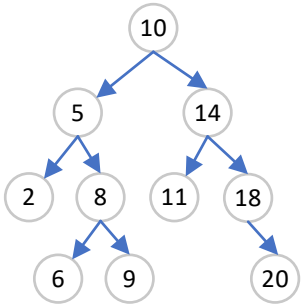
Définitions

Un **arbre binaire de recherche** (ABR ou BST en anglais, Binary Search Tree) est un arbre binaire dans lequel :

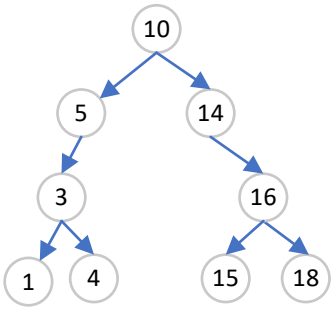
- chaque nœud possède une clé, souvent l'étiquette du nœud, mais on peut stocker dans chaque nœud un couple (clé, étiquette) ;
- les clés sont comparables entre elles (généralement par la relation <=) ;
- la clé d'un nœud est supérieur ou égale à la clé de chaque nœud de son sous arbre-gauche ;
- la clé d'un nœud est strictement inférieure à la clé de chaque nœud de son sous arbre-droit.

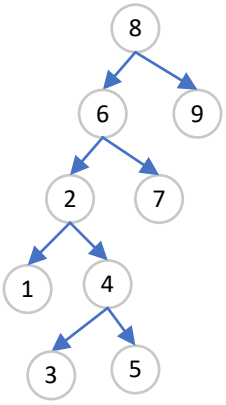
Un arbre binaire est complètement **équilibré** si tous ses niveaux sont complets, sauf éventuellement le dernier.

Exemples



ABR équilibré





ABR non équilibrés

Petit exercice : modifier l'arbre du milieu pour obtenir un arbre équilibré

Rechercher une clé dans un ABR

Rechercher une clé dans un ABR équilibré est semblable à une recherche dichotomique dans un tableau trié : on divise la zone de recherche en deux à chaque comparaison. La complexité est donc la même, en $O(\log_2(n))$ où n est la taille de l'arbre.

Si l'arbre n'est pas équilibré, la recherche sera moins efficace, avec une complexité en $O(n)$ dans le pire des cas, un arbre filiforme ou un arbre peigne (exemple de droite).

Insérer une clé dans un ABR

On parcourt l'arbre pour déterminer la position où insérer la nouvelle clé, comme dans le cas d'une recherche de clé, donc cette opération est en $O(\log_2(n))$ dans le cas d'un arbre équilibré. Ensuite, l'insertion s'effectue en temps constant, contrairement à l'insertion dans un tableau trié qui nécessite de décaler les éléments (complexité linéaire).

Après l'insertion de nouveaux éléments, il se peut que l'arbre ne soit plus équilibré. En pratique, un bon algorithme effectue une insertion suivie de modifications, le tout avec une complexité de $O(\log_2(n))$, pour garder l'arbre suffisamment équilibré afin que les insertions et recherches suivantes s'exécutent toujours en $O(\log_2(n))$.

Exercice

Reproduire l'arbre de l'exemple de gauche en insérant successivement les clés 4, 12, 3, 16 et 19 sans déplacer les valeurs déjà existantes. On ne se préoccupera pas de conserver un arbre équilibré.

Arbres binaires de recherche : travaux pratiques

Ouvrir les fichiers abr.py et outils_abr.py. S'assurer que le fichier arbre.py est dans le même dossier. outils_abr.py contient des fonctions que vous utiliserez pour tester vos codes :

- genere_abr_float(h, p=0.9, a=0, b=1) renvoie un ABR de hauteur h dont les étiquettes/clés sont des flottants de l'intervalle [a, b]. Le flottant p est la probabilité qu'un nœud ait deux branches.
- genere_abr_int(h) renvoie un ABR de hauteur h dont les étiquettes/clés sont des entiers pas forcément différents.
- montrer_arbre(arbre) permet d'afficher un ABR en utilisant graphviz. Cette fonction représente correctement les ABR avec plusieurs clés identiques et affiche aussi les arbres vides pour distinguer si un fils unique est un fils gauche ou un fils droit.

Rechercher une clé dans un ABR

Écrire dans le fichier abr.py la fonction d'entête **def** recherche_abr(abr, e) -> bool: qui renvoie True si la clé e est présente dans l'ABR abr et False sinon.

Insérer une clé dans un ABR

Écrire la fonction d'entête **def** insere_abr(abr, e) -> None: qui insère la clé e dans l'ABR abr.

- On parcourra l'ABR comme lors d'une recherche jusqu'à atteindre la position où insérer le nouveau nœud, en remplacement d'un arbre vide. Pour cela, on se permettra exceptionnellement de violer le principe d'encapsulation en remplaçant abr.g ou abr.d par le nouveau nœud.
- On ne se préoccupera pas de l'équilibrage de l'arbre. Les algorithmes d'insertion qui conservent un arbre équilibré sont hors-programme.

Écrire une autre fonction qui n'insère la clé que si elle n'est pas déjà présente dans l'arbre.

J'ai préféré ne pas modifier la classe Noeud mais il aurait été beaucoup plus propre d'écrire une méthode de cette classe. Idéalement, on aurait fait une sous-classe Noeud_abr par exemple, héritant de la classe Noeud en rajoutant les méthodes spécifiques aux ABR, mais l'héritage est hors-programme.

On peut maintenant utiliser les fonctions genere_abr_lst et genere_abr_alea qui permettent de générer des ABR en insérant successivement tous les éléments d'une liste.

Trier une liste en utilisant un ABR

a. Créer une liste de 50 entiers aléatoires distincts, puis générer un ABR dont les clés sont ces entiers.
b. Utiliser la fonction montrer_arbre pour l'afficher.
c. Quel parcours permet d'afficher ces nombres dans l'ordre croissant ?
d. Écrire une fonction *nom_du_parcours*(abr, lst_triee) prenant en argument un ABR et une liste vide, et ajoutant les clés de l'ABR à la liste dans l'ordre.
e. Écrire la fonction tri_abr(lst) renvoyant une copie triée de la liste lst en utilisant un ABR.

Vérification (plus difficile)

Écrire une fonction verification_abr renvoyant True si un ABR passé en argument est valide, c'est-à-dire si la clé de chaque nœud est supérieure ou égale à la clé de chaque nœud de son sous arbre-gauche et inférieure ou égale à la clé de chaque nœud de son sous-arbre droit.