EA4 – Éléments d'algorithmique
TP n° 4 : arbres binaires de recherche

Dans ce TP, on représente un arbre binaire par une liste `[r, T]`, où `T` est une liste de nœuds, et `r` est l'indice de la racine dans la liste `T`. Chaque nœud est une liste de taille 4 au format suivant :

`[étiquette, indFilsGauche, indFilsDroit, indPère]`,

où les trois derniers champs sont les indices des nœuds correspondants dans la liste (ou `None` s'ils n'existent pas).
Par convention,

- la racine de l'arbre est son propre père (et c'est le seul nœud ayant cette propriété) ;
- tous les arbres manipulés seront des arbres binaires ***complétés*** par des pseudo-feuilles, vides, de la forme `[None, None, None, i]`.

Par ailleurs on acceptera la présence de cases vides (`None`) dans le tableau `T`. En particulier l'arbre vide pourra être représenté par tous les couples de la forme `(r, T)` où la seule case de `T` différente de `None` est `T[r]`, de la forme `[None, None, None, r]`.
Dans la suite, chaque nœud de l'arbre est désigné par son indice dans la liste.

Le fichier `tp4.py` (importé dans `tp4_ex1.py`) définit un certain nombre d'utilitaires pour accéder aux caractéristiques d'un arbre ou d'un nœud, les tester ou les modifier. Il définit également :

- une fonction `arbreBinaireDeFichier(fichier)` qui construit l'arbre décrit par le fichier dont le nom est passé en paramètre ;
- plusieurs arbres pour les tests ;
- une fonction `dessineArbreBinaire(arbre)` qui crée un fichier `arbre.pdf` représentant l'arbre passé en paramètre.

Dépendances : la fonction `dessineArbreBinaire(arbre)` nécessite d'avoir installé `graphviz` [1] sur son ordinateur, qui permet (notamment) d'utiliser la commande `dot` dans le terminal.

Exercice 1 : recherches et ajouts dans un ABR

1. Écrire une fonction `estUnABR(arbre)` qui renvoie `True` si les étiquettes des nœuds de l'`arbre` vérifient les conditions d'un ABR ***en temps linéaire en la taille de l'arbre***.
2. Écrire des fonctions ***non récursives*** `minimumABR(arbre, noeud=None)` et `maximumABR(arbre, noeud=None)` qui retournent respectivement l'indice du noeud d'étiquette minimale ou maximale dans le sous-`arbre` de racine `noeud`, ou dans l'`arbre` entier si `noeud==None`.
3. Écrire une fonction `rechercheABR(arbre, elt)` retournant l'indice d'un nœud contenant `elt`, s'il en existe un, et celui de la feuille vide à laquelle la recherche aboutit, sinon.
4. À l'aide de la fonction auxiliaire `rechercheABR(arbre, elt)` définie ci-dessus, écrire (et tester) une fonction `insertionABR(arbre, elt)` qui ne fait rien si l'`arbre` contient `elt`, et l'insère à la bonne place dans l'`arbre` sinon.
5. Le test `testInsertion2` construit un ABR par insertions successives d'éléments à partir d'un arbre vide, puis crée les fichiers d'extension `.dot` et `.pdf` contenant un dessin de l'ABR. Sur le même modèle, vous pouvez construire d'autres ABR par ajouts successifs d'éléments. Représenter le résultat en utilisant la fonction `dessineArbreBinaire()`.

1. Pour l'obtenir avec votre gestionnaire de paquets : `apt-get install graphviz` (remplacer `apt-get` par `yum`, `brew`,... bref votre gestionnaire de paquets si ce n'est pas `apt`)

Exercice 2 : génération aléatoire par insertions successives

Dans cet exercice, on souhaite mesurer expérimentalement la hauteur moyenne d'un arbre construit par insertions successives à partir d'une permutation aléatoire de taille n.

1. Écrire (et tester) une fonction `hauteur(arbre)` qui retourne la hauteur de l'`arbre`. On rappelle que la hauteur de l'arbre vide est -1.
2. Écrire une fonction `genererABRparInsertion(perm)` qui construit un arbre binaire de recherche par insertions successives des éléments de la permutation `perm`.
3. En utilisant les fonctions précédentes, ainsi que la fonction `permutation(n)` (qui renvoie une permutation aléatoire de taille `n`), écrire une fonction `statsHauteursABRparInsertion(n, m)` qui renvoie la liste des hauteurs de `m` arbres de taille `n` construits aléatoirement selon ce procédé.
4. À l'aide de la fonction `tracer(limite, pas, m)` fournie, observer la distribution des hauteurs des arbres construits de cette manière.

Exercice 3 : suppressions

Écrire (et tester) une fonction `suppressionABR(arbre, elt, alea=False)` qui supprime le nœud d'étiquette `elt` s'il existe selon la méthode vue en cours. Si `alea` vaut `False`, lorsque le nœud contenant l'élément à supprimer a 2 fils, on le remplacera par son ***prédécesseur***. Lorsque `alea` vaut `True` en revanche, on choisira à pile ou face le prédécesseur ou le successeur.
Chaque suppression d'élément entraîne la suppression de 2 sommets de l'ABR, qui peuvent être remplacés par `None` dans le tableau. Pour limiter l'encombrement en mémoire, vous pouvez cependant faire appel à la fonction `nettoieTab(arbre, aSupprimer)` fournie dans `tp4.py`, qui réordonne les sommets pour pouvoir raccourcir le tableau en supprimant les noeuds dont les indices sont passés dans la liste `aSupprimer`.

Exercice 4 : génération aléatoire par insertions successives puis suppressions

Nous pouvons maintenant expérimenter des modèles un peu plus réalistes d'ABR aléatoires, obtenus à partir d'insertions mais également de suppressions.

1. Écrire une fonction `genererABRparInsPuisSup(perm)` qui construit un arbre de taille n selon le procédé suivant :
   - construire un arbre de taille n^2 par insertions successives à partir d'une permutation `perm` de taille n^2 ;
   - supprimer $n^2 - n$ éléments de l'arbre, chacun choisi uniformément parmi les éléments restants.
2. Écrire une fonction `statsHauteursABRparInsPuisSup(n, m)` qui produit la liste des hauteurs de `m` arbres de taille `n` obtenus par ce procédé à partir d'une permutation aléatoire, puis examiner les courbes obtenues.
3. Écrire une fonction `genererABRparInsEtSup(permins, permsup)` qui construit un arbre de taille au plus n à partir de deux permutations de taille n en alternant aléatoirement entre l'insertion d'un élément de `permins` et la suppression d'un élément de `permsup` (qui sera sans effet si l'élément n'est pas encore dans l'arbre). La fonction retournera un couple formé de l'arbre et de sa taille.
4. Écrire une fonction `statsHauteursABRparInsEtSup(n, m)` qui renvoie un tableau de couples constitués des tailles et hauteurs de `m` arbres obtenus par le procédé précédent à partir de deux permutations aléatoires de taille `2n`, puis examiner les courbes obtenues.