tp_l8dn003_21_01

March 17, 2021

0.1 TP n°1: arbres binaires de recherche

Les arbres binaires de recherche sont des structures de données arborescentes. Ce sont des arbres binaires, c'est-à-dire que chaque noeud a au plus deux fils ; le principe général est que chaque noeud contient une donnée de manière à ce que les données soient ordonnées. Nous allons travailler avec des ABR dont les données stockées sont des mots.

Soit un ABR (r, g, d) où r est la racine de l'arbre, g et d sont les deux sous-arbres (fils *gauche* et fils *droit*) de r (qui sont par définition des ABR). Alors tous les mots du sous-arbre g sont inférieurs ou égaux au mot associé à la racine (ordre lexicographique), et tous les mots du sous-arbre d sont strictement supérieurs.

Exemple: on peut représenter un arbre par un tuple (r,g,f) où r est la racine, qu'on va représenter par le mot qu'elle contient, et g et f sont d'autre tuples. Par convention, une feuille (un noeuds sans fils) sera représenté par un tupe de la forme (m, None, None) où m est une chaîne de caractère.

Dans le suite de ce TP, on va utiliser un ABR pour représenter une liste de mots (ou de mots-formes), et étudier d'abord le bénéfice de ce genre de structure pour la recherche, puis s'intéresser à la façon de contruire un ABR.

0.1.1 Recherche dans un ABR

La recherche d'un mot dans un ABR exploite le principe d'organisation vu plus haut, la structure même de l'arbre permet une recherche essentiellement dichotomique: si le mot qu'on cherche n'est pas la racine de l'arbre, alors il suffit de le comparer à la racine de l'arbre pour savoir dans lequel des deux sous-arbres on fait la recherche. On réitère le processus jusqu'à tomber sur un (sous-)arbre vide (auquel cas on peut conclure que le mot n'est pas dans l'ABR).

Implémenter la recherche d'un mot dans un ABR, préférentiellement avec un algorithme récursif. (On peut aussi raisonner de façon non récursive, mais ce n'est pas forcément plus facile).

On notera que si l'ABR est bien équilibré (c'est-à-dire que pour chaque sous-arbre, le nombre de mots du fils gauche est peu différent du nombre de mots du fil droit), un tel algorithme va mettre un temps proportionnel à $\log_2 N$ où N est le nombre de mots.

0.1.2 Remplissage d'un ABR

Insertion d'un mot Un des intérêts des ABR est que l'algorithme utilisé pour ranger un mot dans l'arbre est essentiellement le même que l'algorithme de recherche. En repartant de l'algorithme de recherche de la question précédente, écrire une fonction qui insère le mot qu'on lui donne à la bonne place dans l'arbre. On va faire l'hypothèse que si le mot est déjà présent, on ne l'ajoute pas.

Mesure de la profondeur Ecrire une fonction qui, étant donné un ABR, donne la profondeur de l'arbre, la profondeur étant définie ici comme la longueur en arcs du plus long chemin d'une feuille vers la racine.

Insertion du "lexique" d'un texte En partant d'une liste de mots-formes extraite d'un texte, on peut créér un ABR en insérant à son tour chaque mot dans l'ABR.

En ordonnant de différentes manières les mots insérés, vérifier que dans le pire des cas, la profondeur de l'arbre correspond au nombre de mots insérés. Quel est le moyen d'insérer les mots pour avoir un arbre ayant la plus petite profondeur possibles ?

Suppression d'un mot Un ABR est une structure de données, donc on veut pouvoir réaliser les 3 fonctions habituelles d'une SdD : recherche, insertion, suppression. Il reste donc à définir une fonction de suppression, qui ôte d'un ABR le mot passé en paramètre (si ce dernier est présent dans l'arbre). Cette suppression implique une réorganisation locale de l'arbre.

0.1.3 [Bonus] Arbre binaire équilibré (AVL)

En général, il n'est pas intéressant d'être obligé de trier des données avant de remplir un ABR. Alors pour réduire le risque d'avoir un arbre de trop grande profondeur (ie mal équilibré), on peut utiliser une méthode de restructuration dynamique (et locale) d'un ABR. L'idée est que lorsque le mot qu'on insère risque d'augmenter la profondeur de l'arbre, on tente de réorganiser localement les noeuds pour éviter cette augmentation de profondeur. Les premières approches qui ont été proposées sont décrites sout le nom d'arbre AVL https://fr.wikipedia.org/wiki/Arbre_AVL.

Implémenter la méthode AVL en vous inspirant de la description de la page wikipedia (un peu sommaire en français, un peu trop abstraite pour la page anglaise). Vous pouvez aussi vous inspirer de la proposition qui est donnée ici: https://blog.coder.si/2014/02/how-to-implement-avl-tree-in-python.html (mais elle fait usage de notations "objet").

[4]:
```
# ABR équilibré contenant les 8 mots
# sois sage ma douleur tiens toi plus tranquille
minilex = ("sois", ("ma", ("douleur", None, None), ("sage", ("plus", None,␣
 ↪None), None)),
           ("toi", ("tiens", None, None), ("tranquille", None, None)))
```

[5]:
```
# Un petit programme pour afficher de façon un peu plus lisible un ABR
# Algorithme de "pretty print" classique.
def pretty_abr(t, prof=0):
    if t is not None:
        (r,fg,fd) = t
        print("%s%s" % (" "*prof, r))
        pretty_abr(fg,prof+2)
        pretty_abr(fd,prof+2)
#    else:
#        print("-"*prof, "nil")
```

[6]:
```
pretty_abr(minilex)
```

```
sois
  ma
    douleur
    sage
      plus
  toi
    tiens
    tranquille
```