

# tp\_l8dn003\_21\_01-ABR-sketch

April 15, 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'autres tuples. Par convention, une feuille (un noeuds sans fils) sera représenté par un tupe de la forme  $(m, \text{None}, \text{None})$  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.

```
[1]: # ABR équilibré contenant les 8 mots
# sois sage ma douleur tiens toi plus tranquille (ordre quelqconque)
minilex = ("sois", ("ma", ("douleur", None, None), ("sage", ("plus", None,
→None), None)),
          ("toi", ("tiens", None, None), ("tranquille", None, None)))
```

```
[2]: # 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:
        # si on veut afficher les feuilles vides
        print("%s%s" % (" "*prof, "nil"))
```

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

sois

```

ma
  douleur
    nil
    nil
  sage
    plus
      nil
      nil
    nil
toi
  tiens
    nil
    nil
  tranquille
    nil
    nil

```

### 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.

```
[4]: def recherche(abr, mot):
      if abr is None: return False
      (r,fg,fd) = abr
      if mot == r: return True
      if mot <= r:
          return recherche(fg, mot)
      else:
          return recherche(fd, mot)
```

### 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.

```
[5]: # Principe : on ne réinsère pas un mot déjà présent
```

```
def insere_mot(mot, abr):  
    if abr is None:  
        return [mot, None, None]  
    (r,fg,fd) = abr  
    if mot == r: return abr  
    if mot <= r:  
        return [r, insere_mot(mot, fg), fd]  
    else:  
        return [r, fg, insere_mot(mot, fd)]
```

```
[16]: # On peut aussi utiliser la fonction d'insertion pour créer un arbre:
```

```
# il faut juste commencer avec l'arbre le plus petit possible: None  
a = None  
for m in "la tribu prophétique aux prunelles ardentes hier s'est mise en route".  
    ↪split():  
    a = insere_mot(m, a)  
pretty_abr(a)
```

```
la  
  aux  
    ardentes  
      nil  
      nil  
    hier  
      en  
        nil  
        nil  
      nil  
    tribu  
      prophétique  
        mise  
          nil  
          nil  
        prunelles  
          nil  
        s'est  
          route  
            nil  
            nil  
          nil  
        nil  
      nil
```

**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.

```
[7]: def profondeur(abr):
      if abr == None: return 0
      (r,fg,fd) = abr
      return max(profondeur(fg), profondeur(fd)) + 1
```

```
[17]: print(profondeur(a))
      print(profondeur(minilex))
```

6  
4

On remarque que la profondeur de `a`, qui ne contient que 11 mots, est plus grande que celle de `minilex`, qui en contient 15. On peut aussi noter que les profondeurs obtenues (6 ou 5) sont toutes les deux supérieures à la valeur théorique minimale: puisque 11 et 15 sont compris entre  $2^3$  et  $2^4$  alors on sait qu'un arbre binaire de profondeur 4 peut contenir tous les mots.

On peut observer que la profondeur dépend bien sûr du nombre de mots, mais aussi de façon cruciale de la façon dont les mots sont insérés.

```
[9]: # Création d'un nouvel arbre avec les mêmes mots, insérés dans l'ordre
      ↪lexicographique
      b = None
      for m in sorted("la tribu prophétique aux prunelles ardentes hier s'est mise en
      ↪route".split()):
          b = insere_mot(m,b)
      pretty_abr(b)
      print("Arbre de profondeur %d." % profondeur(b))
```

```
ardentes
  nil
  aux
    nil
    en
      nil
      hier
        nil
        la
          nil
          mise
            nil
            prophétique
              nil
              prunelles
                nil
                route
                  nil
                  s'est
                    nil
                    tribu
```

```
nil
nil
```

Arbre de profondeur 11.

Cet arbre “en peigne” correspond au cas le pire: chaque nouveau mot inséré est venu s’insérer comme fils droit de l’arbre précédent. La profondeur correspond alors au nombre total de mots ! La recherche d’un mot dans un tel arbre va coûter exactement autant de comparaisons que si on avait simplement les mots dans une liste.

Pour avoir un arbre équilibré, dans quel ordre faut il insérer les mots ? Il faut les insérer en choisissant comme racine le mot “médian”: celui qui se trouve au milieu de la liste triée.

```
[10]: # La liste de mots reçue en paramètre est d'abord triée,
# puis on construit un abr en choisissant la meilleure racine possible
# On n'utilise pas la fonction insere_mot(), mais quand la liste ne
# contient qu'un mot ou deux, on crée directement l'arbre
def abr_equilibre(l):
    if len(l) == 1:
        return [l[0], None, None]
    if len(l) == 2:
        return [l[0], None, [l[1], None, None]]
    ls = sorted(l)
    m = len(ls)//2
    return [ls[m], abr_equilibre(ls[:m]), abr_equilibre(ls[m+1:])]

# Pour faciliter les manipulations par la suite, on définit une
# fonction similaire pour la création d'un abr dans l'ordre
def abr_quelconque(l):
    abr = None
    for m in l:
        abr = insere_mot(m, abr)
    return abr
```

```
[11]: c = abr_equilibre("la tribu prophétique aux prunelles ardentes hier s'est mise_
↳en route".split())
pretty_abr(c)
print("Arbre de profondeur %d." % profondeur(c))
```

```
mise
  en
    ardentes
      nil
      aux
        nil
        nil
      hier
        nil
        la
          nil
```

```

        nil
route
  prophétique
    nil
    prunelles
      nil
      nil
s'est
  nil
tribu
  nil
  nil

```

Arbre de profondeur 4.

**Insertion du “lexique” d’un texte** En partant d’une liste de mots-formes extraite d’un texte, on peut créer 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 ?

```

[12]: # Fonction utilitaire: nombre de noeuds d'un ABR
# Le programme est basé toujours sur un parcours en profondeur
def nb_noeuds(abr):
    if abr == None: return 0
    (r,fg,fd) = abr
    return nb_noeuds(fg) + nb_noeuds(fd) + 1

# petite fonction utilitaire pour afficher les propriétés d'un abr
def proprietes_abr(abr):
    import math
    n = nb_noeuds(abr)
    p = profondeur(abr)
    pt = math.log(n,2)
    print("L'abr de %d mots a une profondeur de %d (prof théorique %.2f)" %
    ↪(n,p,pt))

```

```

[13]: # Programmes très simples pour charger un lexique à partir d'un fichier texte.
def charge_fichier(nf):
    lgns = []
    with open(nf, "r", encoding="utf8") as f:
        for ligne in f:
            lgns.append(ligne.strip())
    return lgns
def charge_lexique(nf):
    liste_lignes = charge_fichier(nf)
    lexique = []
    for l in liste_lignes:

```

```

    for w in l.split():
        if w not in lexique:
            lexique.append(w)
    return lexique

```

[14]: *# Récupération lexiques des Pensées et du texte de Verne*

```

fp = charge_lexique("pensees_pascal_utf8.txt")
fv = charge_lexique("demo-file-utf8.txt")
p_abr = abr_quelconque(fp)
proprietes_abr(p_abr)
v_abr = abr_quelconque(fv)
proprietes_abr(v_abr)
v_bal = abr_equilibre(fv)
proprietes_abr(v_bal)

```

L'abr de 8842 mots a une profondeur de 60 (prof théorique 13.11)

L'abr de 16023 mots a une profondeur de 38 (prof théorique 13.97)

L'abr de 16023 mots a une profondeur de 14 (prof théorique 13.97)

[15]: *# Une autre série de tests, basée sur un réordonnement aléatoire  
# de la liste de mots. Permet de voir la variation de profondeur.*

```

import random
random.shuffle(fv)
for x in range(10):
    random.shuffle(fv)
    proprietes_abr(abr_quelconque(fv))

```

L'abr de 16023 mots a une profondeur de 34 (prof théorique 13.97)

L'abr de 16023 mots a une profondeur de 31 (prof théorique 13.97)

L'abr de 16023 mots a une profondeur de 34 (prof théorique 13.97)

L'abr de 16023 mots a une profondeur de 32 (prof théorique 13.97)

L'abr de 16023 mots a une profondeur de 31 (prof théorique 13.97)

L'abr de 16023 mots a une profondeur de 31 (prof théorique 13.97)

L'abr de 16023 mots a une profondeur de 32 (prof théorique 13.97)

L'abr de 16023 mots a une profondeur de 33 (prof théorique 13.97)

L'abr de 16023 mots a une profondeur de 34 (prof théorique 13.97)

L'abr de 16023 mots a une profondeur de 34 (prof théorique 13.97)

**Commentaire** A moins d'avoir vraiment beaucoup de chance, la profondeur réelle est nettement plus grande que la profondeur théorique. D'où l'intérêt de mettre en oeuvre des algos d'équilibrage.

**Maximum d'un ABR** Intuitivement, la valeur minimale d'un ABR est celle qui est le plus "à gauche" dans l'arbre. Plus rigoureusement, la valeur minimale est le premier noeud sans fils gauche que l'on rencontre dans un parcours en profondeur d'abord (révision: <https://xkcd.com/761/>). Implémenter une fonction qui trouve la valeur minimale d'un ABR. Il peut aussi être intéressant de réfléchir à une fonction qui saurait extraire de l'ABR le sous-arbre issue de la valeur minimale.

[ ]:

**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.

Le poly suivant peut-être très utile. [https://ressources.unisciel.fr/algoprogram/s46bst/emodules/br00macours1/res/br00macours1\\_texte-xxx.pdf](https://ressources.unisciel.fr/algoprogram/s46bst/emodules/br00macours1/res/br00macours1_texte-xxx.pdf)

En pratique, il y a différents cas de figure à considérer, et dans le cas le plus complexe il faut remarquer que le moyen le plus simple de remplacer une valeur à supprimer est de la remplacer par le minimum de son sous-arbre droit, ou par le maximum de son sous-arbre gauche.

[ ]:

### 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 sous le nom d'arbre AVL [https://fr.wikipedia.org/wiki/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").

[ ]: