tp_l8dn003_22_01-ABR-corr

May 12, 2022

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 tuple 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 ?

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/algoprog/s46bst/emodules/br00macours1/res/br
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 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").

[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.
# 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
```

0.1.4 Recherche dans un ABR

[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)
```

[5]:
```
# Pour tester, quelques mots au hasard:
mots_test = ['sois', 'soi', 'sage', 'maison', 'bateau', 'ama', 'ma', 'douleur']
for m in mots_test:
    print("%10s: %s" % (m, recherche(minilex,m)))
```

```
      sois: True
       soi: False
      sage: True
    maison: False
    bateau: False
       ama: False
        ma: True
   douleur: True
```

0.1.5 Remplissage d'un ABR

Insertion d'un mot

[6]:
```
# 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)]
```

[7]:
```
# Petit essai : on insère des nouveaux mots dans minilex
for m in "tu reclamais le soir il descend le voici".split():
    minilex = insere_mot(m, minilex)
pretty_abr(minilex)
```

```
sois
  ma
    douleur
      descend
        nil
        nil
      le
        il
          nil
          nil
        nil
    sage
      plus
        nil
        reclamais
          nil
          nil
      soir
        nil
        nil
```

```
  toi
    tiens
      nil
      nil
    tranquille
      nil
      tu
        nil
        voici
          nil
          nil
```

[8]:
```
# 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
```

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.

[9]:
```
def profondeur(abr):
    if abr == None: return 0
    (r,fg,fd) = abr
    return max(profondeur(fg), profondeur(fd)) + 1
```

[10]:
```
print(profondeur(a))
print(profondeur(minilex))
```

```
6
5
```

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: puique 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.

[11]:
```
# 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.

[12]:
```
# 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
```

[13]:
```
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 créér un ABR en insérant à son tour chaque mot dans l'ABR.

[14]:
```
# 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))
```

[15]:
```
# 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
```

[16]:
```
# 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)
```

[17]:
```
# Une autre série de tests, basée sur un réordonnancement 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 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 32 (prof théorique 13.97)
L'abr de 16023 mots a une profondeur de 30 (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 36 (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)
```

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.

0.1.6 Maximum d'un ABR

Fonction de recherche du maximum (ou du minimum): on peut s'inspirer de l'algorithme de parcours, mais on peut aussi raisonner récursivement de la manière suivante: le maximum d'un arbre est soit la racine, si cet arbre n'a pas de fils droit, soit le maximum de son fils droit. Cette fonction renvoie l'arbre dont la racine est le maximum.

[18]:
```
def maximum(abr):
    "renvoie le sous-arbre dont la racine est le max"
    if abr == None: return None
    (r,fg,fd) = abr
```

```
    if fd == None: return abr
    return maximum(fd)

def maximum_val(abr):
    "renvoie la valeur max dans un ABR - calcul structurel"
    if abr == None: return None
    (r,fg,fd) = abr
    if fd == None: return r
    return maximum(fd)
```

[19]:
```
# cellule de vérification de la fonction maximum
#pretty_abr(c)

def parcours_test(abr):
    print(maximum(abr))
    (r,fg,fd) = abr
    if fg != None:
        parcours_test(fg)
    if fd != None:
        parcours_test(fd)

#parcours_test(c)
#d = abr_equilibre("la prophétique aux prunelles rz ardentes hier s'est mise en␣
 ↪route".split())
#pretty_abr(d)
#parcours_test(d)
```

[20]:
```
print(maximum(["10", ['8', None, None], ["14", ["13", ["12", None, None],␣
 ↪None], None]]))
```

```
['14', ['13', ['12', None, None], None], None]
```

Suppression d'un mot La suppression est la partie la plus complexe, en particulier parce qu'il y a beaucoup de cas différents à prendre en compte, et aussi parce que certains de ces cas sont eux-même intrinsèquement complexes. Voyons ces différents cas: 1. le mot à supprimer n'est pas dans l'arbre. Trop facile ! 2. le mot à supprimer est une feuille : `("cible", None, None)` est simplement remplacé par `None` 3. le mot à supprimer a un seul fils non vide : p. ex. `("cible", fg, None)`. Il suffit de remplacer cet arbre par l'arbre fg. *Noter que ce cas peut-être vu comme un cas particulier du précédent* 4. le mot à supprimer a deux enfants non vides: `("cible", fg, fd)`. Il est nécessaire dans ce cas de décider ce qu'on va mettre à la place de la cible.

L'idée la plus courante pour ce dernier cas est de remplacer la cible par le maximum de son fils gauche (ou de manière équivalente par le minimum de son fils droit). En effet, par construction, le maximum du fils gauche est à la fois inférieur à la cible et à tous les noeuds du fils droit, et supérieur à tous les autres noeuds du fils gauche. Il peut donc occuper la place de la cible sans modifier les propriétés ABR de l'arbre.

On peut remarquer aussi que ce noeud (ou feuille) qui est le maximum du fg est forcément démuni de fils droit (car s'il avait un fils droit il ne serait pas le maximum). Il est donc relativement facile

de supprimer ce noeud de l'arbre (pour le copier la place de la cible): on est dans les cas no 2 ou 3 détaillés ci-dessus.

[21]:
```
# Rappel: un noeud qui n'a pas de fils droit est le max de son sous-arbre
# On suppose l'arbre non vide (pas de contrôle d'erreur)
def pop_maximum(abr):
    "renvoie le max d'un arbre, et l'arbre débarrassé de ce max"
    (r,fg,fd) = abr
    if fd == None:
        return r, fg
    max, fd_red = pop_maximum(fd)
    return max, [r, fg, fd_red]

print(pop_maximum(c))
print(pop_maximum([10, [8, None, None], [14, [13, [12, None, None], None],␣
 ↪None]]))
```

```
('tribu', ['mise', ['en', ['ardentes', None, ['aux', None, None]], ['hier',
None, ['la', None, None]]], ['route', ['prophétique', None, ['prunelles', None,
None]], ["s'est", None, None]]])
(14, [10, [8, None, None], [13, [12, None, None], None]])
```

[24]:
```
# bidouilles de test
#for t in (a,b,c):
#    pretty_abr(a)
#    val, tree = pop_maximum(t)
#    print(val, "-----------------")
#    pretty_abr(tree)
#    print("---------------------------")
```

[22]:
```
def suppression(abr, val):
    if abr is None: return None
    (r,fg,fd) = abr
    if val < r: return [r, suppression(fg, val), fd]
    if val > r: return [r, fg, suppression(fd, val)]
    # on est dans le cas où abr est le noeud à supprimer
    if fd == None:
        return fg
    if fg == None:
        return fd
    rmax, fg_poped = pop_maximum(fg)
    return [rmax, fg_poped, fd]
```

[23]:
```
# Quelques tests
#pretty_abr(c)
#for w in "la tribu prophétique aux prunelles ardentes hier s'est mise en␣
 ↪route".split():
#    print(w, ":")
```

```
#    pretty_abr(suppression(c, w))
```