Tris récursifs ; piles

Samedi 10 décembre 2016

Buts du TP

- Reprendre les tris récursifs.
- Être capable d'écrire des petites fonctions sur les piles en faisant abstraction de leur implémentation effective.
- Utiliser les piles dans trois applications classiques (notation polonaise inversée, tri par insertion et parcours de graphe).

Exercice 1. *Créer (au bon endroit) un dossier associé à ce TP. Y placer une copie du fichier récupéré dans le dossier partagé de travail de la classe :* `cadeau_piles.py`*; Observer et comprendre les cinq fonctions de ce fichier concernant l'implémentation des piles.*
Lancer Spyder/Pyzo/Idle, sauvegarder au bon endroit le fichier `tdpilesouuntruccommeça.py`*; écrire une commande absurde, de type* `print( 6*7 )` *dans l'éditeur, sauvegarder et exécuter.* Si vous êtes sous Spyder, changez (via F6) les options d'exécution, pour avoir à chaque exécution une nouvelle console et garder la main dessus. (il y a donc deux cases à vérifier/cocher).

Exercice 2. *Vérifier que le premier exo a effectivement bien été fait : tout manquement donnera lieu à une agitation néfaste pour tout le monde...*

1 Tri « fusion »

On rappelle le principe du tri *fusion* ou *dichotomique* : si un tableau a une taille $\geqslant 2$, on le casse en deux, on trie récursivement les deux morceaux T_1 et T_2, puis on les fusionne via l'algorithme suivant, où on adjoint (via la méthode `append`) à un tableau initialement vide successivement les éléments de T_1 et T_2 :

Entrées : T_1, T_2
$Res \leftarrow []$ # le tableau qui sera rendu
$i_1, i_2 \leftarrow 0, 0$ # les indices qu'on regarde dans les tableaux T_1 et T_2
tant que $i_1 < |T_1|$ *ou* $i_2 < |T_2|$ **faire**
  si $i_2 = |T_2|$ *ou* ($i_1 < |T_1|$ *et* $T_1[i_1] < T_2[i_2]$) **alors**
    Adjoindre $T_1[i_1]$ à Res
    $i_1 \leftarrow i_1 + 1$
  sinon
    Adjoindre $T_2[i_2]$ à Res
    $i_2 \leftarrow i_2 + 1$
Résultat : Res

Exercice 3. *Écrire une fonction réalisant la fusion de deux tableaux supposés triés.*

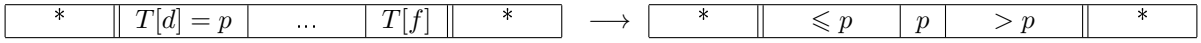
C'est presque fini !

Exercice 4. *Écrire une fonction réalisant le tri-fusion d'un tableau. Cette fonction doit renvoyer un nouveau tableau (et non pas trier en place comme dans les deux premières parties).*

Exercice 5. *Tester et chronométrer le tri fusion.*

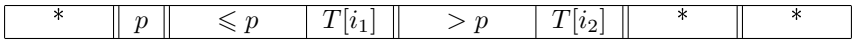
2 Tri rapide

Le principe du tri rapide consiste, pour trier une zone $[\![d, f]\!]$ d'un tableau, si $f > d$, à effectuer une première phase de préparation de la zone, à l'issue de laquelle les éléments de cette zone ont été permutés pour avoir le schéma :



En plus de la préparation de cette zone, l'algorithme doit retourner la nouvelle position du pivot (ce qui sera crucial pour lancer les deux appels récursifs ultérieurs pour trier la zone).

Un algorithme permettant de préparer la zone $[\![d, f]\!]$ consiste à manipuler deux indices i_1 et i_2 délimitant les fins respectives de zones d'éléments respectivement *majorés par* et *strictement minorés par* le pivot $p = T[d]$:



L'invariant préservé est :

$$(d < i \leqslant i_1 \implies T[i] \leqslant p) \quad \text{et} \quad (i_1 < i \leqslant i_2 \implies T[i] > p)$$

Entrées : T, d, f
$i_1, i_2 \leftarrow d, d$ # Si, vraiment !
tant que $i_2 < f$ **faire**
  si $T[i_2 + 1] \leqslant p$ **alors**
    $T[i_1 + 1] \leftrightarrow T[i_2 + 1]$
    $i_1 \leftarrow i_1 + 1$
  $i_2 \leftarrow i_2 + 1$
$T[d] \leftrightarrow T[i_1]$
Résultat : i_1

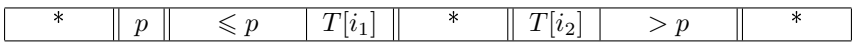
On note que cet algorithme est à « effets de bord » (il modifie le tableau qui a été donné) et retourne également un résultat (l'indice du pivot à la fin).

Exercice 6. *Écrire une fonction réalisant ce travail de préparation. La tester.*

L'élément principal du tri rapide est écrit ; terminons le travail !

Exercice 7. *Écrire une fonction réalisant en place le tri rapide d'un tableau ; tester et chronométrer.* On observera en particulier le comportement sur les tableaux déjà triés.

S'il vous reste du temps : mettez au point et programmez l'algorithme de préparation de tableau (dans le tri rapide) consistant à maintenir l'invariant décrit ainsi :



On réfléchira à l'initialisation de i_1 et i_2, au principe leur permettant de se rejoindre... puis on implémentera le tout.

Au chronomètre, j'obtiens des temps légèrement meilleurs (de l'ordre de 15%) qu'avec la première méthode.

3 Des petites fonctions sur les piles

Dans cette partie, il est essentiel de n'utiliser que les fonctions/méthodes de création, test de vacuité, empilement et dépilement.

Exercice 8. *Importer les fonctions fournies dans la première partie du fichier* `implementations`
Si un brave « `from implementations import *` » ne marche pas... faites un copier/coller des fonctions dont vous avez besoin !
Lire et comprendre le code des 5 *fonctions standards de manipulation pour les deux premières implémentations.*
Pour chacune des implémentations, créer une nouvelle pile, empiler successivement 7, 8, 12, 10, 42. Afficher l'état de la pile, dépiler deux éléments, et afficher à nouveau l'état de la pile.
On pourra faire ces opérations dans la fenêtre d'exécution avant de recopier/commenter le résultat vers la fenêtre de script.

```
>>> p = creer_pile()
>>> for x in [7, 8, 12, 10, 42]:
        empiler(x, p)
>>> affichage(p)
'7 ; 8 ; 12 ; 10 ; 42'
>>> depiler(p)
42
>>> affichage(p)
'7 ; 8 ; 12 ; 10'
>>> depiler(p)
    ...
```

Exercice 9. *Écrire une fonction réalisant la copie d'une pile.*
La pile donnée en argument devra être laissée intacte.

On peut tester la fonction précédente par exemple comme ceci :

```
p3 = creer_pile()
for x in [7, 8, 12, 10, 42]:
    empiler(x, p3)
print(affichage(p3))
p4 = copie(p3)
print(affichage(p3))
print(affichage(p4))
```

Si tout se passe bien, vous devez obtenir :

```
7 ; 8 ; 12 ; 10 ; 42
7 ; 8 ; 12 ; 10 ; 42
7 ; 8 ; 12 ; 10 ; 42
```

Exercice 10. *Écrire une fonction réalisant l'échange des deux éléments (supposés exister) en haut de pile. Rien ne doit être retourné.*

Testez !

```
>>> p3
[7, 8, 12, 10, 42]
>>> echange(p3)
>>> p3  # ici encore, la méthode __repr__ est appelée pour l'affichage
7 ; 8 ; 12 ; 42 ; 10
```

Exercice 11. *Écrire une fonction réalisant la « rotation » des n éléments en haut de pile selon le modèle suivant :*

```
>>> affichage(p4)
'7 ; 8 ; 12 ; 10 ; 42'
>>> roll(p4, 4)
>>> affichage(p4)
'7 ; 42 ; 8 ; 12 ; 10'
```

Exercice 12. *Écrire enfin une fonction « superposant » deux piles suivant le modèle suivant :*

```
>>> affichage(p3)
'7 ; 8 ; 12 ; 10 ; 42'
>>> affichage(p4)
'7 ; 42 ; 8 ; 12 ; 10'
>>> affichage(superposition(p3, p4))
'7 ; 42 ; 8 ; 12 ; 10 ; 7 ; 8 ; 12 ; 10 ; 42'
>>> affichage(p3)
'7 ; 8 ; 12 ; 10 ; 42'
>>> affichage(p4)
'7 ; 42 ; 8 ; 12 ; 10'
```

4 Trois applications classiques

4.1 Notation polonaise inversée : pop/pop/push

On fournit dans le fichier `implementations` une fonction (raisonnablement robuste) transformant une expression arithmétique (avec entiers, additions, multiplications et parenthèses) en notation polonaise inversée :

```
>>> chaine_vers_npi('  612 + 3 *  7')
[612, 3, 7, '*', '+']
>>> chaine_vers_npi('  (612 + 3) *  7')
[612, 3, '+', 7, '*']
>>> chaine_vers_npi('  (1+2*(3+(4+5))+6)*(7*8+  9 )  ')
[1, 2, 3, 4, 5, '+', '+', '*', 6, '+', '+', 7, 8, '*', 9, '+', '*']
```

Exercice 13. *Écrire une fonction réalisant (à l'aide d'une pile d'opérandes comme expliqué en cours) l'évaluation d'une expression écrite en polonaise inversée.*

```
>>> evaluation([10, 4, 8, '*', '+'])
42
>>> evaluation(chaine_vers_npi('1*((2+(3+(4+5)))*(6*((7+8)+9)))'))
2016
```

4.2 Tri d'une pile par insertions successives

On rappelle qu'on peut trier une pile en empilant les éléments successifs dans une pile qui reste à tout moment triée. Cette fonction d'insertion est externalisée :

Entrées : x_0, P
$Buf \leftarrow$`PileVide()`
Continuer $\leftarrow$`True`
tant que *Continuer et (P n'est pas vide)* **faire**
  $y \leftarrow Depiler(P)$
  si $y \leqslant x_0$ **alors**
    Empiler y dans P
    Continuer $\leftarrow$`False`
  sinon
    Empiler y dans Buf
  Empiler x_0 dans P
  Déverser Buf dans P

Exercice 14. *Comprendre cet algorithme ; l'exécuter virtuellement pour insérer* 9 *dans une pile contenant (de bas en haut)* $7; 8; 10; 12; 42$.

Exercice 15. *Écrire une fonction réalisant l'insertion d'un élément dans une pile en maintenant l'ordre (croissant) des éléments de ladite pile. Rien n'est retourné, mais la pile donnée en argument est modifiée.*

```
>>> p = creer_pile()
>>> for x in [7, 8, 10, 12, 42]:
...     empiler(x, p)
...
>>> affichage(p)
'7 ; 8 ; 10 ; 12 ; 42'
>>> inserer(9, p)
>>> affichage(p)
'7 ; 8 ; 9 ; 10 ; 12 ; 42'
>>> inserer(50, p)
>>> inserer(2, p)
>>> affichage(p)
'2 ; 7 ; 8 ; 9 ; 10 ; 12 ; 42 ; 50'
"""
```

On peut maintenant trier les piles.

Exercice 16. *Écrire une fonction réalisant le tri d'une pile EN PLACE (rien n'est retourné, mais la pile donnée en argument a été modifiée).*

```
p = creer_pile()
for _ in range(10):
    empiler(randint(1, 50), p)
pt = tri_insertion_exterieur(p)
print(affichage(p))
print(affichage(pt))

"""
38 ; 37 ; 36 ; 19 ; 28 ; 12 ; 3 ; 35 ; 32 ; 22
3 ; 12 ; 19 ; 22 ; 28 ; 32 ; 35 ; 36 ; 37 ; 38
"""
```

La petite variante qui suit doit vous prendre moins de 5 minutes, tests compris !

Exercice 17. *Écrire une fonction réalisant le tri d'une pile en retournant une pile avec le même contenu, mais trié ; la pile initiale doit être laissée intacte.*

```
p = creer_pile()
for _ in range(10):
    empiler(randint(1, 50), p)
print(affichage(p))
pt = tri_insertion_exterieur(p)
print(affichage(p))
print(affichage(pt))

"""
21 ; 15 ; 18 ; 36 ; 13 ; 27 ; 14 ; 47 ; 33 ; 29
21 ; 15 ; 18 ; 36 ; 13 ; 27 ; 14 ; 47 ; 33 ; 29
13 ; 14 ; 15 ; 18 ; 21 ; 27 ; 29 ; 33 ; 36 ; 47
"""
"""
```

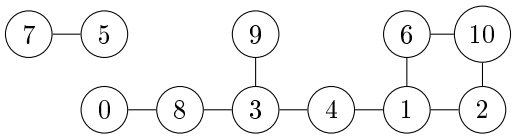
4.3 Parcours d'un graphe

On termine avec le parcours d'un graphe à l'aide d'une pile. Il s'agit de visiter tous les sommets appartenant à la composante d'un sommet s_0. On tient à jour la pile des sommets qui restent à traiter, le tableau (de booléens) précisant le caractère visité ou non des sommets, et enfin la liste des sommets

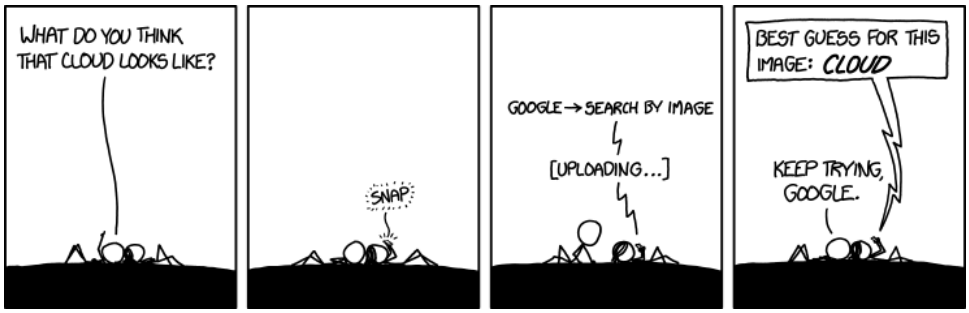
visités.

Entrées : s_0, G
$TODO \leftarrow$ `PileVide()`
$VU \leftarrow$ `[False, ..., False]`
$Visites \leftarrow [s_0]$
Empiler s_0 dans $TODO$
$VU[s_0] \leftarrow$ `True`
tant que *TODO n'est pas vide* **faire**
  $s \leftarrow Depiler(TODO)$
  pour *chaque voisin* v *de* s **faire**
    si $NON(VU[v])$ **alors**
      $VU[v] =$`True`
      Empiler v dans $TODO$
      Ajouter v à *visites*
  Résultat : *visites*

Exercice 18. *Comprendre l'algorithme précédent ! On l'appliquera à la main sur le graphe suivant (fourni dans la variable* `graphe0` *de* `implementations.py`*) en réalisant le parcours depuis les sommets* 0, 4 *et* 5 *:*



Exercice 19. *Écrire une fonction prenant en entrée un sommet* s_0 *et un graphe* G*, et retournant la liste des sommets visités (dans l'ordre de visite !) lors du parcours expliqué plus haut ; appliquer à l'exemple précédent.*



XKCD 1444 – *Cloud computing has a ways to go.*