Université de Lille 1 SCIENCES ET TECHNOLOGIES

ASD

2016-2017
Licence SESI - Semestre 4

DS2 - documents de cours, TD, TP autorisés - durée 2h - Éléments de correction

---

Note : lorsque, dans une question, est demandée la complexité, c'est une fonction de la taille de la donnée qui est attendue. Lorsque c'est le comportement asymptotique, la réponse attendue est soit une notation $\mathcal{O}$, soit une notation Θ soit une notation Ω (le choix vous incombe).

Vous pourrez utiliser toute fonction vue en cours, TD, TP à la condition expresse de spécifier correctement les entrées et les sorties.

Le barême et le temps sont donnés à titre indicatif.

Exercice 1 : AVL [4 points, 20 minutes]

Compétence évaluée : rassembler des connaissances, calculer une complexité.

Un étudiant annonce : « J'ai un nouvel algorithme de tri en $\Theta(n \log n)$: étant donnés n éléments, je les ajoute successivement, sans pré-traitement, à un AVL, puis je réalise un parcours infixé de l'arbre ».

Q 1.1 [1] Expliquez pourquoi l'algorithme est correct (il trie dans l'ordre croissant).

Corrigé

Un AVL est un arbre binaire de recherche. Par conséquent un parcours infixé sortira d'abord les étiquettes à droite d'un nœud (les plus petites) plus l'étiquette du nœud, puis les étiquettes à gauche du nœud (les plus grandes). Nous aurons donc les valeurs dans l'ordre croissant.

Q 1.2 [1] Quelle est la complexité d'ajout de l'ensemble des n éléments ? Justifiez clairement en détaillant vos calculs.

Corrigé

L'ajout d'un élément dans un AVL se fait en $\Theta(h)$ si h est la hauteur de l'AVL. Insérer n éléments nécessite donc

$$\sum_{m=1}^{n} h_m$$

où h_m est la hauteur de l'arbre après insertion de m éléments. h_1 vaut 0, h_2 vaut 0, h_3 vaut 1, etc. h_i vaut de l'ordre de $\log_2 i$ et il y a . On aboutit donc à
e nombre de nœuds présents dans l'arbre au moment de
Une autre réponse, moins précise serait : L'ajout d'un élément dans un AVL se fait en $\Theta(\log m)$ si m est le nombre de nœuds présents dans l'arbre au moment de l'insertion. Insérer n éléments nécessite donc

$$\sum_{m=1}^{n} \log m < n \log n$$

On est donc en $\Theta(n \log n)$.

Q 1.3 [1] Quelle est la complexité du parcours ?

Corrigé

Un parcours infixé nécessite de passer une et une seule fois sur chaque nœud. On est donc en $\Theta(n)$.

Q 1.4 [1] Concluez sur l'exactitude de la complexité proposée par l'étudiant ?

Corrigé

L'insertion nécessite $\Theta(n \log n)$ et l'extraction $\Theta(n)$, l'algorithme au total nécessite $\Theta(n \log n)$.

Exercice 2 : Des listes, des listes, toujours des listes [5 points, 40 minutes]

Compétence évaluée : choisir une représentation de données, implanter en respectant une complexité.

On veut disposer d'une structure de données stockant, pour un ensemble S de n éléments, plusieurs ordres pour ces éléments. Chaque ordre est donc une séquence ordonnée, de même longueur, qui contient exactement une fois chaque élément. Par exemple, pour $S = \{'a', 'b', 'c', 'd'\}$ on souhaite pouvoir stocker les ordres $o_1 = ('a', 'c', 'b', 'd')$, $o_2 = ('b', 'c', 'd', 'a')$ et $o_3 = ('a', 'b', 'd', 'c')$.

Pour réaliser cela, on propose d'utiliser des listes chaînées qui permettront de stocker les différents ordres. Afin d'éviter la redondance des données, chaque cellule des listes ne stockera pas les éléments eux-mêmes mais une référence vers ceux-ci.

La structure de données en Python pourra être un dictionnaire stockant les éléments d'un côté, et les ordres d'un autre : `{ 'elements' : XXX, 'ordres' : YYY }` où `XXX` et `YYY` sont des structures de données que vous aurez à définir.

On souhaite réaliser les opérations suivantes sur la structure de données, avec les complexités indiquées (n représente le nombre d'éléments et m le nombre d'ordres) :

— ajouter un élément en $\Theta(1)$: ajoute le nouvel élément en tête de chaque ordre
— supprimer un élément en $\Theta(m)$: il est supprimé de l'ensemble des m ordres
— créer un nouvel ordre en $\Theta(n)$: l'ordre des éléments est passé en paramètre sous la forme d'une liste

De plus, sur un ordre donné, on souhaite pouvoir réaliser l'opération suivante :

— accéder à l'élément dans l'ordre en $\Theta(1)$,
— échanger deux éléments consécutifs en $\Theta(1)$: on suppose connaître (avoir des références sur) les deux éléments à échanger

Il est important de lire l'ensemble des questions avant de répondre.

Q 2.1 [1.5] Pour représenter chaque ordre, choisissez-vous une implantation avec liste simplement chaînée ou liste doublement chaînée ? Justifiez en expliquant en quoi ce choix permet de garantir les complexités des opérations d'ajout, de suppression et d'échange.

Corrigé

Une liste doublement chaînée : l'ajout en tête est en temps constant, il suffit d'ajouter une nouvelle cellule, la suppression est en temps constant car on a accès au successeur et prédécesseur en temps constant, l'échange est en temps constant pour les mêmes raisons.

Q 2.2 [0.5] Quelle structure de données choisissez-vous pour représenter l'ensemble des ordres (`YYY`) ?

Corrigé

Une liste simplement chaînée est suffisante ici.

Q 2.3 [1] En plus de la valeur explicite d'un élément de S, quelle(s) information(s) faut-il ajouter à un élément pour pouvoir garantir l'opération de suppression en $\Theta(m)$ (on précise ici que la complexité en espace de la structure permettant de stocker un élément est en $\Theta(m)$). Justifiez la complexité en $\Theta(m)$ de cette opération.

Corrigé

L'opération de suppression d'un élément dans une liste en en $\Theta(1)$. Pour garantir une opération de suppression dans toutes les listes en $\Theta(m)$ il faut avoir un accès en temps constant à l'élément dans les m listes. Il faut donc conserver une référence vers la cellule où se trouve l'élément dans les listes.

Q 2.4 [0.5] Quelle structure de données choisissez-vous pour représenter l'ensemble des éléments (`XXX`) ?

Corrigé

Par exemple, une table de hachage qui à un élément associe la liste des références à cet élément dans les ordres.

Q 2.5 [1.5] Dessinez la structure de données pour l'exemple donné ci-dessus (on utilisera les représentations usuelles des listes et des flèches pour indiquer des références).

Corrigé

Il s'agissait de dessiner correctement la structure de données pour l'exemple. Autrement dit il fallait bien voir les 3 ordres, la liste des éléments et pour chaque élément le lien vers leur position dans les ordres.

Exercice 3 : PQ-arbres [10 points, 60 minutes]

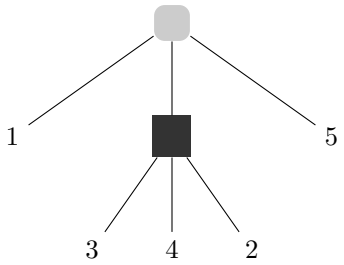
Compétence évaluée : comprendre une nouvelle structure de données, manipuler des arbres à arité variable, évaluer des complexité, écrire des algorithmes récursifs.

On suppose manipuler des arbres, nommés $PQ-$arbres, disposant de deux types de nœuds internes :
— des nœuds Q dont l'arité est au moins 2 et dont l'ordre des fils est important,
— des nœuds P dont l'arité est au moins 3 et dont l'ordre des fils est sans importance.
Les nœuds internes ne portent pas de valeur, seules les feuilles en contiennent.

Les m fils $x_1, x_2, ..., x_m$ d'un nœud Q pourront être parcourus en traitant les fils soit dans l'ordre $x_1, x_2, ..., x_m$ soit dans l'ordre $x_m, x_{m-1}, ..., x_1$. Les m fils $x_1, x_2, ..., x_m$ d'un nœud P pourront être parcourus en traitant les fils dans n'importe quel ordre.

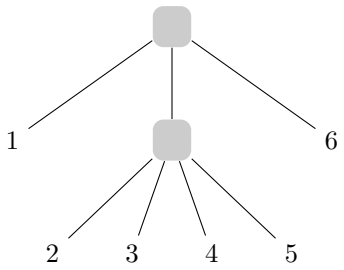
La différence entre les nœuds Q (fils ordonnés) et les nœuds P (fils non ordonnés) permet une représentation compacte d'un ensemble de données.

On se sert par exemple de ce type de structure de données pour représenter des permutations dans le cas où les feuilles contiennent des entiers. Par exemple, l'arbre suivant :



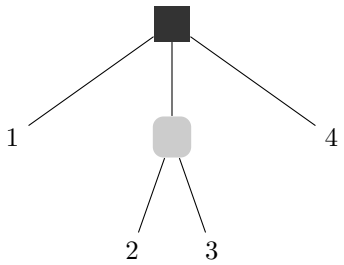
où les nœuds Q figurent en gris clair, les nœuds P en gris foncé, permet de représenter les 12 permutations $1, 2, 3, 4, 5$ - $1, 2, 4, 3, 5$ - $1, 3, 2, 4, 5$ - $1, 3, 4, 2, 5$ - $1, 4, 3, 2, 5$ - $1, 4, 2, 3, 5$ - $5, 2, 3, 4, 1$ - $5, 2, 4, 3, 1$ - $5, 3, 2, 4, 1$ - $5, 3, 4, 2, 1$ - $5, 4, 3, 2, 1$ - $5, 4, 2, 3, 1$.

Pour représenter les quatres permutations $1, 2, 3, 4, 5, 6$ - $1, 5, 4, 3, 2, 6$ - $6, 2, 3, 4, 5, 1$ - $6, 5, 4, 3, 2, 1$ on peut utiliser le $PQ-$arbre suivant :



On se propose dans cet exercice de calculer le nombre de permutations d'objets que contient un arbre. Dans les arbres ci-dessus ce serait respectivement 12 et 4.

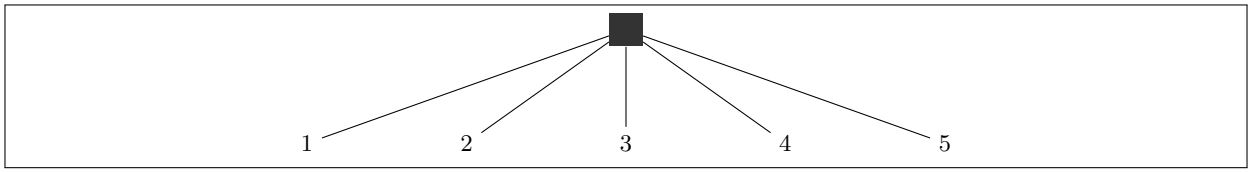
Q 3.1 [1] Listez l'ensemble des permutations produites par le $PQ-$arbre suivant :



Corrigé

1,2,3,4 1,4,2,3 2,3,1,4 2,3,4,1 4,1,2,3 4,2,3,1

Q 3.2 [1] Dessinez un $PQ-$arbre permettant de représenter toutes les permutations de l'ensemble $\{1, 2, 3, 4, 5\}$.
Corrigé



Q 3.3 [1] Quel est le nombre de permutations des nœuds fils d'un nœud P en fonction du nombre m de ses fils ?
Corrigé

$m!$

Q 3.4 [0.5] Quel est le nombre de permutations des nœuds fils d'un nœud Q en fonction du nombre m de ses fils ?
Corrigé

2

On se dote des primitives suivantes, sur un nœud `node` :
- `type(node)` = `Q` si `node` est un nœud Q, `P` si c'est un nœud P, F si c'est une feuille,
- `childs(node)` retourne la liste des fils de `node` (de gauche à droite),
- `arr(node)` retourne le nombre de permutations des nœuds fils de `node` (fournit la réponse donnée aux deux questions précédente).

Q 3.5 [1.5] Ecrivez en pseudo-code une fonction récursive qui étant donné le nœud racine `node` d'un $PQ-$arbre retourne le nombre de permutations encodées dans cet arbre.

Corrigé

```
nb_permutations(n):
  si type(n) == F:
    retourner 1
  sinon:
    nb = 1
    pour chauqe f de childs(n):
      nb *= nb_permutations(f)
    retourner nb*arr(n)
```

On appelle *frontière* d'un $PQ-$arbre la liste des valeurs aux feuilles lorsqu'on lit l'arbre de gauche à droite. La frontière de l'arbre donné dans le premier exemple est $1, 3, 4, 2, 5$.

On se dote des primitives suivantes, sur une feuille `leaf` :
- `value(leaf)` retourne la valeur d'une feuille.

Q 3.6 [1.5] Ecrivez en pseudo-code une fonction `frontier` qui étant donné le nœud racine d'un $PQ-$arbre retourne sa frontière sous forme d'une liste d'entiers.

Corrigé

```
frontier (n):
  si type(n) == F:
    retourner [ value(n) ]
  sinon:
    l = []
    pour chaque m de childs(n):
      l += frontier(m)
    retourner l
```

Q 3.7 [1] Quel est le comportement asymptotique en temps de la fonction `frontier` en fonction de la taille n du $PQ-$arbre ? Justifiez. (on pourra supposer que la concaténation de deux listes est réalisée en temps constant).

Corrigé

On parcourt tout l'arbre, on est donc en $\Theta(n)$.

Q 3.8 [1] Décrivez en français un algorithme permettant de décider si deux $PQ-$arbres ont même frontière. *Corrigé*

Deux possibilités pour cette question :
- soit on considère l'ordre des nœuds P : deux frontières de nœuds P sont égales si elles contiennent les mêmes éléments dans le même ordre
- soit on ne considère pas l'ordre des nœuds P : deux frontières de nœuds P sont égales si elles contiennent les mêmes éléments, peu importe leur ordre

Dans le premier cas, c'est assez simple. On calcule les deux frontières. Puis on réalise un parcours simultané des deux listes. Dès qu'un élément est différent on s'arrête, les frontières sont différentes. Si on épuise les deux listes au même moment, alors les frontières sont identiques.
Dans le second cas, c'est plus compliqué. On ne peut pas y arriver en extrayant la frontière. Cela demande de calculer l'équivalence entre deux PQ-arbres.
C'est la première réponse qui était attendue. En faisant bien attention de ne pas trier sinon on détruit la propriété des nœuds P.

Q 3.9 [1] Quel est le comportement asymptotique en temps de cet algorithme ? *Corrigé*

Si n_1, n_2 sont le nombre de noeuds de chaque PQ-arbre et f_1, f_2 le nombre de feuilles. Le calcul des frontières est en $\mathcal{O}(n_1 + n_2)$. La comparaison des deux listes est en $\mathcal{O}(\min(f_1, f_2))$. Donc l'algorithme en en $\mathcal{O}(n_1 + n_2)$ puisque $f_1 \leq n_1$ et $f_2 \leq n_2$.