Université de Lille 1 SCIENCES ET TECHNOLOGIES

ASD

2015-2016
Licence SESI - Semestre 4

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

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 : Parcours d'arbres

Compétence évaluée : manipuler des arbres binaires.

On suppose manipuler des arbres binaires pour lesquels on dispose des primitives vues en TP.

Q 1.1 Dessiner un arbre qui contient plus de nœuds à la profondeur 3 qu'à la profondeur 2.

Corrigé

□

Q 1.2 Donner (en Python, C ou pseudocode) le code d'une fonction qui étant donné un arbre binaire retourne un entier correspondant à une des profondeurs qui contient le plus de nœuds (on ne cherche pas à obtenir une méthode particulièrement efficace).

Corrigé

Comme il n'est rien précisé sur la complexité attendue, on peut se contenter d'une version basique et peu efficace.

```
def hauteur (a):
    if a == None:
        return -1
    else:
        return a + max(hauteur(a["left"]),hauteur(a["right"]))

def nombre_noeuds (a,p):
    if a == None:
        return 0
    elif p == 0:
        return 1
    else:
        return nombre_noeuds(a["left"],p-1) + nombre_noeuds(a["right"],p-1)

def profondeur_max_noeuds (a):
    h = hauteur (a)
    m = 1
    p = 0
    for pr in range (1,h+1):
        n = nombre_noeuds (a,pr)
        if n > m:
            m = n
            p = pr
    return p
```

Q 1.3 Quelle est la complexité de votre algorithme dans le pire des cas ? Justifier. *Corrigé*

> Le calcule de la hauteur se fait en $\Theta(n)$. Le calcul du nombre de nœuds à une profondeur donnée se fait en $\mathcal{O}(2^p)$. Le calcul du de la profondeur contenant un maximum de nœuds nécessite le calcul de la hauteur de l'arbre puis, pour chaque profondeur le calcul du nombre de nœuds est réalisé. On a donc $\Theta(n) + \sum_1^{h+1} \mathcal{O}(2^p)$. Dans le pire des cas la hauteur est le nombre de nœuds, soit $\mathcal{O}(n^2)$.

Exercice 2 : Comparaison de listes

Compétence évaluée : choisir des structures de données et des algorithmes pour répondre à un problème.

Etant données deux listes d'entiers, on souhaite écrire un prédicat qui teste si les deux listes de longueur respective n et m, $n > m$ sont *équivalentes*. On dira que deux listes sont équivalentes si elles contiennent les mêmes éléments, peu importe l'ordre et le nombre d'occurrences. Par exemple :

`[1; 9; 6; 1; 1; 5]` et `[ 5; 6; 1; 9; 5]` sont équivalentes,
`[ 5; 6; 1; 9; 5]` et `[ 6; 1; 9; 1]` ne sont pas équivalentes (la valeur 5 n'apparaît pas).

On ne présuppose rien sur les éléments contenus dans les listes si ce n'est que ce sont des entiers.

Q 2.1 Proposer un algorithme en français ou en pseudocode pour réaliser cette tâche dont la complexité asymptotique en temps sera dans le pire des cas $\Theta(n \log n)$ (on s'intéresse à la comparaison d'élements).
Corrigé

> On transforme une des deux listes en un tableau que l'on trie puis on recherche chaque élément de l'autre liste dans le tableau.

Q 2.2 Justifier le comportement asymptotique de l'algorithme proposé.
Corrigé

> Le tri en en $n \log(n)$, la recherche d'un élément par dichotomie en $\log(n)$. La création du tableau en n en espace.

On suppose maintenant que les valeurs dans les listes sont comprises entre 0 et `MAX`, une constante.

Q 2.3 Proposer un algorithme en français ou en pseudocode pour réaliser cette tâche dont la complexité asymptotique en temps sera $\Theta(n)$ (on s'intéresse à la comparaison d'élements) et en $\Theta(1)$ en espace.
Corrigé

> Une solution, parmi toutes celles qui utilisent un tableau annexe puisqu'on sait qu'on rencontrera au maximum MAX valeurs différentes : on pourra utiliser deux tableaux `t1` et `t2` d'entiers indicés de 0 à MAX initialisés à faux. Pour chaque élément i de la première liste on met à vrai la valeur de `t1.(i)`, idem pour la seconde liste avec `t2`. Il reste à comparer les deux tableaux qui doivent être identiques pour avoir deux listes équivalentes.

Q 2.4 Justifier les comportements asymptotiques de l'algorithme proposé.
Corrigé

> En $\Theta(n + m)$ puisqu'on parcourt une fois chaque liste. Le parcours des tableaux à la fin est en temps constant puisque qu'on parcourt un nombre constant de cases.

Q 2.5 Que penser du second algorithme dans la pratique ? Argumenter.
Corrigé

> Si les valeurs sont éparses, alors cet algorithme consomme beaucoup d'espace pour rien.

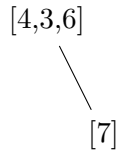
Exercice 3 : Arbres binaires avec liste de valeurs

Compétence évaluée : comprendre un algorithme, une nouvelle structure de données, évaluer une complexité en fonction des structures de données choisies.

On manipule ici des arbres binaires dont l'étiquette est (dans un premier temps) une liste de valeurs de longueur maximale `MAX` fixée. On considérera que `MAX` est une variable globale. `MAX` peut être arbitrairement grand.

Les valeurs dans tout l'arbre seront toujours toutes différentes deux à deux.

Un exemple d'un tel arbre (avec `MAX=3`) est donné ci-dessous :



L'implantation de cette structure de données pourra être réalisée ainsi (l'arbre vide étant représenté par `None`) :

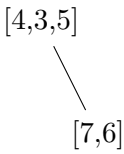
```
    def create_node (left, right):
        """
        :param left: The left son
        :type left: dict or None
        :param right: The right son
        :type right: dict or None
        :return: a tree
        :rtype: dict
        """
        return { "values" : [], "left" : left, "right" : right }
```

L'arbre en exemple ci-dessous sera alors représenté par le dictionnaire :

```
{ 'values' : [4,3,6],
  'left'   : None,
  'right'  : { 'values' : [ 7 ], 'left' : None, 'right' : None } }
```

Pour ajouter de nouvelles étiquettes dans l'arbre, on suit le schéma récursif suivant (implanté dans une fonction `add`). S'il existe un fils gauche (respectivement un fils droit) et que la nouvelle étiquette est inférieure au minimum des étiquettes de la racine (respectivement supérieure au maximum des étiquettes de la racine) alors on insère dans le sous-arbre gauche (respectivement le sous-arbre droit). Dans le cas contraire, si la racine n'est pas pleine, on ajoute la nouvelle étiquette à celles de la racine. Dans le dernier cas on extrait le maximum de la racine puis on ajoute la nouvelle valeur à la racine et on insère le maximum extrait dans le sous-arbre droit (remarquez que l'ordre des étiquettes dans un nœud peut varier au fur et à mesure des insertions).

Par exemple, l'insertion de 5 dans l'arbre ci-dessus aboutira à :



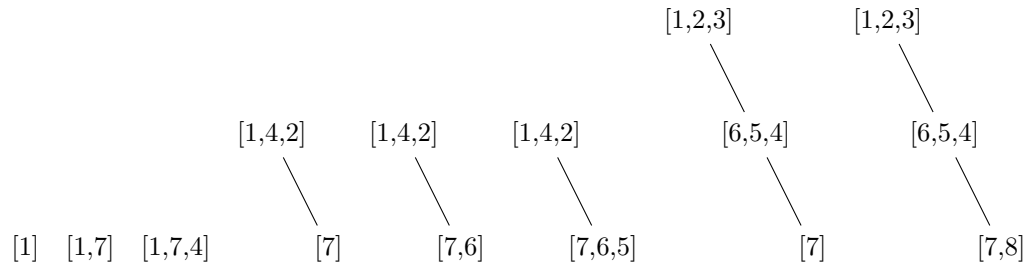
Q 3.1 Dessiner les arbres successifs obtenus avec l'algorithme de construction (avec `MAX=3`) avec la suite d'instructions suivante :

```
a = None
a = add(a,1)
a = add(a,7)
a = add(a,4)
```

```
a = add(a,2)
a = add(a,6)
a = add(a,5)
a = add(a,3)
a = add(a,8)
```

Corrigé

L'ordre des étiquettes dans les nœuds n'a pas d'importance.

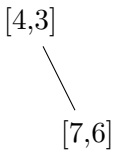


Q 3.2 Quelle propriété remarquable a-t-on entre les étiquettes de la racine et celles des sous-arbres gauche et droit ?

Corrigé

Par construction, les valeurs du sous-arbre gauche sont plus petites que les valeurs à la racine, elles-mêmes plus petites que les valeurs du sous-arbre droit (comme dans un ABR). On remarque également que l'arbre est désiquilibré à droite, en effet, on n'insère jamais de nouvelle valeur à droite.

Q 3.3 Proposer en Python une fonction `extractMax` qui étant donné un arbre retourne la valeur max de la racine et la supprime de la liste des étiquettes. Après l'exécution de `extractMax` sur l'arbre ci-dessus, on récupère 5 et l'arbre est dans l'état suivant :



Corrigé

```
def extractMax (a):
    m = getMax(a)
    a["values"].remove(m)
    return m
```

Dans la suite on supposera l'existence d'une fonction `extractMin`.

Q 3.4 Proposer en Python une implémentation de la fonction `add` qui étant donné un arbre et une étiquette ajoute celle-ci dans l'arbre et retourne l'arbre modifié. *Corrigé*

```
def getMin (a):
    m = a["values"][0]
    p = 0
    for i in range(1,len(a["values"])):
        if a["values"][i] < m:
            m = a["values"][i]
            p = i
    return m

def getMax (a):
    m = a["values"][0]
    p = 0
    for i in range(1,len(a["values"])):
        if a["values"][i] > m:
            m = a["values"][i]
            p = i
    return m

def add (a, v):
    global MAX
    if a == None:
        a = create (None,None)
        a["values"].append(v)
    else:
        n = len(a["values"])
        # si on a remarque qu'il n'y a pas de fils gauche
        # on peut supprimer ces deux lignes
        if a["left"] and v < getMin(a):
            a["left"] = add(a["left"],v)
        elif a["right"] and v > getMax(a):
            a["right"] = add(a["right"],v)
        else:
            if n >= MAX:
                m = extractMax(a)
                a["values"].append(v)
                a["right"] = add(a["right"],m)
            else:
                a["values"].append(v)
    return a
```

Q 3.5 Proposer en Python un prédicat `hasLabel` qui étant donné un arbre et une étiquette retourne vrai si l'arbre contient l'étiquette et faux sinon.

Corrigé

```
def isIn (v,l):
    for x in l:
        if x == v:
            return True
    return False

def hasLabel (a,v):
    if a == None:
        return False
    else:
        if isIn (v,a["values"]):
            return True
        else:
            n = len(a["values"])
            if v < getMin(a):                    idem question ci-dessus
                return hasLabel(a["left"],v)
            elif v > getMax(a):
                return hasLabel(a["right"],v)
            return False
```

Q 3.6 Donner un meilleur des cas pour le prédicat `hasLabel`. Donner la complexité dans ce cas. Justifier.

Corrigé

Un meilleur des cas est si la valeur recherchée est plus petite que le minimum. La recherche s'arrête immédiatement après avoir réalisé un appel à `isIn` et un appel à `getMax`. On est en $\Theta(1)$.

Q 3.7 Donner un pire des cas pour le prédicat `hasLabel`. Donner la complexité dans ce cas. Justifier.

Corrigé

Un pire des cas est que la valeur recherchée est supérieure à la valeur maximale contenur dans l'arbre. Dans ce cas il faudra traverser tous les sous-arbre gauche, dont chaque nœud est, par construction, plein. Il y aura un appel à `isIn` et un appel à `getMax` pour chaque nœud, mais comme `MAX` est une constante, ces appels sont en $\Theta(1)$. Si il y a n valeurs dans l'arbre, alors le nombre de nœuds est $\lfloor \frac{n}{\texttt{MAX}} \rfloor$. On est donc en $\Theta(n)$.

Q 3.8 Pour améliorer la structure de données, on propose d'utiliser des listes qu'on maintient triées. Que pensez-vous de cette proposition vis-à-vis des complexités des fonctions `add` et `hasLabel` ? Argumenter.

Corrigé

Cela ne changera rien en théorie si on considère que `MAX` est une constante.
En pratique, l'accès au max et min des étiquettes d'un nœud devient immédiat, et la recherche peut-être dichotomique, `hasLabel` sera plus efficace.
Par contre l'ajout demandera de maintenir la liste triée, ce qui est plus coûteux puisqu'en $\mathcal{O}(m^2)$ pour une liste à m éléments. Ce sera donc plus coûteux.

Q 3.9 Que proposer comme autre structure de données pour remplacer les listes ?

Corrigé

On pourrait proposer un AVL.

Q 3.10 En quoi celle-ci améliorera ou non le comportement de l'ajout ou de la recherche ? Argumenter.

Corrigé

En théorie, cela ne changera toujours rien. En pratique `hasLabel` aura la même efficacité qu'avec des listes triées. L'ajout sera lui plus efficace qu'avec les listes triées car l'AVL permet l'ajout en $\Theta(\log(n))$.