TP5

May 9, 2016

1 Structure de tas

1.1 Arbres parfaits

Retrouver la définition des arbres binaires parfaits dans le cours et leur représentation sous forme de tableau. On commence par écrire du code pour faciliter la manipulation des arbres binaires parfaits.

In [ ]:
```
"""
Père
Retourne l'indice du père du noeud donné en paramètre

Input :
    - i, l'indice d'un noeud dans un arbre binaire parfait
Output :
    l'indice du père de cet élément dans le tableau. Si l'indice 'i' est 0
    (donc la racine de l'arbre), on retournera la valeur ''None''
"""
def Pere(i):
    # écrire le code ici
    pass
```

In [ ]:
```
# tests
assert(Pere(10) == 4)
assert(Pere(9) == 4)
assert(Pere(5) == 2)
assert(Pere(1) == 0)
assert(Pere(2) == 0)
assert(Pere(0) == None)
```

In [ ]:
```
"""
Fils
Retourne la liste des indices fils

Input :
    - i, l'indice d'un noeud dans un arbre binaire parfait
    - n, la taille de l'arbre binaire parfait
Output :
Une liste comportant 0 (l'éléments n'a pas de fils), 1 ou 2 indices
"""
def Fils(i,n):
    # écrire le code ici
    pass
```

In [ ]:
```
# tests
assert(Fils(0,5) == [1,2])
```

```
assert(Fils(1,5) == [3,4])
assert(Fils(2,5) == [])
assert(Fils(3,5) == [])
assert(Fils(2,6) == [5])
```

On se sert de la fonction Fils pour écrire un parcours en profondeur récursif sur l'arbre binaire parfait. On écrira un parcours préfixe, c'est-à-dire : noeud, fils gauche, fils droit.

In [ ]:
```
"""
Parcours Préfixe
Affiche la valeur des noeuds en ordre préfixe

Input :
    - t, un tableau représentant un arbre binaire parfait
    - i, l'indice de départ
    - n, la taille de l'arbre (on travellera parfois avec des tableaux plus grand que l'arbre)
"""
def ParcoursPrefixe(t,i,n):
    # écrire le code ici
    pass
```

In [ ]:
```
t = [1,2,3,4,5]
ParcoursPrefixe(t,0,5) # doit afficher 1 2 4 5 3
```

In [ ]:
```
t = [1,2,3,4,5,6]
ParcoursPrefixe(t,0,6) # doit afficher 1 2 4 5 3 6
```

In [ ]:
```
t = [1,2,3,4,5,6]
ParcoursPrefixe(t,0,3) # doit afficher 1 2 3
```

On rappelle qu'un **tas** est un arbre binaire parfait tel que la valeur de chaque noeud soit supérieure ou égale à la valeur de ses fils, ou autrement dit, que la valeur de chaque noeud soit inférieure ou égale à la valeur de son père.

Ecrire une fonction qui teste si un arbre binaire parfait est un tas (Remarque, il y a plusieurs façons de l'écrire)

In [4]:
```
"""
TestTas
Teste si un arbre binaire parfait est un tas

Input :
    - t, un tableau représentant un arbre binaire parfait
    - n, la taille de l'arbre
"""
def testTas(t, n):
    # écrire le code ici
    pass
```

In [ ]:
```
# tests
assert(not testTas([1,2,3,4,5],5))
assert(testTas([5,4,3,1,2],5))
assert(testTas([5,4,3,1,4],5))
assert(testTas([5,4,2,1,3],5))
assert(not testTas([5,4,2,1,5],5))
assert(not testTas([5,4,2,1,3,3],6))
assert(testTas([5,4,2,1,3,1],6))
assert(testTas([5,4,2,1,3,3],5))
```

1.2 Insertion

On va écrire une fonction d'insertion dans un tas. * On place le nouvel élément à la fin du tas * On teste la valeur du nouvel élément par rapport à celel de son père, si elle est supérieure, on les échange * Si un échange a eu lieu, on se place sur le père et on recommence * On continue tant que des échanges ont lieu ou que l'on a pas atteint la racine

Remarque : l'algorithme (avec exemples) a été vu en cours

Pour faciliter l'utilisation de la fonction par la suite, on supposera que l'élément à insérer est déjà dans le tableau. On considère que les n premières cases du tableau représente le tas avant l'insertion. L'élément à insérer est $t[n]$, il peut y avoir d'autres valeurs après qui ne font pas partie du tas.

In [5]:
```
"""
Insertion
Insère l'élément t[n] dans le tas de taille n
Input :
    - t, le tableau représentant le tas
    - n, la taille du tas avant l'insertion
"""
def insertionTas(t,n):
    # écrire le code ici
    pass
```

In [ ]:
```
# tests
# exemple du cours
t = [12,10,8,9,10,5,6,8,5,6,7,3,9]
insertionTas(t,12)
assert(t == [12,10,9,9,10,8,6,8,5,6,7,3,5])
assert(testTas(t,len(t)))
# insertion des éléments 1 par 1
t = [1,2,3,4,5]
for i in xrange(5):
    insertionTas(t,i)
assert(t == [5,4,2,1,3])
assert(testTas(t,len(t)))
t = range(100)
for i in xrange(len(t)):
    insertionTas(t,i)
assert(testTas(t,len(t)))
```

1.3 Tamiser (heapify)

L'opération de tamisage a la propriété suivante : si le sous-arbre issu d'un noeud v a la propriété d'être un tas sauf en sa racine v. Alors le tamisage de v assure que le sous-arbre résultat sera un tas. L'algorithme est le suivant : * Si v est plus petit qu'un de ses fils, l'échanger avec le maximum de ses fis. * Si un échange a eu lieu avec un noeud v', relancer le tamisage sur v'

Exemple l'algorithme de suppression de la racine vu en cours revient à échanger la racine et la dernière valeur du tas puis à lancer un tamisage sur la nouvelle racine.

In [6]:
```
"""
Tamisage
Tamise le tableau t à partir de la racine i

Input :
    - t, un tableau représentant un arbre binaire parfait
    - i, l'indice où le tamisage doit être appliqué
```

```
        - n, la taille de l'arbre
    """
    def tamiser(t,i,n):
        # écrire le code ici
        pass
```

In [ ]:
```
# tests
t = [5, 10, 9, 9, 10, 8, 6, 8, 5, 6, 7, 3]
tamiser(t,0,len(t))
testTas(t,len(t))
assert(t == [10, 10, 9, 9, 7, 8, 6, 8, 5, 6, 5, 3])
t = [5, 10, 9, 9, 10, 8, 6, 8, 5, 6, 7, 3, 12]
tamiser(t,0,len(t)-1)
assert(t == [10, 10, 9, 9, 7, 8, 6, 8, 5, 6, 5, 3, 12])
t = [10, 3, 9, 9, 7, 8, 6, 8, 5, 6, 5, 3]
tamiser(t,1,len(t))
assert(t == [10, 9, 9, 8, 7, 8, 6, 3, 5, 6, 5, 3])
```

In [7]:
```
"""
Suppression de la racine
Supprime la racine de l'arbre en la plaçant à la position n-1 du tableau
et effectue un tamisage à la racine du nouveau tas de taille n-1

Input :
    - t, un tableau représentant un arbre binaire parfait
    - n, la taille de l'arbre
"""
def supprimeRacine(t,n):
    # écrire le code ici
    pass
```

In [ ]:
```
# tests
t = [12, 10, 9, 9, 10, 8, 6, 8, 5, 6, 7, 3, 5]
supprimeRacine(t,len(t))
testTas(t,len(t)-1)
assert(t == [10, 10, 9, 9, 7, 8, 6, 8, 5, 6, 5, 3, 12])
t = [12, 10, 9, 9, 10, 8, 6, 8, 5, 6, 7, 3, 5]
for n in xrange(len(t),0,-1):
    supprimeRacine(t,n)
assert(t == [3, 5, 5, 6, 6, 7, 8, 8, 9, 9, 10, 10, 12])
```

1.4 Tri par tas

Le principe du tri par tas est de d'abord former un tas avec les valeurs du tableau, puis d'extraires les éléments maximums successivement. Deux stratégies existent pour la première étape (création du tas). La première est celle que nous avons vu en cours d'insérer les éléments dans le tas un par insertion succesive.

In [8]:
```
"""
Création d'un tas par insertion
transforme un tableau de valeur t en un tas par insertion successive de ses valeurs dans le tas

Input :
    - t, un tableau
"""
```

```
        def creeTasInsertion(t):
            # écrire le code ici
            pass
```

```
In [ ]: t = [1,2,3,4,5]
        creeTasInsertion(t)
        assert(testTas(t, len(t)))
        assert(t == [5,4,2,1,3])
        t = [8,7,6,3,7,6,5,10]
        creeTasInsertion(t)
        assert(testTas(t, len(t)))
        assert(t == [10, 8, 6, 7, 7, 6, 5, 3])
```

```
In [9]: """
        Tri Par Tas 1
        Tri par tas où la première étape est faite par insertion

        Input :
            - t, un tableau de valeurs
        """
        def triTas1(t):
            # écrire le code ici
            pass
```

```
In [ ]: t = [1,2,3,4,5]
        triTas1(t)
        assert(t == [1,2,3,4,5])
        t = [8,7,6,3,7,6,5,10]
        triTas1(t)
        assert(t == [3,5,6,6,7,7,8,10])
```

La deuxième stratégie possible est considérer directement l'ensemble du tableau comme un arbre binaire parfait et d'utiliser le tamisage pour former un tas. Cependant, contrairement à la suppression de la racine, il faut effectuer plusieurs tamisage. **Travaillez sur des exemples pour voir sur quels élements et dans quel ordre les tamisages doivent se faire pour obtenir un tas**.

```
In [10]: """
         Création d'un tas par tamisage successif
         Input :
             - t, un tableau de valeurs
         """
         def creeTasTamiser(t):
             # écrire le code ici
             pass
```

```
In [ ]: t = [1,2,3,4,5]
        creeTasTamiser(t)
        assert(testTas(t, len(t)))
        assert(t == [5,4,3,1,2])
        t = [8,7,6,3,7,6,5,10]
        creeTasTamiser(t)
        assert(testTas(t, len(t)))
        assert(t == [10,8,6,7,7,6,5,3])
```

```
In [11]: """
         Tri Par Tas 2
```

```
        Tri par tas où la première étape est faite par tamisage
        Input :
            - t, un tableau de valeur
        """
        def triTas2(t):
            # écrire le code ici
            pass
```

In [ ]:
```
t = [1,2,3,4,5]
triTas2(t)
assert(t == [1,2,3,4,5])
t = [8,7,6,3,7,6,5,10]
triTas2(t)
assert(t == [3,5,6,6,7,7,8,10])
```

1.5 Comparaison

Reprenez les fonctions écrites précédemment (insertion, tamiser, supprimeRacine, ...) pour faire en sorte qu'elles retournent les nombres **d'écritures et de comparaisons**, comme nous l'avions fait pour les tris précédents.

- Comparer les deux stratégies de tri par tas, laquelle est plus rapide ?
- Comparar le tri par tas avec les tris précédents

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: