Nom : _____

Code permanent : _____

Numéro de place : _____

Directives pédagogiques :

- Inscrivez votre nom, prénom, code permanent et numéro de votre place.
- Lisez attentivement toutes les questions et **répondez directement sur le questionnaire**.
- Seule l'utilisation d'un crayon ou stylo est permise, **aucune documentation, calculatrice, téléphone cellulaire, ordinateur, ou autre objet permis**.
- Cet examen contient 4 questions pour 110 points au total. Il y a donc 10 points bonis.
- Faites attention au temps car le barème est établi à 1 point par minute.
- Cet examen contient 16 pages, incluant 3 pages à la fin pour vos brouillons.
- Pour les questions à développement, **écrivez lisiblement et détaillez vos réponses**.
- Vous avez 100 minutes pour compléter cet examen.

BONNE CHANCE !

1	/ 30
2	/ 25
3	/ 20
4	/ 35
Total	/100

Q1. (30) Supposons qu'on veuille trouver le kième plus petit élément d'une collection, A, de données qui n'est pas triée. Par exemple, le 3ième plus petit élément de la collection A = [18, 72, 88, 13] est 72. Un algorithme trivial est de trier la collection. Le kième élément se retrouve alors à la position A[k-1], A trié = [13, 18, 72, 88], donc A[2] = 72.

a) (5) Quelle est la complexité en moyenne de l'algorithme trivial ?

Trier par comparaison est au mieux O(n log n), donc l'algorithme trivial est O(n log n).

Si on peut trier en O(n), si les clés sont discrètes et organisables en "buckets", alors on peut le faire en O(n).

b) (15) Proposez un algorithme qui soit O(n) en moyenne en vous inspirant de la méthode de tri par la médiane. Le code Python du tri par la médiane vous est fourni en **Appendice A**.

Appel direct à `select( A, k, 0, n-1 )`

c) (5) Quelle est la complexité en pire cas de l'algorithme que vous proposez en (b) ?

`select` est $O(n^2)$ en pire cas, donc $O(n^2)$.

d) (5) Existe-t-il un algorithme en O(n) en pire cas (que nous avons vu en cours) ? Lequel ?

BFPRT

Q2. (25) Soit l'ADT Queue (voir **Appendice B**).

a) (15) Donnez une implantation des opérations `enqueue`, `dequeue` et `first` en utilisant que deux piles comme variables d'instances et tel que ces opérations s'exécutent en O(1) en temps amorti.

```
class TwoStackQueue:
    def __init__( self ):
        self._iBox = ArrayStack()
        self._oBox = ArrayStack()
        self._size = 0
        self._enq = True

    def __len__( self ):
        return self._size

    def is_empty( self ):
        return self._size == 0

   def enqueue( self, elem ):
        self._iBox.push( elem )
        self._size += 1

    def first( self ):
        if self.is_empty():
            return False
        if self._oBox.is_empty():
            while not self._iBox.is_empty():
                self._oBox.push( self._iBox.pop() )
        return self._oBox.top()

    def dequeue( self ):
        if self.is_empty():
            return False
        if self._oBox.is_empty():
            while not self._iBox.is_empty():
                self._oBox.push( self._iBox.pop() )
        self._size -= 1
        return self._oBox.pop()
```

b) (10) Donnez une preuve formelle que vos opérations sont en O(1) en temps amorti.

Le coût supplémentaire vient des transferts entre les piles iBox et oBox pour les appels à `first` et `dequeue`.

Si on transfert à chaque fois qu'on `enqueue` un élément, on aura pour N `enqueue` N transfert de 1 élément chaque fois, donc pour N éléments, N opérations `dequeue` * 1 élément à transférer / N éléments gérés au total, donc le coût associé à `dequeue`, T(N) = N / N = 1 est O(1).

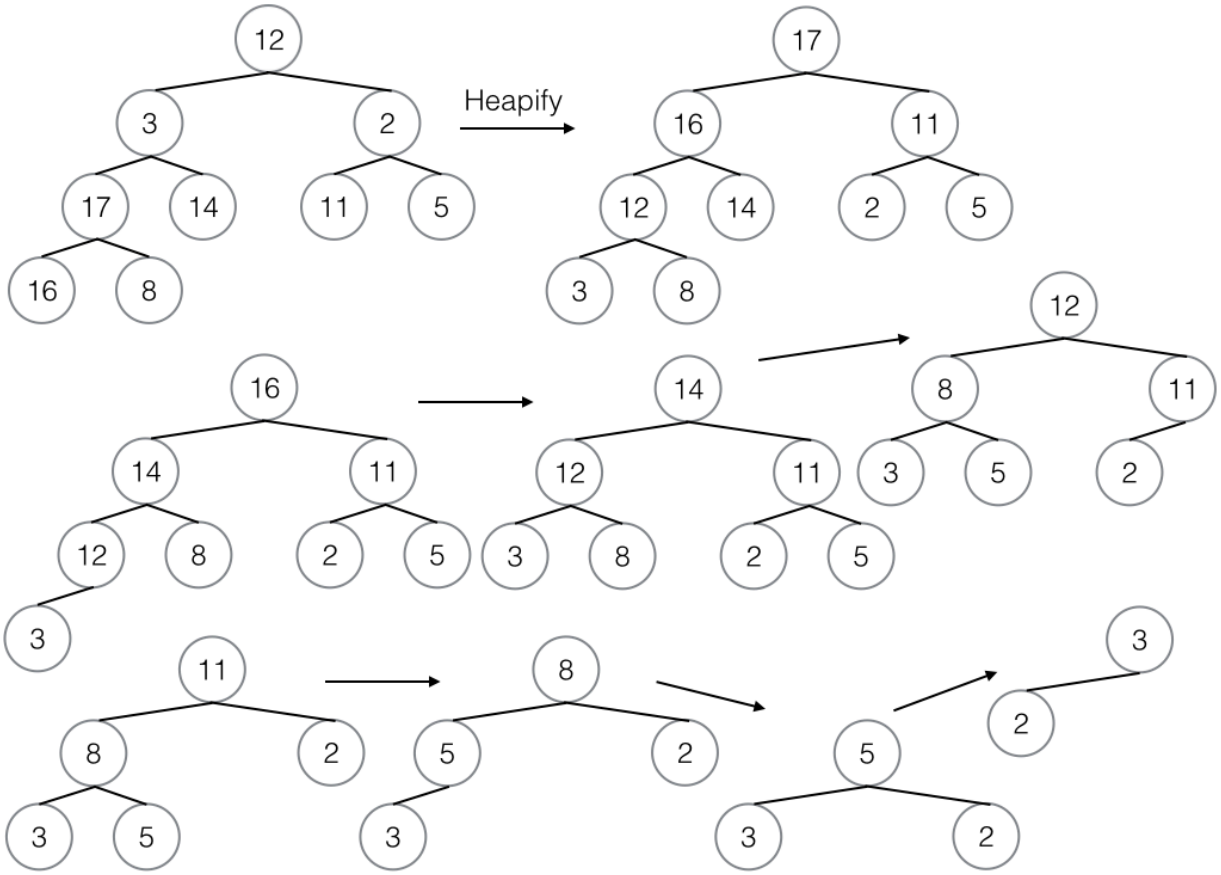
À l'autre extrémité, si on attends que les k éléments soient dans la queue avant de faire un appel à `first` ou `dequeue`, on transfert k éléments pour le premier `dequeue` mais le coût sera nul pour les k-1 suivants, donc pour k éléments, T(N) = k transferts; le coût associé à `dequeue` pour k éléments si on effectue k `first` ou `dequeue` est k/k, soit O(1) si on l'amorti sur pour gérer k opérations.

La somme des éléments à transférer pendant la gestion de N éléments est exactement N, alors en temps amorti pour N opérations T(N) = N/N = 1, donc O(1).

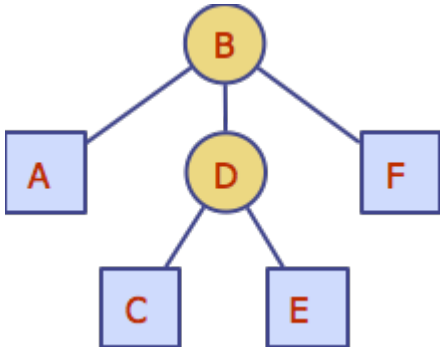
Q3. (20) Remplissez le tableau ci-dessous en montrant à chaque ligne les changements d'états du tableau lorsqu'on le trie par la méthode du monceau (heapsort). Utilisez la page suivante pour dessiner le monceau initial et ses altérations au fur et à mesure que l'algorithme progresse. (*NB. le nombre de lignes dans le tableau ne représente pas nécessairement le nombre exact d'états du tableau*.)

12	3	2	17	14	11	5	16	8
17	16	11	12	14	2	5	3	8
8	16	11	12	14	2	5	3	17
16	14	11	12	8	2	5	3	17
3	14	11	12	8	2	5	16	17
14	12	11	3	8	2	5	16	17
5	12	11	3	8	2	14	16	17
12	8	11	3	5	2	14	16	17
2	8	11	3	5	12	14	16	17
11	8	2	3	5	12	14	16	17
5	8	2	3	11	12	14	16	17
8	5	2	3	11	12	14	16	17
3	5	2	8	11	12	14	16	17
5	3	2	8	11	12	14	16	17
2	3	5	8	11	12	14	16	17
3	2	5	8	11	12	14	16	17
2	3	5	8	11	12	14	16	17
2	3	5	8	11	12	14	16	17

Vos monceaux ici :



Q4. (35) Soit l'arbre général suivant:



a) (10) Décrivez la classe Node pour stocker l'information de chaque noeud d'un arbre général sachant qu'on veut rejoindre rapidement, en O(1), ses enfants et son parent.

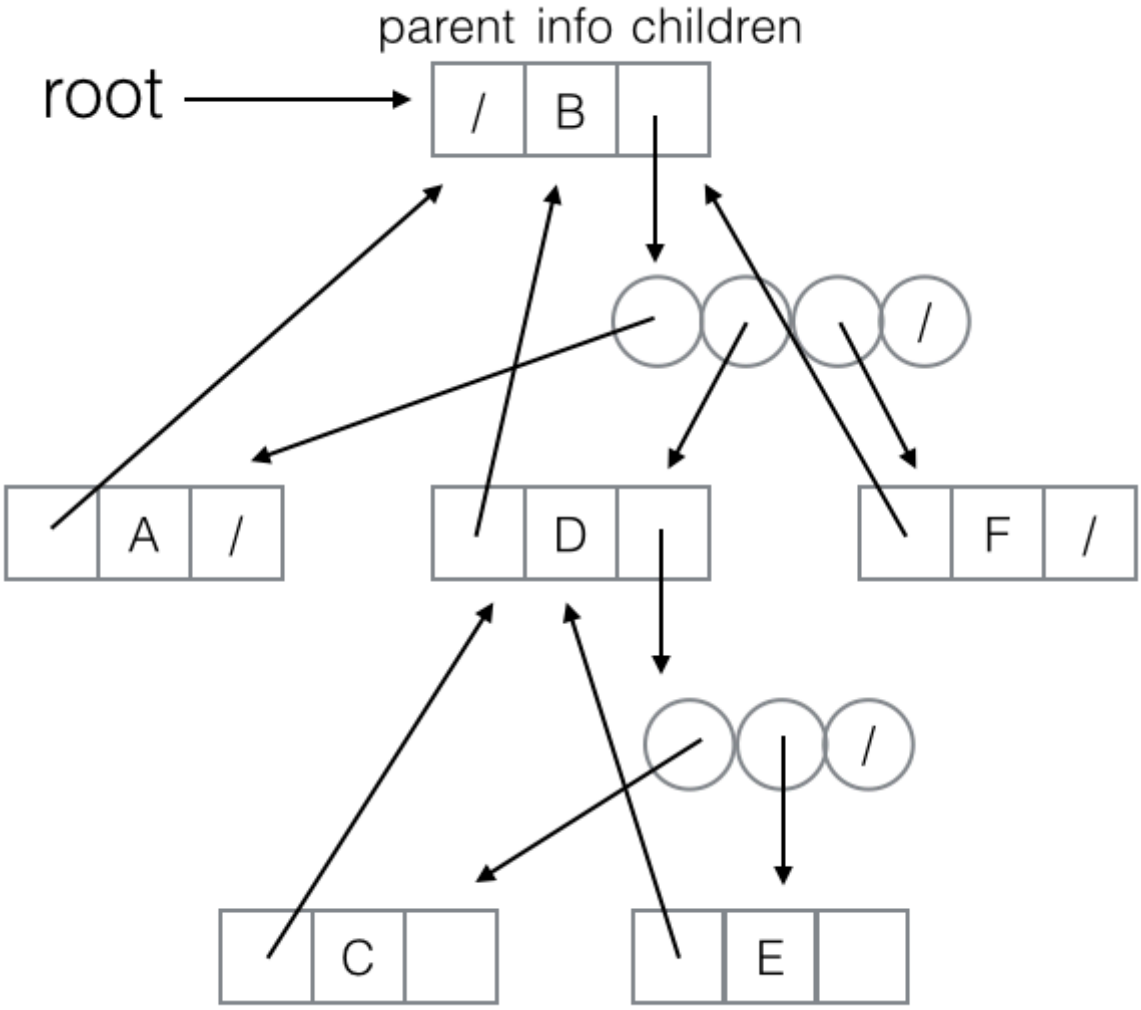
```
class Node:
    def __init__ ( self, info, parent = none, children = [ ] ):
    self._info = info
    self._parent = parent
    self._children = children

def information( self ):
    return self._info

def children( self ):
    return self._children

def parent( self ):
    return parent
```

b) (10) Dessinez la structure interne de cet arbre en utilisant votre classe Node, décrite en (a).



c) (5) Dans quel ordre seront visités les noeuds de cet arbre si on effectue un parcours préfixe ?

B, A, D, C, E, F

d) (5) Dans quel ordre seront visités les noeuds de cet arbre si on effectue un parcours post-fixe ?

A, C, E, D, F, B

e) (5) Dans quel ordre seront visités les noeuds de cet arbre si on effectue un parcours en largeur ?

B, A, D, F, C, E

Appendice A : Tri par la médiane

```
import random

def swap( A, i, j ):
    tmp = A[i]
    A[i] = A[j]
    A[j] = tmp

def partition( A, g, d, iPivot ):
    pivot = A[iPivot]
    swap( A, iPivot, d )

    iPivot = g
    for i in range( g, d ):
        if A[i] <= pivot:
            swap( A, iPivot, i )
            iPivot += 1

    swap( A, iPivot, d )
    return iPivot

def select( A, k, g, d ):
    i = random.randint( g, d )
    iPivot = partition( A, g, d, i )

    if ( g + k - 1 ) == iPivot:
        return iPivot
    if ( g + k - 1) < iPivot:
        return select( A, k, g, iPivot-1 )
    else:
        return select( A, k - ( iPivot-g+1 ), iPivot + 1, d )

def triMediane( A, g, d ):
    if d <= g:
        return

    milieu = (d - g + 1 ) // 2
    mediane = select( A, milieu, g, d )

    triMediane( A, g, mediane - 1 )
    triMediane( A, mediane + 1, d )
```

Appendice B : ADT Queue

```
class Queue:

    def __init__( self ):
        pass

    def __len__( self ):
        pass

    def __str__( self ):
        pass

    def is_empty( self ):
        pass

    #ajouter un élément dans la Queue
    def enqueue( self, element ):
        pass

    #retirer un élément de la Queue
    def dequeue( self ):
        pass

    #retourner le premier élément de la Queue
    #sans le retirer
    def first( self ):
        pass
```

Brouilon 1

Brouilon 2

Brouilon 3