Nom :_____

Numéro de votre place :_____

Code permanent :_____

Directives pédagogiques :

- Inscrivez votre nom, numéro de place et code permanent.
- Sortez votre carte étudiante et mettez la à vue.
- Lisez attentivement toutes les questions et **répondez directement sur le questionnaire**.
- Seule l'utilisation d'un crayon est permise, **aucune documentation, calculatrice, téléphone cellulaire, ordinateur, ou autre objet**.
- Cet examen contient 10 questions pour 175 points au total, dont 10 points en bonus.
- Le barème est établi à 1 point par minute environ.
- Cet examen contient 18 pages, incluant 3 pages détachables à la fin pour vos brouillons.
- Pour les questions à développement, **écrivez lisiblement et détaillez vos réponses**.
- Vous avez 165 minutes pour compléter cet examen.

BONNE CHANCE !

1	/ 15
2	/ 15
3	/ 30
4	/ 10
5	/ 20
6	/ 15
7	/ 15
8	/ 15
9	/30
10	/10
Total	/165

1. (15) Un roi fou possède *n* bouteilles de vin et un espion en a empoisonné une sans savoir laquelle. Une seule goutte de ce poison diluée même dans un ratio de 1:1000000 cause la mort après 1 mois. Mettez au point une stratégie pour déterminer exactement quelle bouteille de vin a été empoisonnée en un seul mois et en utilisant dans O(log *n*) dégustateurs.

On utilise [lg n] dégustateurs. On numérote les bouteilles de 1 à n, mais en nombres binaires de [lg n] bits, où chacun bit correspond à l'un des [lg n] dégustateurs. Si l'on aligne les numéros binaires des bouteilles les uns en dessous des autres, une colonne de bits correspond à un dégustateur. Chaque dégustateur devra goûter les bouteilles dont le numéro en binaire contient un 1 à son bit (sa colonne). La bouteille empoisonnée sera celle dont le pattern des dégustateurs morts correspondra à son numéro binaire.

2. (15) Montrez comment utiliser une pile `S` et une queue `Q` pour générer tous les sous-ensembles possibles (sans répétition) d'un ensemble de *n* éléments `T` de manière non récursive.

```
def subsetsi( T ):

    S = ArrayStack()
    Q = ArrayQueue()

    output = []
    i = 0
    while i < len(T):
        for n in range(i,len(T)):
            S.push(n)
            Q.enqueue(S)

        i += 1
        while not Q.is_empty():
            temp_stack = Q.dequeue()
            temp = []
            while not temp_stack.is_empty():
                temp.append(temp_stack.pop())
            output.append(temp)

    return output
```

3. (30) Deux arbres T1 et T2 sont dits isomorphes si et seulement si une des 2 propriétés suivantes est réalisée: i) T1 et T2 sont vides; ou, ii) les racines de T1 et T2 possèdent le même nombre de sous-arbres et le ième sous-arbre de T1 est isomorphe au ième sous-arbre de T2 pour i = 1..k, où k est le nombre maximal d'enfants par noeud. La hauteur du plus petit arbre est *h*.

   a) (15) Donnez un algorithme (en python ou pseudo-code) qui teste si 2 arbres sont isomorphes.

```
def isomorphic( T1, T2 ):
    if T1.is_empty() and T2.is_empty():
        return True
    if T1.height2( T1.root() ) != T2.height2( T2.root() ):
        return False
    for c1,c2 in [T1.children( T1.root() ),T2.children( T2.root() )]:
        if not isomorphic( c1, c2 ):
            return False
    return True
```

b) (5) Quel est le temps d'exécution de votre algorithme dans le meilleur cas ?

$O(1)$

c) (10) Quel est le temps d'exécution de votre algorithme en pire cas ?

$O(\sum i=1..h \quad 2^i)$

4. (10) Considérez une queue de priorité implantée par un monceau-min dans un tableau dont les indices débutent à 0.

a) (1) À quel(s) indice(s) peut se retrouver la plus petite clé ?

{0}

b) (2) À quel(s) indice(s) peut se retrouver la 3ème plus petite clé ?

{1, 2, 3, 4, 5, 6}

c) (3) À quel(s) indice(s) peut se retrouver la 4ème plus petite clé ?

{1, 2, 3, …, 14}

d) (4) À quel(s) indice(s) peut se retrouver la ième plus petite clé, si i > 1 ?

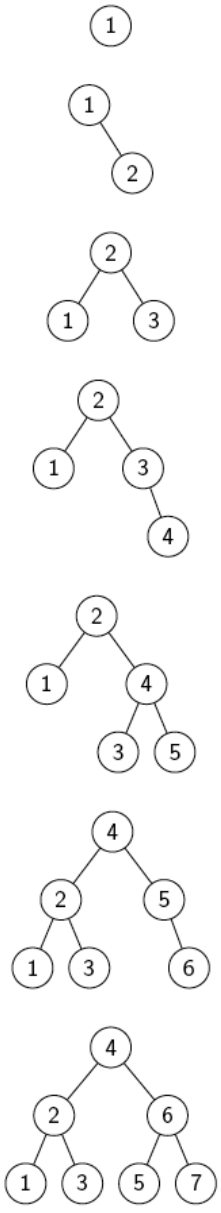
$\{1, 2, 3, \ldots, 2^i-2\}$

5. (20) Un groupe d'enfants veut jouer à un jeu nommé *Inmonopoly*, où à chaque coup le joueur avec le plus d'argent doit donner la moitié de son argent au joueur avec le plus petit montant d'argent. Quel(s) structure(s) de données doit-on utiliser pour jouer à ce jeu de manière efficace ? Pourquoi ?

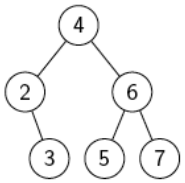
Un monceau-min et un monceau-max. Il faut avoir accès efficacement à la fois au joueur le plus pauvre et au joueur le plus riche, mettre à jour leurs clés efficacement et mettre à jour efficacement les nouveaux joueurs le plus pauvre et le plus riche. Pour ceci, on peut donc maintenir les joueurs à la fois dans un monceau-min et dans un monceau-max.

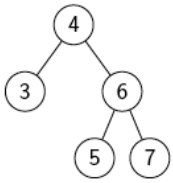
6. (15) Considérez les arbres de recherche AVL.

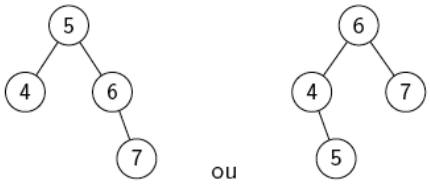
   a) (10) Insérez les clés {1, 2, 3, 4, 5, 6, 7} dans cet ordre dans un arbre AVL initialement vide. Dessiner les arbres résultants après chaque opération d'insertion.

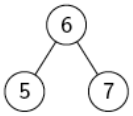


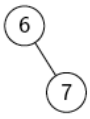
b) (5) Supprimez une à une toutes les clés de votre arbre AVL résultant en (a), en ordre croissant. Dessiner l'arbre résultant après chaque opération de suppression. Lorsqu'un noeud doit être remplacé, utilisez le prédécesseur. À noter : si votre arbre AVL de départ (le dernier obtenue en (a)) n'est pas bon, vous obtiendrez la note 0 en (b).





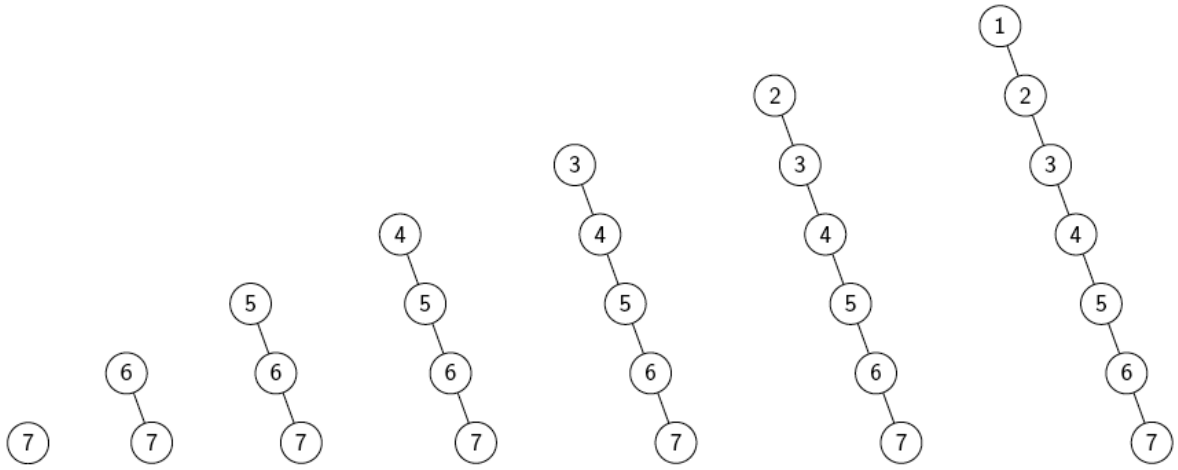




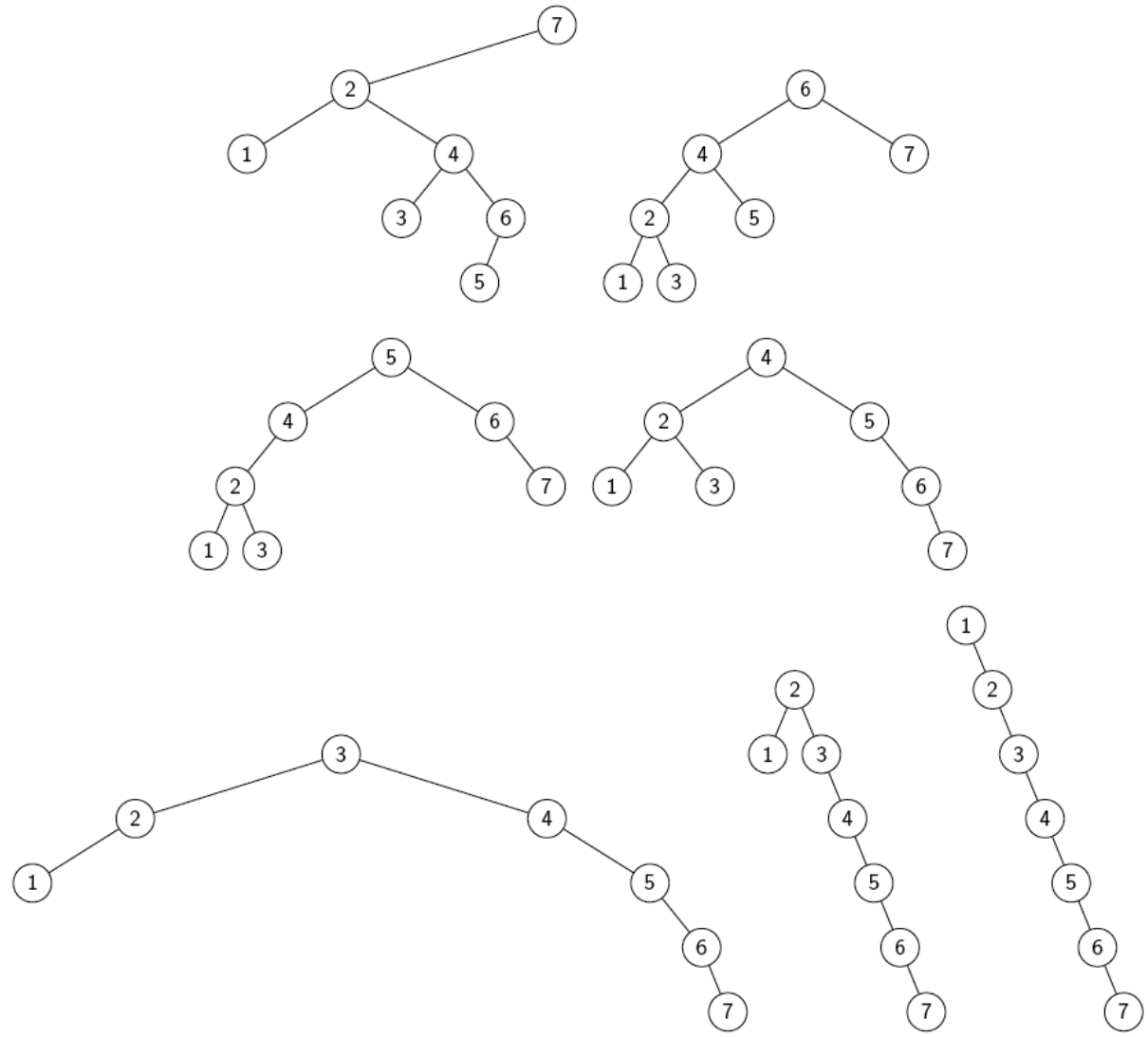


7. (15) Considérez les arbres de recherche « Splay ».

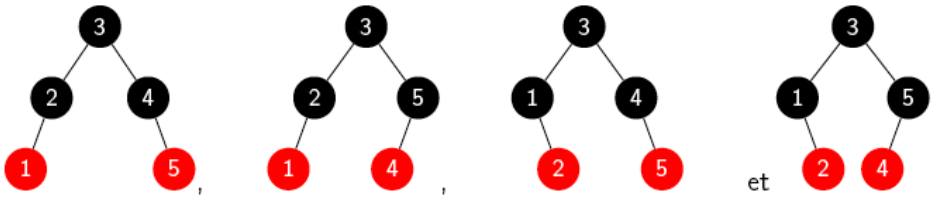
   a) (10) Insérez les clés {7,6,5,4,3,2,1} dans cet ordre dans un arbre « Splay » initialement vide.

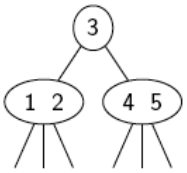


b) (5) Accédez une à une à toutes les clés de votre arbre « Splay » résultant de (a), en ordre décroissant. Dessinez l'arbre résultant après chaque opération d'accession. À noter : si votre arbre « Splay » de départ (le dernier obtenu en (a)) n'est pas bon, vous aurez 0 en (b).



8. (15) Imaginez et dessinez un arbre de recherche 2-4 qui possède au moins 4 représentations différentes en arbres rouge-noir et dessinez ces 4 arbres rouge-noir.

Par exemple, les arbres 

correspondent tous à l'arbre 

.

9. (30) Considérez l'ADT Map et les structures de données « Skip List » et arbres de recherche rouge-noir pour l'implanter.

a) (1) Combien de pointeurs nécessitent un noeud d'une « Skip List » ?

4 (ou 5 en incluant explicitement l'élément).

b) (1) Combien de pointeurs nécessitent un noeud d'un arbre rouge-noir ?

3 (ou 4 en incluant explicitement l'élément).

c) (10) On s'attend à utiliser combien de mémoire pour stocker une Map de *n* clés dans une « Skip List » ?

O(n)

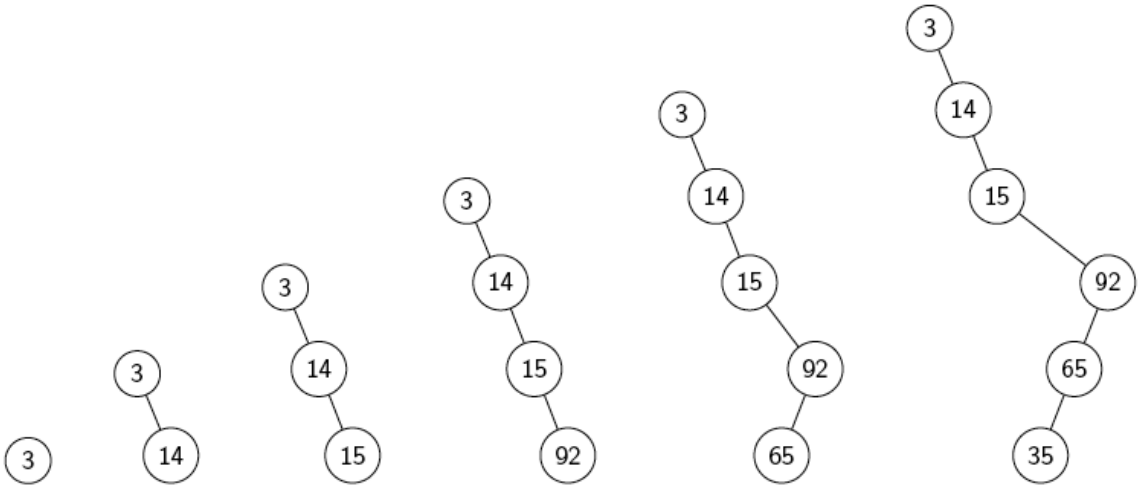
d) (8) On s'attend à utiliser combien de mémoire pour stocker une Map de *n* clés dans un arbre rouge-noir ?

O(n)

e) (10) Énumérez toutes les bonnes raisons que vous connaissez pour implanter une Map avec un arbre rouge-noir plutôt qu'avec une « Skip List ».

- Demande moins de mémoire.
- Opérations en pire cas en O(log n) garanties.

10. (10) Considérez les arbres binaires de recherche. Insérez les clés {3,14,15,92,65,35} dans cet ordre dans un arbre binaire de recherche (ABR) initialement vide.



Brouillon :

Brouillon :

Brouillon :