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 8 questions pour 160 points au total.
- Le barème est établi à 1 point par minute environ.
- Cet examen contient 19 pages, incluant 3 Appendices et 3 pages détachables à la fin pour vos brouillons.
- Pour les questions à développement, **écrivez lisiblement** et **détaillez vos réponses**.
- Vous avez 160 minutes pour compléter cet examen.

BONNE CHANCE et BON ÉTÉ !

1	/ 20
2	/ 20
3	/ 35
4	/ 30
5	/ 20
6	/ 10
7	/ 10
8	/ 15
Total	/ 160

1. (20) Considérez l'ADT `Map` (Appendice A) et une implantation avec une liste non triée (Appendice B).

a) (10) Donnez une implantation de la méthode `items()` directement dans la class `UnsortedListMap` qui exécute en O(n), où n est le nombre de clés. Rappelez-vous que `items()` est une méthode qui implante un itérateur permettant de parcourir toutes les paires, *clé-valeur,* d'une `Map`.

```
def __items__( self ):
    for item in self._T:
        yield ( item._key, item._value )
```

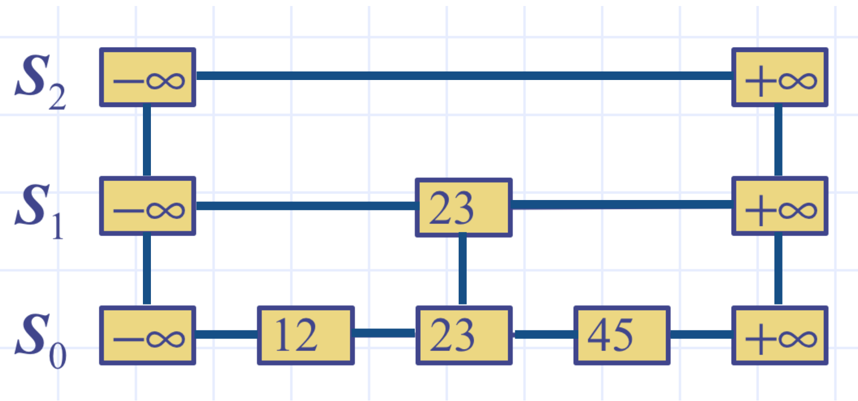
b) (5) Quelle est la complexité en temps en pire cas pour insérer n paires *clé-valeur* dans une `UnsortedListMap` initialement vide. Expliquez votre raisonnement.

Chaque insertion est dans *O(n)*, donc $\sum_{i=1}^{n} i = \underline{O(n^2)}$.

c) (5) Quelle est la complexité en temps en pire cas pour retirer n paires *clé-valeur* d'une `UnsortedListMap` qui contient initialement n paires. Expliquez votre raisonnement.

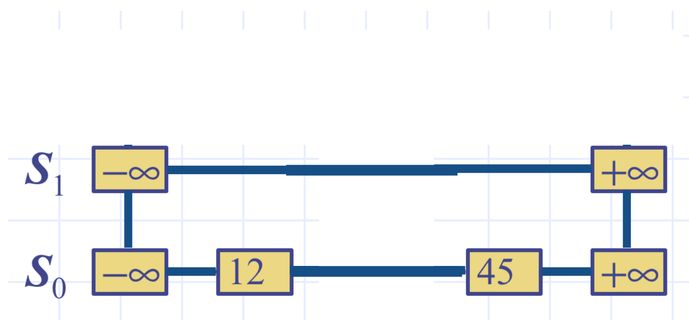
Pour chaque deletion, on traverse la liste, donc $\sum_{i=1}^{n} i = \underline{O(n^2)}$.

2. (20) Considérez la skip list, *S* :

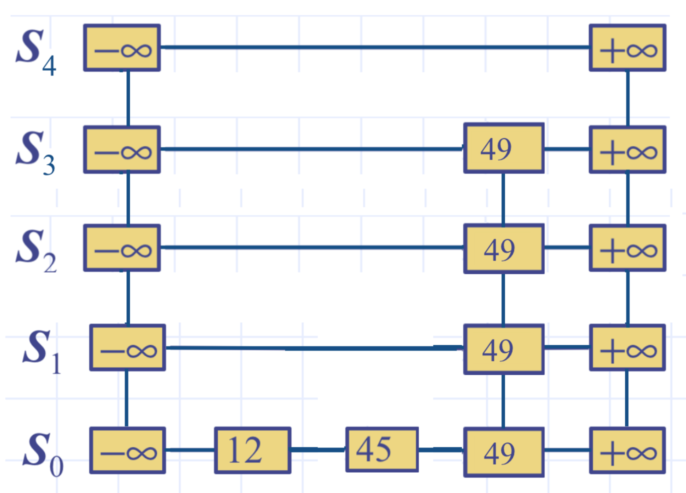


a) (10) Dessinez *S* après chaque opération en prenant les valeurs de `coin_flip()` suivantes : `True, True, True, False, True, False, True, True, False`.

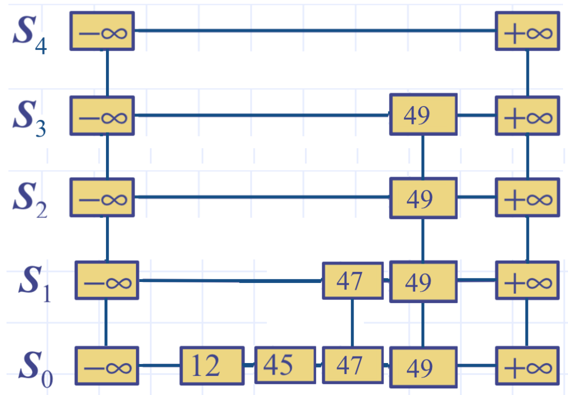
`del S[23]:`



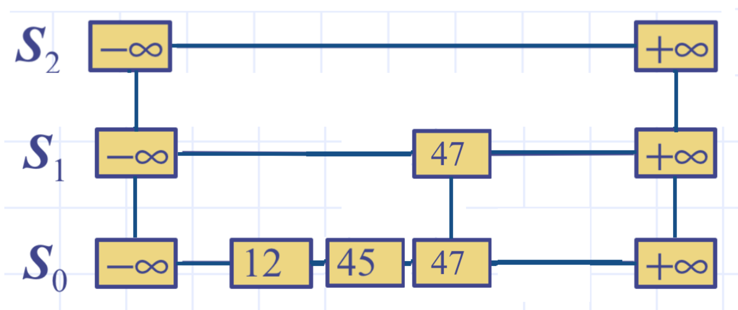
`S[49] = 'x':`



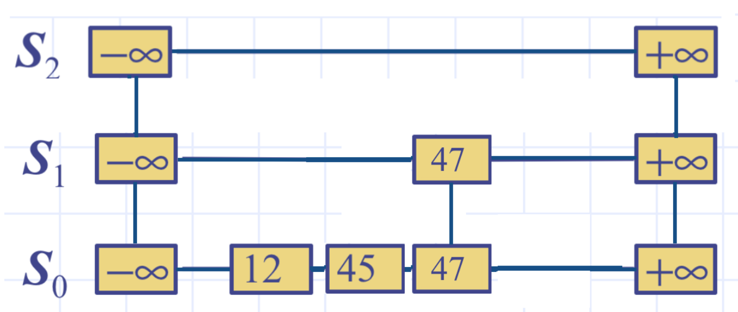
`S[47] = 'y'`



`del S[49]`



`S[45] = 'z'`



b) (10) Combien de comparaisons de clés (`element`) au total ont été effectuées pour les 5 opérations par la méthode `SkipSearch` (Appendice C).

$$3 + 3 + 6 + 5 + 4 = 21$$

3. (35) Considérez les tables de hachage résultant de l'utilisation des fonction de hachage h(k) = (3k + 2) mod 11 (fonction primaire) et d(k) = 5 - (k mod 5) (fonction secondaire) pour insérer les clés 5, 12, 7, 8, 11, 4, 1, 3, 10, 6, 9 dans cet ordre.

a) (15) en assumant que les collisions sont prises en charge par sondage linéaire (linear probing). Montrez les états de la table après chaque insertion (de haut en bas).

						5				
					12	5				
	7				12	5				
	7			8	12	5				
	7	11		8	12	5				
	7	11	4	8	12	5				
	7	11	4	8	12	5	1			
3	7	11	4	8	12	5	1			
3	7	11	4	8	12	5	1			10
3	7	11	4	8	12	5	1		6	10
3	7	11	4	8	12	5	1	9	6	10
0	1	2	3	4	5	6	7	8	9	10

h(5) = **6** ; h(12) = **5** ; h(7) = **1** ; h(8) = **4** ; h(11) = **2** ; h(4) = **3** ;
h(1) = 5, déjà prise : 5 + 1 = 6, déjà prise: 6 + 1 = **7** ;
h(3) = **0** ;
h(10) = **10** ;
h(6) = **9** ;
h(9) = 7, déjà prise : 7 + 1 = **8** ;

b) (15) en assumant que les collisions sont prises en charge par hachage double. Montrez les états de la table après chaque insertion (de haut en bas).

						5				
					12	5				
	7				12	5				
	7			8	12	5				
	7	11		8	12	5				
	7	11	4	8	12	5				
	7	11	4	8	12	5			1	
3	7	11	4	8	12	5			1	
3	7	11	4	8	12	5			1	10
3	7	11	4	8	12	5	6		1	10
3	7	11	4	8	12	5	6	9	1	10
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

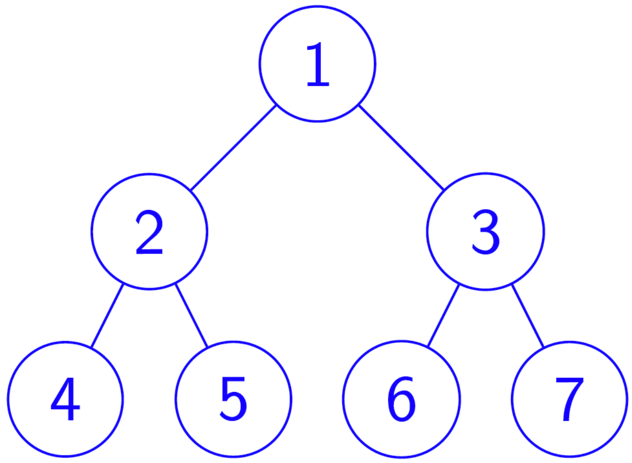
h(5) = **6** ; h(12) = **5** ; h(7) = **1** ; h(8) = **4** ; h(11) = **2** ; h(4) = **3** ;
h(1) = 5, déjà prise : d(1) = 4, donc 5 + 4 = **9** ;
h(3) = **0** ;
h(10) = **10** ;
h(6) = 9, déjà prise : d(6) = 4, donc 9 + 4 = 2, déjà prise : donc 2 + 4 = 6, déjà prise : donc 6 + 4 = 10, déjà prise : donc 10 + 4 = 3, déjà prise : donc 3 + 4 = **7** ;
h(9) = 7, déjà prise : donc d(9) = 1, donc 7+1 = **8** ;

c) (5) Pourquoi le hachage double est-il en général préféré au sondage linéaire ?

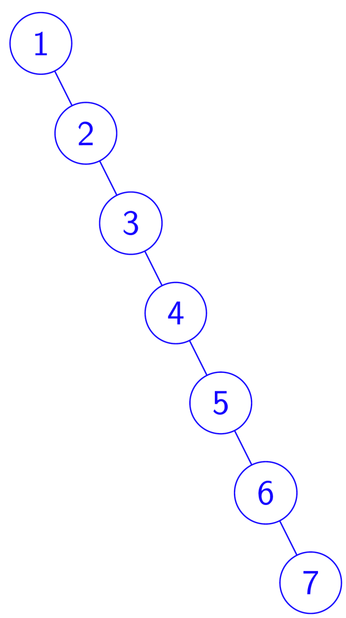
Pour éviter les agglomérations (“clusters”) autour de certaines valeurs, e.g. dans l’exemple autour des valeurs 4 et 5.

4. (30) Dessinez l'arbre final après l'insertion des clés { 1, 2, 3, 4, 5, 6, 7 } dans cet ordre, dans un arbre initialement vide de type :

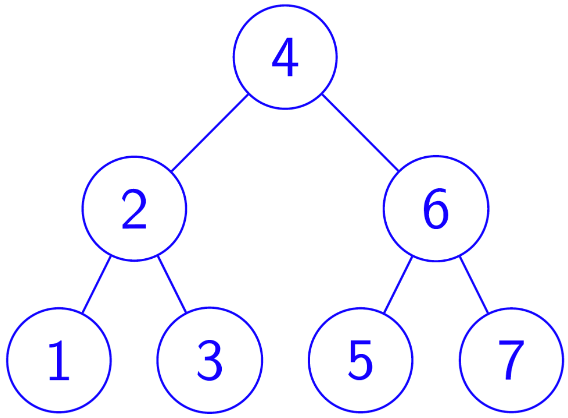
a) (5) Monceau



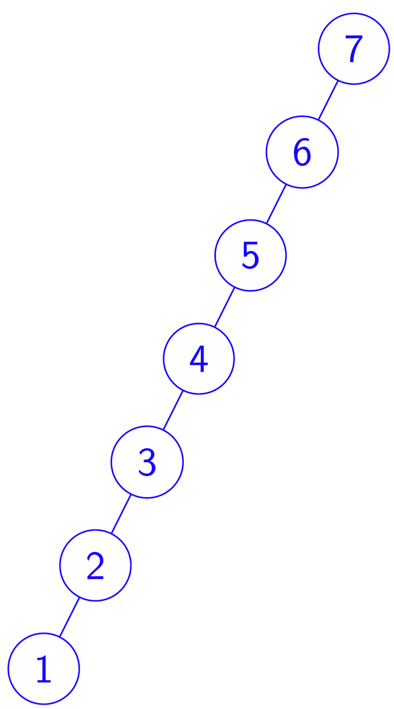
b) (5) Arbre binaire de recherche



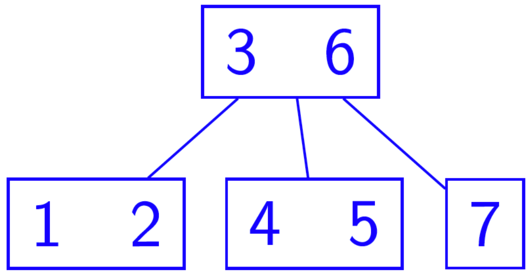
c) (5) Arbre AVL



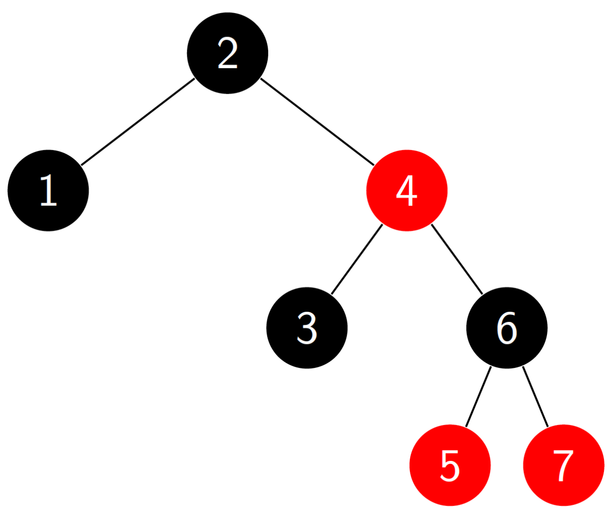
d) (5) Arbre "Splay"



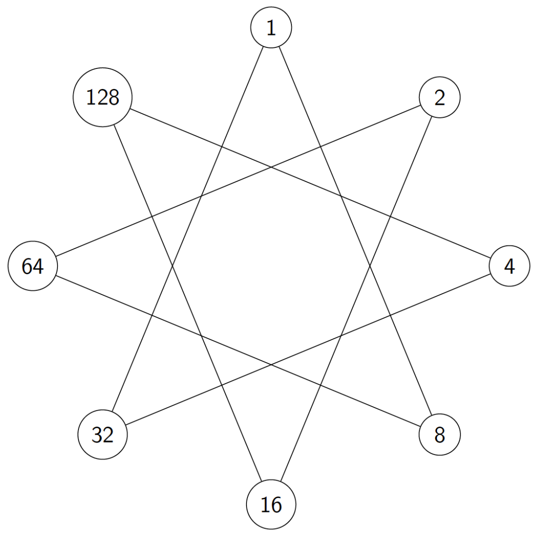
e) (5) Arbre 2-4



f) (5) Arbre rouge-noir



5. (20) Soit le graphe suivant. En respectant l'ordre croissant des noeuds adjacents, dans quel ordre seront visités les noeuds du graphe si on débute au noeud 1 :



a) (10) lors d'un parcours en profondeur ? (PS. une seule réponse est possible)

1, 8, 64, 2, 16, 128, 4, 32

b) (10) lors d'un parcours en largeur ? (PS. une seule réponse est possible)

1, 8, 32, 64, 4, 2, 128, 16

6. (10) Expliquez comment utiliser un arbre AVL ou rouge-noir pour trier *n* éléments comparables en temps dans *O(n log n)* en pire cas (évidemment sans trier au préalable).

On insère toutes les valeurs dans l'arbre au coût dans *O(n log n)*, puisque l'insertion de chaque valeur est dans *O(log n).*

Ensuite on retire *n* fois la plus petite valeur de l'arbre qui se trouve au bout dans la branche à droite de la racine. L'opération delete retourne cette valeur et coûte au plus *log* n, donc *n* fois *log n* = *O(n log n)*.

On aura donc au total *2 x n log n* qui est dans *O(n log n)*.

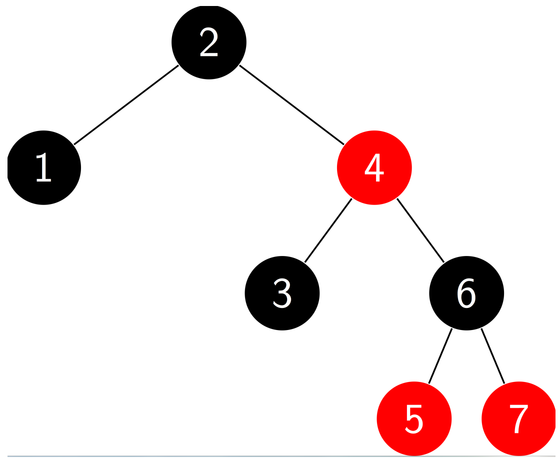
7. (10) Peut-on utiliser un arbre “Splay” pour trier *n* éléments comparables en temps dans $O(n \log n)$ en pire cas (évidemment sans trier au préalable) ? Pourquoi ou pourquoi pas ?

On insère toutes les valeurs dans l’arbre “Splay” où le coût de chaque insertion est dans $O(h)$, où h est la hauteur de l’arbre, qui malheureusement en pire cas est dans $O(n)$. Donc cette opération à elle seule nous coûte $O(n^2)$ en pire cas.

Il est donc impossible d’utiliser un arbre “Splay” pour trier n éléments comparables en temps dans $O(n \log n)$ en pire cas.

8. (15) Soit un arbre rouge-noir.

a) (5) Dessinez un arbre rouge-noir dont la différence entre la hauteur du sous-arbre gauche et celle du sous-arbre droit de la racine est maximale.



b) (10) Dites pourquoi une différence de hauteur plus grande entre deux sous-arbres de n'importe quel noeud n'est pas possible.

Chaque fois qu'on ajoute un noeud noir sur un chemin on ne peut pas rajouter plus de 2 noeuds rouges autour du noir et si on ajoute un noeud noir dans un sous-arbre, il faut en ajouter un dans le sous-arbre voisin.

Appendice A : Map.py

```
import collections

class Map( collections.MutableMapping ):

    #nested _Item class
    class _Item:
        __slots__ = '_key', '_value'

        def __init__( self, k, v = None ):
            self._key = k
            self._value = v

        def __eq__( self, other ):
            return self._key == other._key

        def __ne__( self, other ):
            return not( self == other )

        def __lt__( self, other ):
            return self._key < other._key

        def __ge__( self, other ):
            return self._key >= other._key

        def __str__( self ):
            return "<" + str( self._key ) + "," + str( self._value ) + ">"

        def key( self ):
            return self._key

        def value( self ):
            return self._value

    def is_empty( self ):
        return len( self ) == 0

    def get( self, k, d = None ):
        if self[k]:
            return self[k]
        else:
            return d

    def setdefault( self, k, d = None ):
        if self[k]:
            return self[k]
        else:
            self[k] = d
            return d
```

Appendice B : UnsortedListMap.py

```
from Map import Map

class UnsortedListMap( Map ):

    def __init__( self ):
        self._T = []

    def __getitem__( self, k ):
        for item in self._T:
            if k == item._key:
                return item._value
        return False

    def __setitem__( self, k, v ):
        for item in self._T:
            if k == item._key:
                item._value = v
                return
        #no match
        self._T.append( self._Item( k, v ) )

    def __delitem__( self, k ):
        for j in range( len( self._T ) ):
            if k == self._T[j]._key:
                self._T.pop( j )
                return
        return False

    def __len__( self ):
        return len( self._T )

    def __iter__( self ):
        for item in self._T:
            yield item._key

    def __contains__( self, k ):
        return self[k]
```

Appendice C : SkipSearch

```
def SkipSearch( self, element ):
    p = self._start
    while not( p._belo is None ):
        p = p._belo
        while element >= p._next._elem:
            p = p._next
    return p
```

Brouillon

Brouillon

Brouillon