Name :____

Place number :__________________________________

Code permanent :________________________________

Directives :

- Write you name, place number and permanent code.
- Put your student card in view
- Read all questions and **answer directly on the questionnaire**.
- You can only use a pen or pencil, **no documentation, calculator, phone, computer or object**.
- This exam contains 9 questions for 165 points, including 10 bonus points.
- Approximately, we estimate about 1 point per minute.
- This exam contains 20 pages, including 3 detachable pages at the end for drafting and 3 Appendix.
- For developing questions, **write clearly and detail your answers**.
- You have 165 minutes to complete this exam.

GOOD LUCK AND HAPPY HOLIDAYS!

1	/ 25
2	/ 20
3	/ 25
4	/ 10
5	/ 25
6	/ 30
7	/ 10
8	/ 10
9	/ 10
Bonus	/ 10
Total	/ 165

1. (25) Consider the data structure `Map` (Appendix A) and an implentation with a list that we keep unstructured (Appendix B).

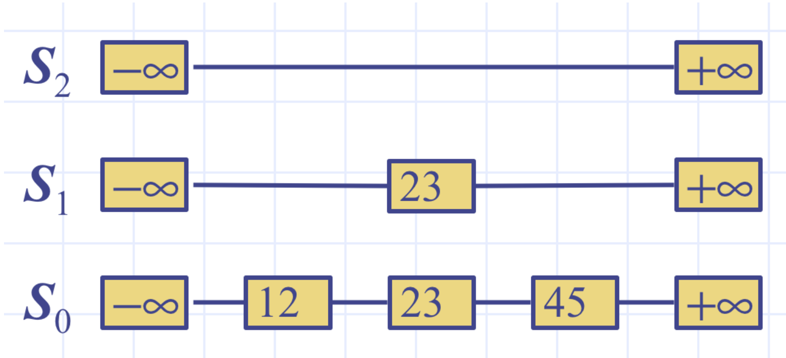
   a) (15) Give an implementation of the `items()` method directly in the class `UnsortedListMap` that guarantees O(n) time, for n keys in the `Map`. Remember that `items()` is a method that implements an `iterator` and which allows to scan all keys in a `Map`.

   ```
   def __items__( ):
   ```

   b) (5) What is the complexity in time in the worst case to insert n pairs `key-value` in an `UnsortedListMap` initially empty.

   c) (5) What is the complexity in time in the worst case to remove n pairs `key-value` in an `UnsortedListMap` that initially contains n pairs `key-value`.

2. (20) Consider the skip list, *S*, and the following operations :



a) (10) Draw *S* after each operation by taking the coin_flip() following values : `_FACE`, `_FACE`, `_FACE`, `_TAILS`, `_FACE`, `_TAILS`, `_FACE`, `_TAILS`. Consult Appendix C for the code of `SkipInsert`.

```
del S[12]:
```

```
S[31] = 'x':
```

```
S[47] = 'y'
```

```
del[31]
```

```
S[45] = 'z'
```

b) (10) How many comparisons in total will be made to accomplish the 5 operations by the `SkipSearch` method (Appendix C).

3. (25) Consider the hashing tables resulting from using the following hashing functions h(i) = (2i + 3) mod 11 (primary function) and h'(k) = 5 - (k mod 5) (secondary function) to insert the keys 3, 13, 26, 23, 11, 36, 54, 12, 8, 65, 20 in this order

a) (10) assuming the collisions are solved by single chaining (linear probing). Show the states of the table after each insertion.

0	1	2	3	4	5	6	7	8	9	10

b) (15) assuming the collisions are solved by double hashing. Show the states of the table after each insertion.

0	1	2	3	4	5	6	7	8	9	10

4. (10) Consider binary search trees. Insert the keys { 1, 61, 80, 33, 98, 87 } in this order in a binary search tree (BST) initially empty.

BONUS (10) Of which famous number the keys to insert form the decimals?

5. (25) Consider the AVL search trees.

a) (15) Insert in an AVL tree initially empty the following keys { 10, 20, 30, 40, 50, 60, 70 }, in this order. Draw the trees resulting after each insertion.

b) (10) Remove one by one and in increasing order the keys in the AVL tree obtained in (a). Draw the resulting tree after each suppression. When a node must be replaced, use the predecessor. N.B. If the AVL tree you obtained in (a) is wrong, you will have 0 here.

6. (30) Consider the 2-4 and red-black trees.

   a) (15) Insert in a 2-4 tree initially empty the following keys { 70, 60, 50, 40, 30, 20, 10 }, in this order. Draw the trees resulting after each insertion.

b) (15) Insert in a red-black tree initially empty the following keys { 70, 60, 50, 40, 30, 20, 10 }, in this order. Draw the trees resulting after each insertion.

7. (10) Given the following dynamic programming table to compare two strings of characters:

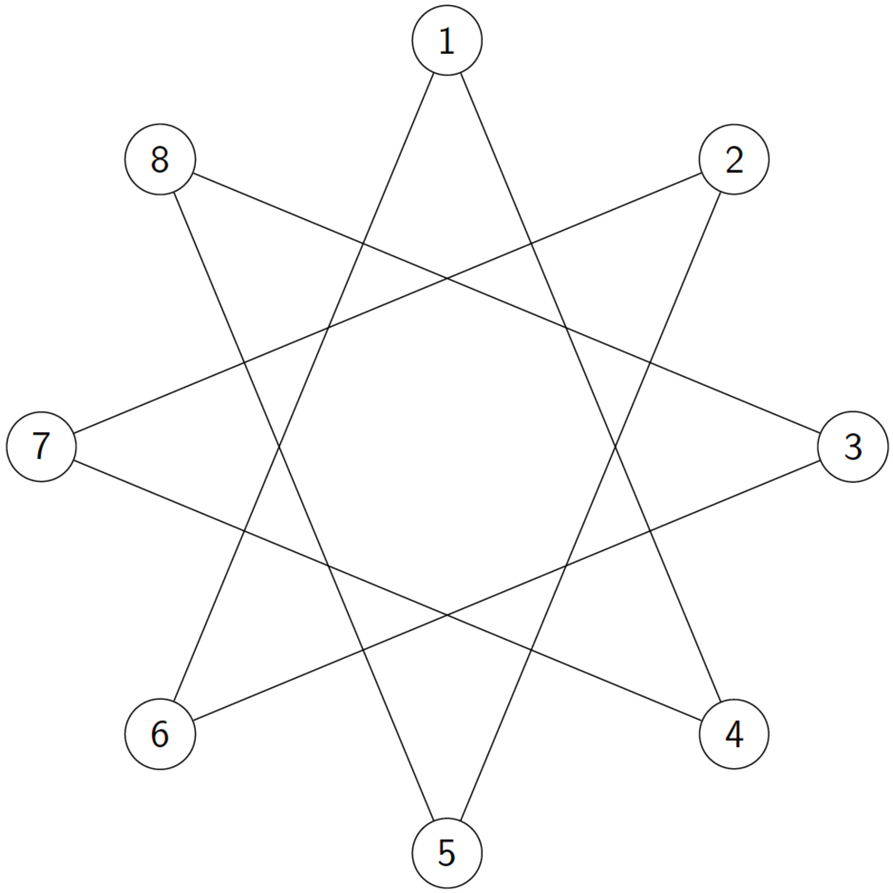
		m	y	m	m	e	c	a	c	e	c	o	i	t	
		0	1	2	3	4	5	6	7	8	9	10	11	12	13
t	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
l	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
y	2	0	0	0	0	0	0	0	0	0	0	0	0	0	1
m	3	0	0	1	1	1	1	1	1	1	1	1	1	1	1
m	4	0	1	1	2	2	2	2	2	2	2	2	2	2	2
e	5	0	1	1	2	3	3	3	3	3	3	3	3	3	3
p	6	0	1	1	2	3	4	4	4	4	4	4	4	4	4
f	7	0	1	1	2	3	4	4	4	4	4	4	4	4	4
c	8	0	1	1	2	3	4	4	4	4	4	4	4	4	4
u	9	0	1	1	2	3	4	5	5	5	5	5	5	5	5
p	10	0	1	1	2	3	4	5	5	5	5	5	5	5	5
i	11	0	1	1	2	3	4	5	5	5	5	5	5	5	5
t	12	0	1	1	2	3	4	5	5	5	5	5	5	6	6
	13	0	1	1	2	3	4	5	5	5	5	5	5	6	7

a) (5) What is the longest common subsequence?

b) (5) Blackened the path visited to obtain the longest common subsequence, i.e. the answer you gave in (a).

8. (10) Draw the standard `trie` that contains the following strings (preserve the alphabetic order of the children) : { arbre, trie, arc, tree, arete, cycle, clique }

9. (10) Given the following graph. By respecting the increasing order of the adjacent nodes, in which order will the nodes be visited if we start at node 1:



a) (5) using a depth-first-search? (PS. only one possible answer)

b) (5) using breadth-first-search? (PS. only one possible answer)

Draft :

Draft :

Draft :

Appendix 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
```

Appendix 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]
```

Appendix C : SkipSearch et SkipInsert

```
#search element
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


#insert element
def SkipInsert( self, element ):
    p = self.SkipSearch( element )
    if p._elem == element: #The keys are equal
        p._elem = element  #We must set the element to the new value
        return p
    #p points to the previous node
    #we're at the bottom level, so belo is None
    q = self.insertAfterAbove( p, None, element )

    #Insert at higher levels as determined by the coin_flip
    i = 0
    coin_flip = self._coin.flip()
    while coin_flip == _FACE:
        i += 1 #i indicates the current level of insertion
        if i >= self._height:
            self.increaseHeight()
        while p._abov is None:
            #we move to the previous Node
            p = p._prev
        #we move up one Node (to get to the desired level)
        p = p._abov
        #q is the previously inserted Node (belo the one to be inserted)
        q = self.insertAfterAbove( p, q, element )
        coin_flip = self._coin.flip()
    #before exiting, we increase the skip list count by one
    self._count += 1
    #we return the last inserted Node (the top level one)
    return q
```