Name :____

Place number :______________________________

Permanent code :___________________________________

Directives:

- Write you name, place number and permanent code.
- Make visible your student identification card.
- Read all questions and **write your answers directly on the questionnaire**.
- Only use a pen or pencil. **No documentation, calculator, cell phone, computer, or other objects allowed**.
- This exam has 8 questions for 160 points in total.
- The scale was established to about 1 point per minute.
- This exam contains 19 pages, including 3 Appendices and 3 detachable sheets at the end for your draft.
- For developing questions, **write clearly and detail your answers**.
- You have 160 minutes to complete this exam.

GOOD LUCK AND HAVE A NICE SUMMER!

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

1. (20) Consider the `Map` ADT (Appendix A) and an implementation using a non sorted list (Appendix B).

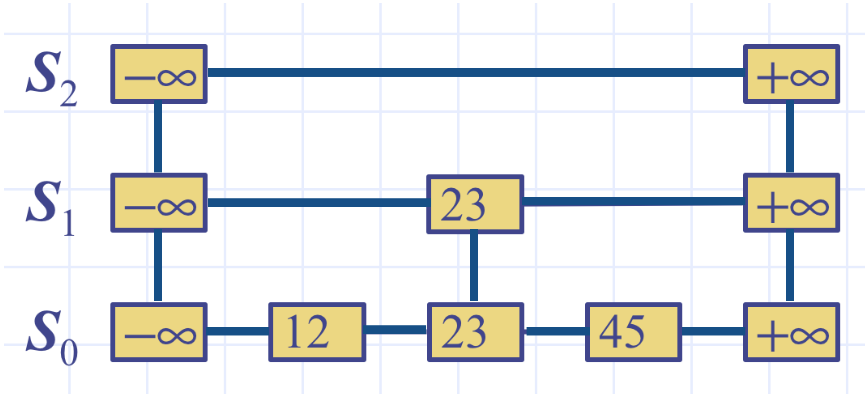
   a) (10) Give an implementation of the `items()` method directly in the `UnsortedListMap` class that execute in O(*n*), where *n* is the number of keys. Recall that `items()` is a method that implements an iterator allowing for visiting all key-*value* pairs in a `Map`.

   ```
   def __items__( self ):
   ```

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

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

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



a) (10) Draw *S* after each operation by taking the following sequence of `coin_flip()`: `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) How many key (`element`) comparisons were made in total for the 5 operations during the `SkipSearch` method (Appendix C).

3. (35) Consider hashing tables resulting from using the following hash functions: h(k) = (3k + 2) mod 11 (primary function) and d(k) = 5 - (k mod 5) (secondary function) to insert the following keys: 5, 12, 7, 8, 11, 4, 1, 3, 10, 6, 9, in this order.

a) (15) by resolving the collisions by linear probing. Show each state of the table, i.e. after each insertion (from top to bottom).

0	1	2	3	4	5	6	7	8	9	10

b) (15) by resolving the collisions by double hashing. Show each state of the table, i.e. after each insertion (from top to bottom)

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

c) (5) Why in general one would prefer double hashing over linear probing?

4. (30) Draw the final tree after the insertion of the keys: { 1, 2, 3, 4, 5, 6, 7 }, in this order, in an initially empty tree of type:

a) (5) Heap

b) (5) Binary search tree

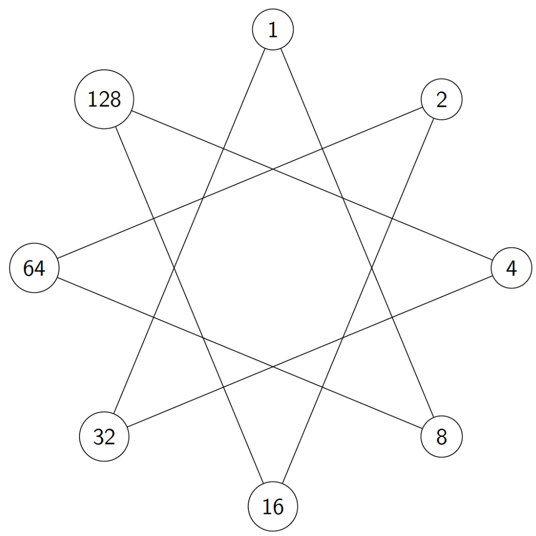
c) (5) AVL tree

d) (5) Splay tree

e) (5) 2-4 tree

f) (5) Red-black tree

5. (20) Consider the following graph. By respecting the increasing order of the vertices, in which order will the vertices be visited if one starts at vertex 1:



a) (10) using a depth-first search? (PS. there is only one possible order)

b) (10) using a breadth-first search? (PS. there is only one possible order)

6. (10) Explain how one can use an AVL or red-black tree to sort *n* comparable elements in time in *O(n log n)* worst-case (without of course pre-sorting the elements).

7. (10) Can we use a Splay tree to sort *n* comparable elements in time in *O(n log n)* worst-case (without of course pre-sorting the elements) ? Why or why not?

8. (15) Consider a red-black tree.

   a) (5) Draw a valid red-black tree with the maximum height difference between the left and right sub-trees of the root.

   b) (10) Say why the maximum height difference between the two sub-trees of any node in a red-black tree cannot be higher.

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

```
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
```

Draft

Draft

Draft