

Dictionnaire
dict Python
Classe *Map*
UnsortedListMap
SortedListMap
SkipListMap
Maxima Set

Dictionnaire

- Un dictionnaire (mais aussi une carte ou “map” en anglais) est une collection interrogable d’éléments qui sont des paires clé-valeur
- Ses principales opérations sont la recherche, l’insertion et la suppression d’éléments
- Plusieurs éléments avec la même clé ne sont pas autorisés
- Applications:
 - carnet d'adresses
 - base de données de clients
 - "mapping" d'information
- La classe *dict* de Python est sans doute sa structure de données la plus significative

dict de Python

- M[k]:** Return the value *v* associated with key *k* in map *M*, if one exists; otherwise raise a `KeyError`. In Python, this is implemented with the special method `__getitem__`.
- M[k] = v:** Associate value *v* with key *k* in map *M*, replacing the existing value if the map already contains an item with key equal to *k*. In Python, this is implemented with the special method `__setitem__`.
- del M[k]:** Remove from map *M* the item with key equal to *k*; if *M* has no such item, then raise a `KeyError`. In Python, this is implemented with the special method `__delitem__`.
- len(M):** Return the number of items in map *M*. In Python, this is implemented with the special method `__len__`.
- iter(M):** The default iteration for a map generates a sequence of *keys* in the map. In Python, this is implemented with the special method `__iter__`, and it allows loops of the form, `for k in M`.

```
>>> d = {1:'bonjour', 2:'hello', 3:'buenos dias'}
>>> d
{1: 'bonjour', 2: 'hello', 3: 'buenos dias'}
>>> d[0]
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    d[0]
KeyError: 0
>>> d[1]
'bonjour'
>>> d[4] = 'صباح الخير'
>>> d
{1: 'bonjour', 2: 'hello', 3: 'buenos dias', 4: 'صباح الخير'}
>>> del d[2]
>>> d
{1: 'bonjour', 3: 'buenos dias', 4: 'صباح الخير'}
>>> len( d )
3
>>> for k in d:
    print( k )

1
3
4
>>>
```

dict de Python

`k in M`: Return `True` if the map contains an item with key `k`. In Python, this is implemented with the special `__contains__` method.

`M.get(k, d=None)`: Return `M[k]` if key `k` exists in the map; otherwise return default value `d`. This provides a form to query `M[k]` without risk of a `KeyError`.

`M.setdefault(k, d)`: If key `k` exists in the map, simply return `M[k]`; if key `k` does not exist, set `M[k] = d` and return that value.

`M.pop(k, d=None)`: Remove the item associated with key `k` from the map and return its associated value `v`. If key `k` is not in the map, return default value `d` (or raise `KeyError` if parameter `d` is `None`).

```
>>> 3 in d
True
>>> d.get( 2, 'pas là' )
'pas là'
>>> d.get( 1 )
'bonjour'
>>> d.get( 5 )
>>> d.setdefault( 1, 'hello' )
'bonjour'
>>> d.setdefault( 5, 'ahoj' )
'ahoj'
>>> d.pop( 6, 'pas là' )
'pas là'
>>> d.pop( 5 )
'ahoj'
>>> d
{'صباح الخير': 4, 'buenos dias': 3, 'bonjour': 1}
```

dict de Python

`M.popitem()`: Remove an arbitrary key-value pair from the map, and return a `(k,v)` tuple representing the removed pair. If map is empty, raise a `KeyError`.

`M.clear()`: Remove all key-value pairs from the map.

`M.keys()`: Return a set-like view of all keys of `M`.

`M.values()`: Return a set-like view of all values of `M`.

`M.items()`: Return a set-like view of `(k,v)` tuples for all entries of `M`.

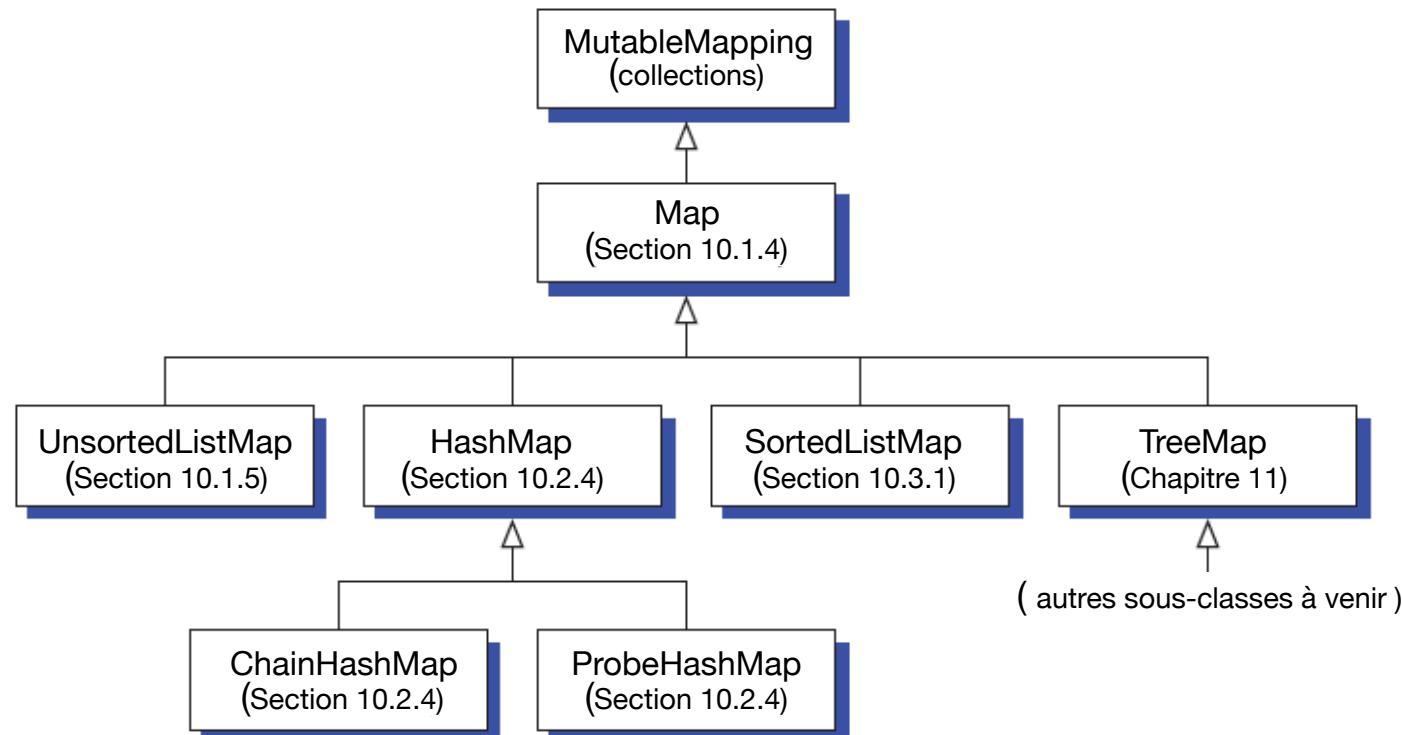
`M.update(M2)`: Assign `M[k] = v` for every `(k,v)` pair in map `M2`.

`M == M2`: Return `True` if maps `M` and `M2` have identical key-value associations.

`M != M2`: Return `True` if maps `M` and `M2` do not have identical key-value associations.

```
>>> d.popitem()
(4, 'صباح الخير')
>>> d.clear()
>>> d = {1:'bonjour', 2:'hello', 3:'buenos dias'}
>>> d.keys()
dict_keys([1, 2, 3])
>>> d.values()
dict_values(['bonjour', 'hello', 'buenos dias'])
>>> d.items()
dict_items([(1, 'bonjour'), (2, 'hello'), (3, 'buenos dias')])
>>> d.update( {6: 'hallo', 7: 'buongiorno'} )
>>> d
{1: 'bonjour', 2: 'hello', 3: 'buenos dias', 6: 'hallo', 7: 'buongiorno'}
>>> d == {1: 'bonjour', 2: 'hello', 3: 'buenos dias', 6: 'hallo', 7: 'buongiorno'}
True
>>> d != {1: 'bonjour', 2: 'hello'}
True
```

Classe Map et ses dérivées



MutableMapping

<code>MutableMapping</code>	<code>Mapping</code>	<code>__getitem__</code> , <code>__setitem__</code> , <code>__delitem__</code> , <code>__iter__</code> , <code>__len__</code>	Inherited <code>Mapping</code> methods and <code>pop</code> , <code>popitem</code> , <code>clear</code> , <code>update</code> , and <code>setdefault</code>
-----------------------------	----------------------	---	--

Tous les objets Python ne sont pas gérés de la même manière. Les objets mutables peuvent être modifiés. Les objets immuables ne peuvent pas être modifiés et de nouveaux objets sont créés lors des mises à jour.

```
#collections inclut MutableMapping
import collections

#ADT Map (Classe de base)
class Map( collections.MutableMapping ):

    #classe imbriquée pour les éléments ( clé, valeur )
    class _Item:
        #enregistrement statique et efficace
        __slots__ = '_key', '_value'

        #constructeur avec valeur None de défaut pour valeur
        def __init__( self, k, v = None ):
            self._key = k
            self._value = v

        #égalité de clé
        def __eq__( self, other ):
            return self._key == other._key

        #non-égalité de clé
        def __ne__( self, other ):
            return not( self == other )

        #clé strictement inférieure
        def __lt__( self, other ):
            return self._key < other._key

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

        #prettyprint d'un élément
        def __str__( self ):
            return "<" + str( self._key ) + "," + str( self._value ) + ">"

        #accès à la clé
        def key( self ):
            return self._key

        #accès à la valeur
        def value( self ):
            return self._value
```

```
#Map vide?
def is_empty( self ):
    return len( self ) == 0

#prettyprint de Map
def __str__( self ):
    if self.is_empty():
        return "{}"
    pp = "{"
    for item in self.items():
        pp += str( item )
    pp += "}"
    pp += " size = "
    pp += str( len( self ) )
    return pp

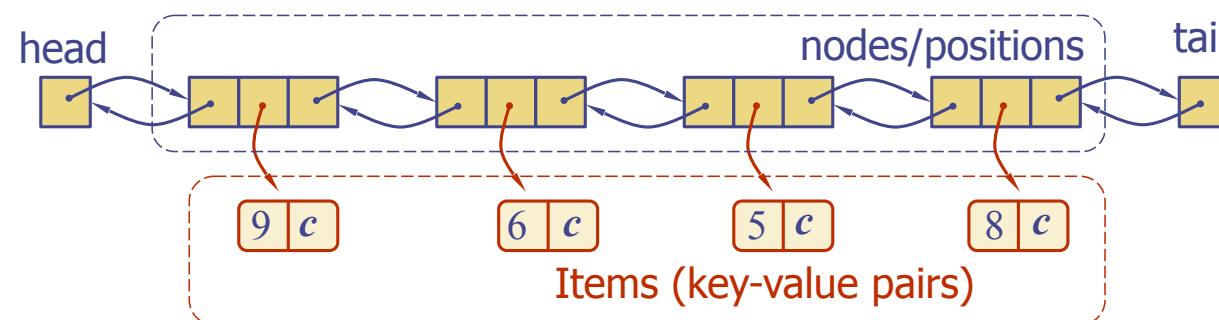
#accède l'élément de clé k
#retourne sa valeur s'il existe, d sinon
#efficacité de __getitem__ va dépendre de
#son implémentation dans la sous-classe
def get( self, k, d = None ):
    global tmp
    try:
        tmp = self[k]
    except KeyError:
        return d
    return tmp

#assigne et retourne l'élément de clé k
#rien ne change si il existe, prend la valeur d sinon
#efficacité de __getitem__ va dépendre de
#son implémentation dans la sous-classe
def setdefault( self, k, d = None ):
    global tmp
    try:
        tmp = self[k]
    except:
        self[k] = d
    return tmp
```

Implémentation de *Map* avec une liste

Nous pouvons implémenter *Map* avec une liste non triée

Nous stockons les éléments d'une *Map* dans une liste S dans un ordre arbitraire, en utilisant une liste doublement chaînée par exemple.



```

from Map import Map
import random
import time

#Implémentation de Map avec une liste non triée
class UnsortedListMap( Map ):

    #on utilise la list Python
    def __init__( self ):
        self._T = []

    #accède à l'élément de clé k en O(n) en pire cas
    #retourne l'élément si présent, None sinon
    def __getitem__( self, k ):
        #on doit parcourir la liste
        for item in self._T:
            if k == item._key:
                return item._value
        raise KeyError( k )

    #insertion d'un élément de clé k en O(n) en pire cas
    def __setitem__( self, k, v ):
        #on doit parcourir la liste
        #car si la clé existe, on change la valeur
        for item in self._T:
            if k == item._key:
                item._value = v
                return
        #si la clé n'existe pas, on ajoute un nouvel élément
        self._T.append( self._Item( k, v ) )

    #suppression de l'élément de clé k en O(n) en pire cas
    #soulève une erreur si la clé n'existe pas
    def __delitem__( self, k ):
        #on doit parcourir la liste
        for j in range( len( self._T ) ):
            if k == self._T[j]._key:
                self._T.pop( j )
                return
        #élément non trouvé : erreur
        raise KeyError( k )

    #taille de la Map
    def __len__( self ):
        return len( self._T )

    #itérateur de clés
    def __iter__( self ):
        for item in self._T:
            yield item._key

    #appartenance de la clé k en O(n) en pire cas
    def __contains__( self, k ):
        return self[k]

    #itérateur de (clé, valeur)
    def __items__( self ):
        for item in self._T:
            yield ( item._key, item._value )

```

Performances de *Map* implémentée avec une liste non-triée

- L'insertion d'un élément prend $O(1)$ -temps parce que nous pouvons insérer le nouvel élément à la fin de la liste (avec *append*)
 - TOUTEFOIS: puisque l'insertion appelle d'abord une recherche de la clé, l'opération d'insertion prendra au total $O(n)$ -temps
- La recherche ou la suppression d'un élément prend $O(n)$ -temps, car dans le pire des cas (l'élément n'est pas trouvé), nous parcourons toute la liste pour rechercher un élément avec la clé donnée
- L'implémentation avec une liste non triée n'est pratique que pour des *Map* de petites tailles.

```
UnsortedListMap unit testing...
Insertion of 31600 (over 50000 attempts) keys in 30.961766958236694 seconds.
Access to 50000 keys in 65.59298777580261 seconds.
Insertion of 50000 keys in a list takes 0.06965470314025879 seconds.
Access of 50000 keys in a list takes 23.15487313270569 seconds.
```

```

from Map import Map
import random
import time

class SortedListMap( Map ):

    #recherche dichotomique pour trouver en O(log n) en pire cas la clé k
    #vue en section 1 Mise à niveau/1.4 Récursivité
    #retourne j, l'index de l'élément le plus à gauche avec clé >= k :
    #tel que T[low:j] ont des clés < k
    #        T[j:high+1] ont des clés >= k
    #j retourné est entre 0 et len( self )
    def _find_index( self, k, low, high ):
        if high < low:
            return high + 1
        else:
            mid = (low + high) // 2
            if k == self._T[mid]._key:
                return mid
            elif k < self._T[mid]._key:
                return self._find_index( k, low, mid - 1 )
            else:
                return self._find_index( k, mid + 1, high )

    #utilise la list de Python
    def __init__( self ):
        self._T = []

    #taille de la Map
    def __len__( self ):
        return len( self._T )

    #accède l'élément de clé k
    #bénéficie de la liste triée pour le faire en O(log n)
    def __getitem__( self, k ):
        j = self._find_index( k, 0, len( self._T ) - 1 )
        #j == len(T) ou T[j] != k => non trouvée
        if j == len( self._T ) or self._T[j]._key != k:
            raise KeyError( k )
        return self._T[j]._value

```

Dans le code de `SortedListMap.py`, on trouve un ensemble de méthodes pour identifier les éléments de clés min et max, des éléments de clés $<$, $>$, \leq , \geq à un clé k passée en argument et pour itérer sur un intervalle d'éléments entre deux valeurs de clés.

► Nous y reviendrons un peu plus tard dans le module.

```

    #insertion d'un élément de clé k et valeur v
    #bénéficie de la liste triée pour trouver l'index d'insertion en O(log n)
    def __setitem__( self, k, v ):
        j = self._find_index( k, 0, len( self._T ) - 1 )
        #si j < len(T) et clé de T[j] == k => clé existe
        #donc on remplace sa valeur par v
        if j < len( self._T ) and self._T[j]._key == k:
            self._T[j]._value = v
        else:
            #clé n'existe pas, on insère un nouvel élément
            self._T.insert( j, self._Item( k, v ) )

    #suppression de l'élément de clé k
    #bénéficie de la liste triée pour trouver son index en O(log n)
    #soulève une erreur si la clé n'existe pas
    def __delitem__( self, k ):
        j = self._find_index( k, 0, len( self._T ) - 1 )
        #si j == len(T) ou clé de T[j] diffère de k => non trouvée
        if j == len( self._T ) or self._T[j]._key != k:
            raise KeyError( k )
        self._T.pop( j )

    #itérateur sur les clés de la Map
    def __iter__( self ):
        for item in self._T:
            yield item._key

    #itérateur sur les clés dans l'ordre inverse
    def __reversed__( self ):
        for item in reversed( self._T ):
            yield item._key

```

Performances de *Map* implémentée avec une liste triée

- L'insertion et la suppression d'un élément prennent $O(n)$ -temps, puisque nous devons décaler les éléments, et ceci malgré la recherche binaire rapide pour trouver une clé en $O(\log n)$ -temps.
- L'implémentation de la liste triée est efficace pour des *Map* de grande taille

```
SortedListMap unit testing...
Insertion of 50000 keys in  0.5663139820098877 seconds.
Access of 50000 keys in  0.7683331966400146 seconds.
Delete   50000 keys in  0.5168347358703613 seconds.
```

```
Insertion of 31600 keys in a list takes  0.08459019660949707 seconds.
Access of 50000 keys in an unsorted list takes  25.462931871414185 seconds.
```

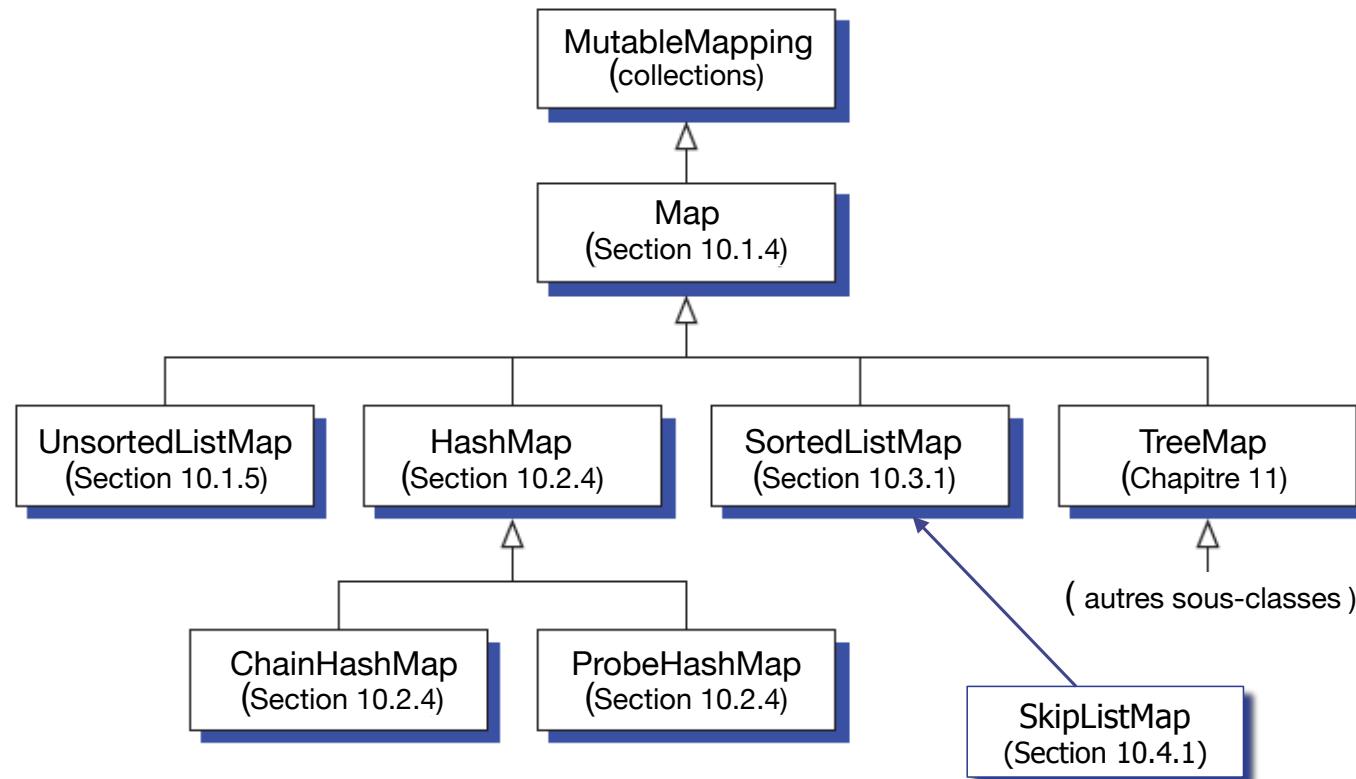
Est-ce qu'on peut faire mieux ?

Une implémentation avec une liste triée est très sympa !

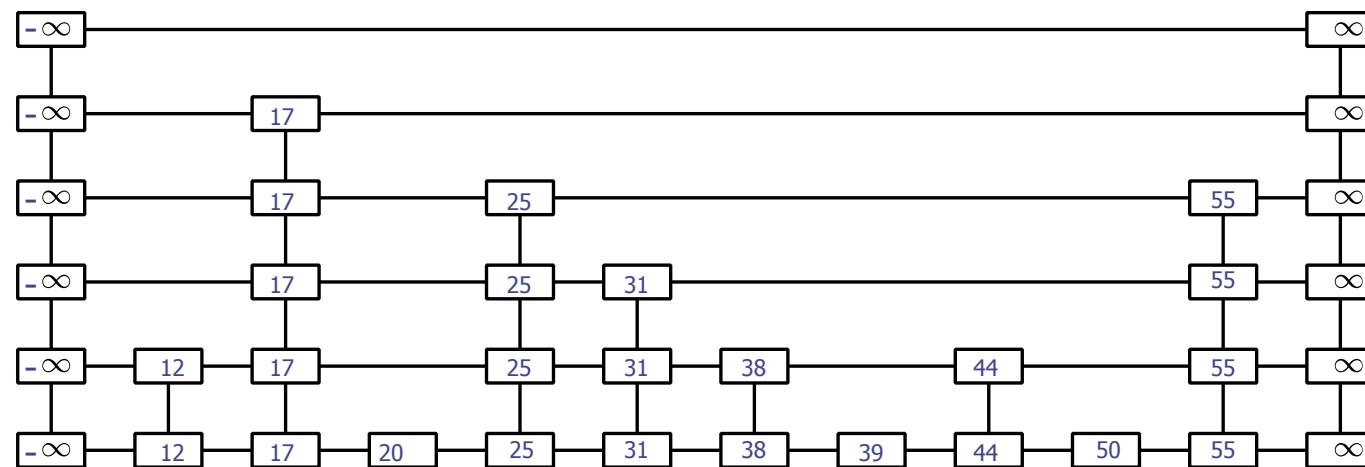
On bénéficie grandement de la recherche binaire en $O(\log n)$ -temps
Cependant, les méthodes de mise à jour sont dans **$O(n)$ -temps**

Pouvons-nous faire mieux?

La SkipList implémente *Map*



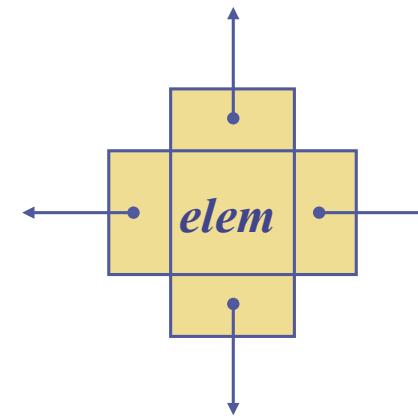
Une SkipList de 10 éléments



```
#class noeud pour une skiplist
class SkipListNode:

    #on a besoin de pointeurs sur l'élément, puis sur les noeuds
    #précédent, suivant, au-dessous et au-dessus
    __slots__ = '_elem', '_prev', '_next', '_belo', '_abov'
    def __init__( self, elem, prev = None, next = None, belo = None, abov = None ):
        self._elem = elem
        self._prev = prev
        self._next = next
        self._belo = belo
        self._abov = abov

    #prettyprint
    def __str__( self ):
        return "(" + str( self._elem ) + ")"
```



```

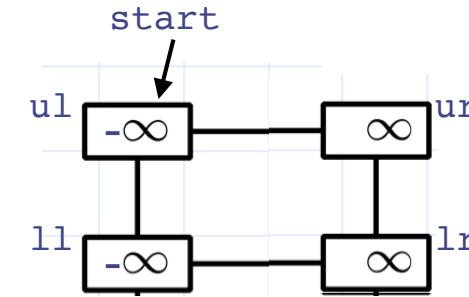
import sys
import random
from SkipListNode import SkipListNode
from Coin import Coin

_FACE = True
_TAILS = False
_MAX = sys.maxsize
_MIN = -sys.maxsize - 1

#ADT SkipList
class SkipList():

    def __init__( self, _MIN_VALUE = _MIN, _MAX_VALUE = _MAX ):
        #valeurs pour les sentinelles, -infini à +infini
        self._MIN_VALUE = _MIN_VALUE
        self._MAX_VALUE = _MAX_VALUE
        #une instance de Coin pour les tours de hauteurs variables
        self._coin = Coin()
        #hauteur de départ
        self._height = 1
        #taille initiale de 0
        self._size = 0
        #les 4 sentinelles et leurs interrelations
        #lr = low-right; ll = low-left; ul = upper-left; ur = upper-right
        sentinel_lr = SkipListNode( self._MAX_VALUE )
        sentinel_ll = SkipListNode( self._MIN_VALUE, None, sentinel_lr, None, None )
        sentinel_lr._prev = sentinel_ll
        sentinel_ul = SkipListNode( self._MIN_VALUE, None, None, sentinel_ll, None )
        sentinel_ur = SkipListNode( self._MAX_VALUE, sentinel_ul, None, sentinel_lr, None )
        sentinel_ul._next = sentinel_ur
        sentinel_ll._abov = sentinel_ul
        sentinel_lr._abov = sentinel_ur
        #par convention, le début de la skiplist est la sentinelle _ul
        self._start = sentinel_ul

```



```

import random

#class Coin pour flipper des coins
class Coin:

    def flip( self ):
        return random.randint( 0, 1 ) == 0

```

```

#taille de la skiplist
def __len__( self ):
    return self._size

# prettyprint
def __str__( self ):
    #on prend le début de la skiplist
    current = self._start
    pp = "SkipList of height " + str( self._height ) + ":\n"
    #pour chaque niveau
    for level in range( self._height, -1, -1 ):
        p = current
        pp += "level " + str( level ) + " ["
        #le premier élément du niveau est le suivant de la sentinelle
        p = p._next
        while not( p is None ):
            if( not p._next is None ):
                pp += str( p._elem )
                if( not p._next._next is None ):
                    pp += ", "
            p = p._next
        #on descend par la sentinelle
        current = current._belo
        pp += "]\n"
    return pp

#iterator : parcours les éléments du niveau 0
def __iter__( self ):
    #on prend le niveau du début de la skiplist (plus haut)
    level = self._start
    #on descend jusqu'au niveau 0 par les sentinelles
    while not( level._belo is None ):
        level = level._belo
    #on parcourt les éléments du niveau 0,
    #le premier étant le suivant de la sentinelle
    current = level._next
    while not( current._next is None ):
        yield current._elem
        current = current._next

```

```

SkipList of height 5:
level 5 [
level 4 [ 44 ]
level 3 [ 44 ]
level 2 [ 44, 55]
level 1 [ 44, 55]
level 0 [12,17,20,25,31,38,39,44,50,55]

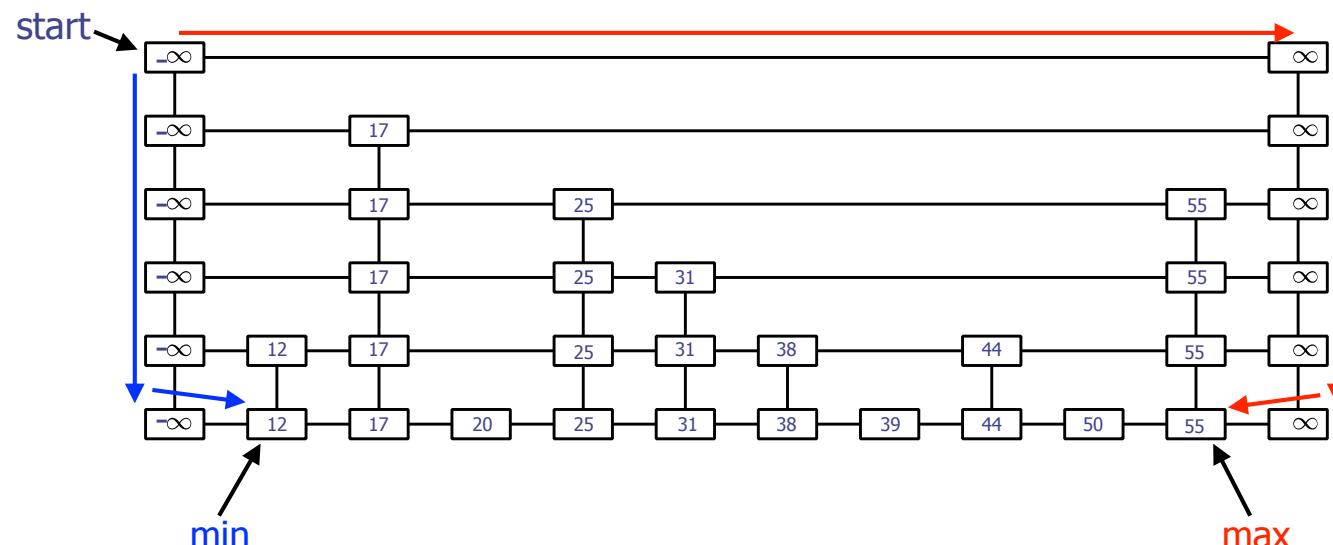
```

```

#retourne l'élément le plus petit de la skiplist
def Min( self ):
    #si la skiplist est vide, il n'existe pas
    if self._size == 0:
        return None
    #on prend le début de la skiplist
    tower = self._start
    #on descend jusqu'au niveau 0 par les sentinelles
    while not( tower._belo is None ):
        tower = tower._belo
    #l'élément le plus petit est le suivant de la sentinelle
    return tower._next._elem

#retourne l'élément le plus grand de la skiplist
def Max( self ):
    #si la skiplist est vide, il n'existe pas
    if self._size == 0:
        return False
    #on prend la sentinelle à droite du début de la skiplist
    tower = self._start._next
    #on descend jusqu'au niveau 0 par les sentinelles de droite
    while not( tower._belo is None ):
        tower = tower._belo
    #l'élément le plus grand est le précédent de la sentinelle
    return tower._prev._elem

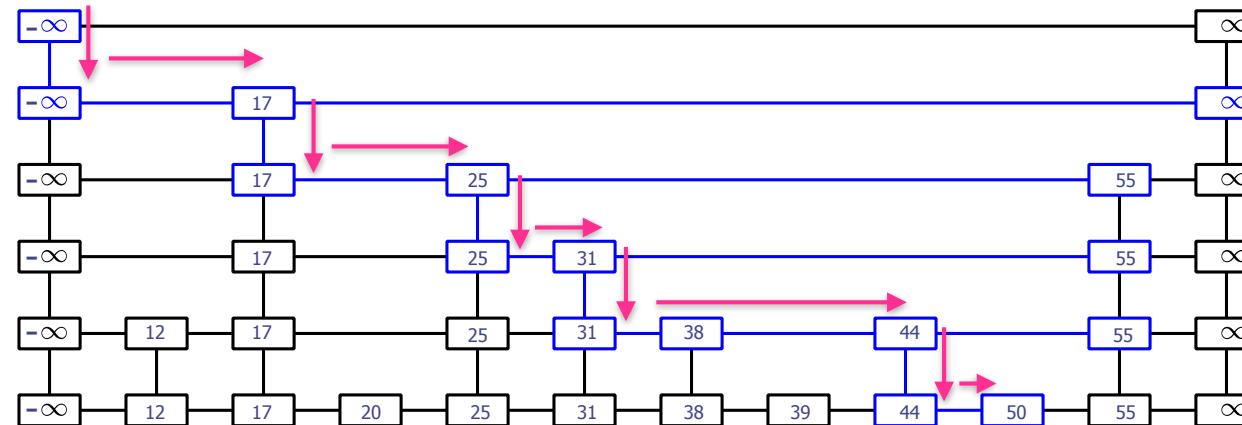
```



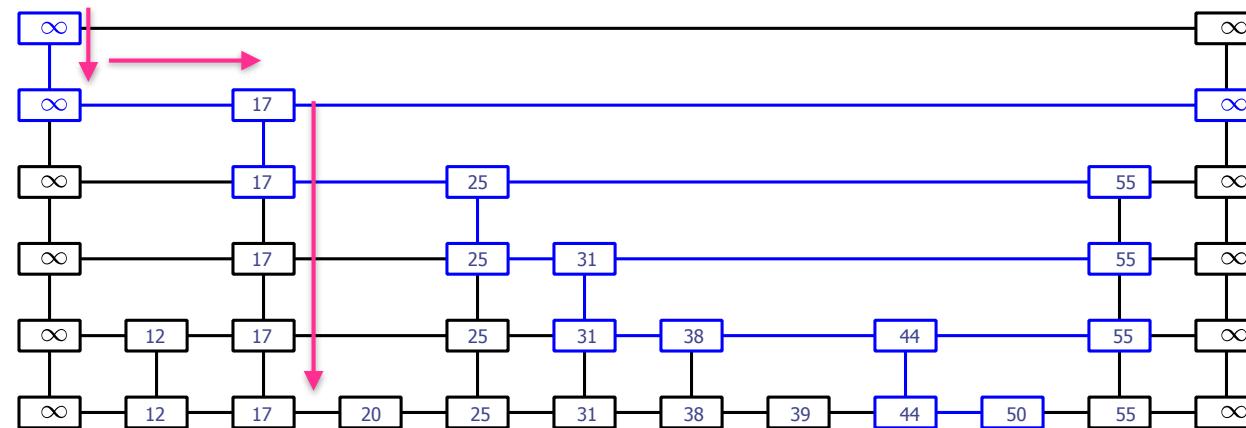
```

#recherche et retourne un élément, ou son suivant s'il n'existe pas
def SkipSearch( self, element ):
    #on prend le début de la skiplist
    p = self._start
    #tant que sur un niveau supérieur à 0
    while not( p._belo is None ):
        #on descend de niveau
        p = p._belo
        #tant que l'élément suivant est < que l'élément recherché
        while element >= p._next._elem:
            #on prend l'élément suivant
            p = p._next
    #s'arrête et retourne l'élément recherché ou son suivant
    return p

```



SkipSearch(50)



SkipSearch(17)

```

#insertion d'un nouvel élément
#retourne le noeud du plus haut niveau du nouvel élément s'il n'existe pas
#sinon, le noeud du niveau 0 de celui qui existait
def SkipInsert( self, element ):
    #on commence par le chercher
    p = self.SkipSearch( element )
    #on sauve le résultat de la recherche
    q = p
    #si l'élément existait
    if p._elem == element:
        #on doit modifier sa valeur à chaque niveau
        #imaginez des éléments du type ( clé, valeur )
        #on doit ajuster la valeur de tous les éléments de sa tour
        while( not p is None ):
            p._elem = element
            p = p._abov
        #dans ce cas, on a terminé, on retourne
        return q

    #comme l'élément n'existe pas, p pointe sur le noeud précédent
    #nous sommes au niveau 0, donc belo est None
    q = self.insertAfterAbove( p, None, element )

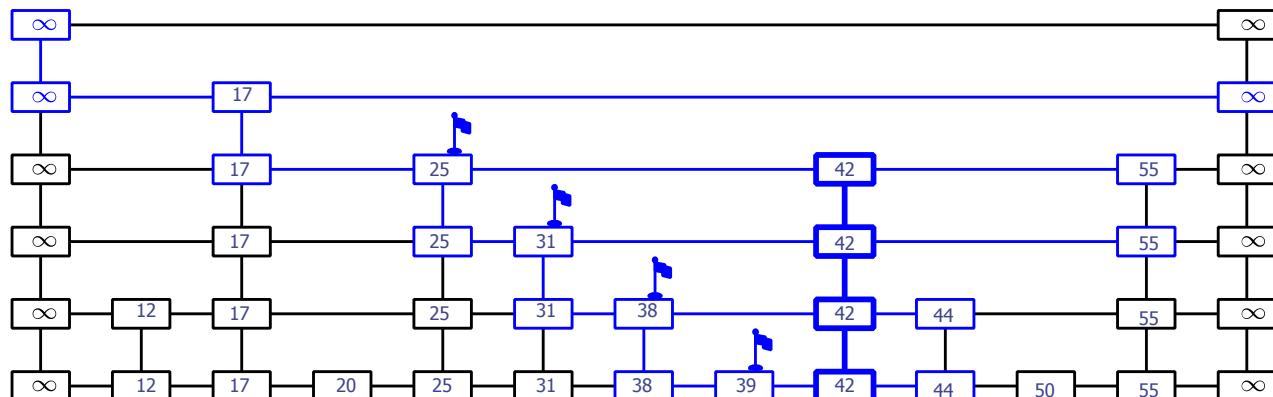
    #on a inséré un noeud au niveau 0 pour le nouvel élément
    #on doit maintenant monter sa tour en fonction de pile ou face
    i = 0
    coin_flip = self._coin.flip()
    while coin_flip == _FACE:
        i += 1 #i indique le niveau courant d'insertion
        #si on a atteint le niveau du haut
        #on doit ajouter un nouveau niveau, on appelle increaseHeight()
        if i >= self._height:
            self.increaseHeight()
        #on doit ajouter un noeud sur le niveau au-dessus
        #on cherche un noeud dans les précédents, débutant à p, qui permet de monter de niveau
        while p._abov is None:
            #on va voir le précédent
            p = p._prev
        #p pointe vers un noeud qui en possède un au-dessus, on le suit pour monter
        p = p._abov
        #q pointe vers le noeud inséré précédemment,
        #il correspond au noeud en dessous du noeud à insérer
        q = self.insertAfterAbove( p, q, element )
        #on relance le jeton
        coin_flip = self._coin.flip()

    #ici on a terminé la tour de l'élément à insérer
    #avant de sortir, on incrémenté de 1 la taille de la skipList
    self._size += 1
    #on retourne le dernier noeud inséré, celui du plus haut niveau ajouté
    return q

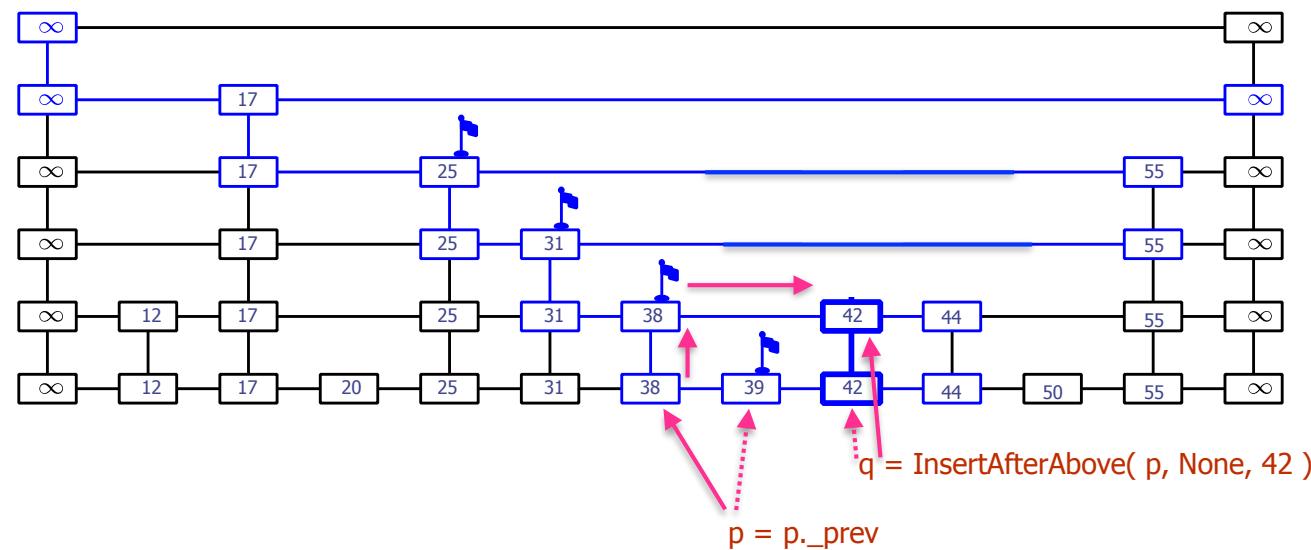
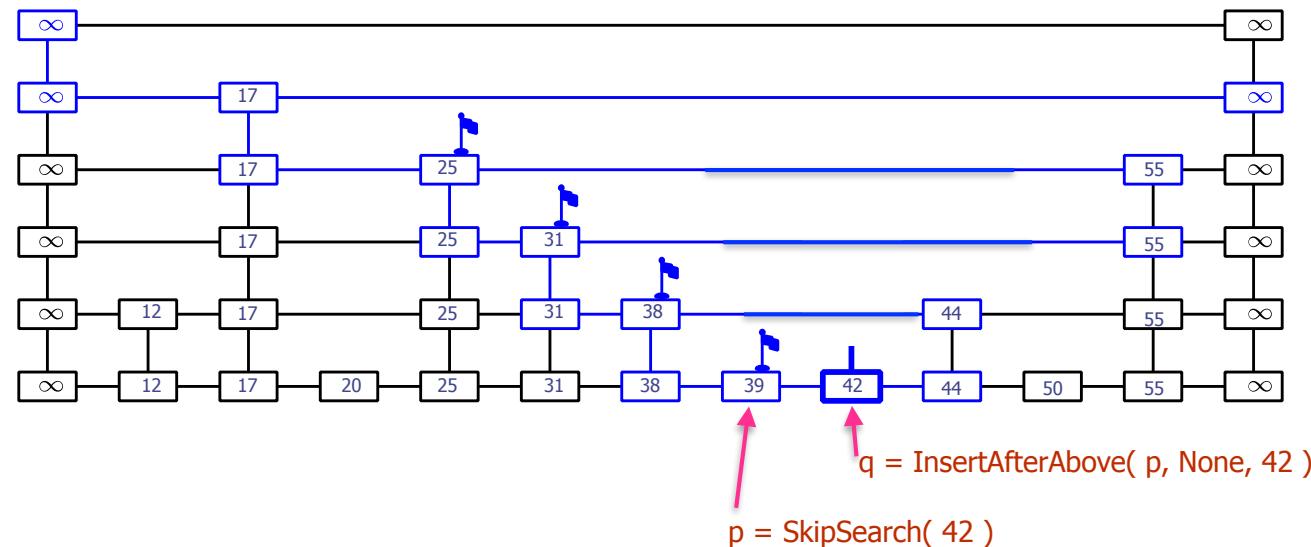
#augmente la hauteur de la skipList
def increaseHeight( self ):
    #on retient les anciennes sentinelles du plus haut niveau
    old_sentinel_l = self._start
    old_sentinel_r = self._start._next
    #on crée deux nouvelles sentinelles pour la nouveau niveau
    new_sentinel_l = SkipListNode( self._MIN_VALUE, None, None, old_sentinel_l, None )
    new_sentinel_r = SkipListNode( self._MAX_VALUE, new_sentinel_l, None, old_sentinel_r, None )
    new_sentinel_l._next = new_sentinel_r
    #on relie les pointeurs du dessus des vieilles sentinelles aux nouvelles
    old_sentinel_l._abov = new_sentinel_l
    old_sentinel_r._abov = new_sentinel_r
    #on augmente la variable qui indique la hauteur de la skipList
    self._height += 1
    #on ajuste le début de la skipList
    self._start = new_sentinel_l

#insère un noeud après le noeud p et au-dessus du noeud q
def insertAfterAbove( self, p, q, element ):
    #p est pour le précédent
    #q est pour celui au-dessous
    #arguments de SkipListNode : element, prev, next, belo, abov
    #abov toujours None car on insère toujours au niveau le plus haut
    #le nouveau noeud a p comme précédent, p._next comme suivant,
    #    q au-dessous et rien au-dessus
    newnode = SkipListNode( element, p, p._next, q, None )
    #on relie le précédent de l'ancien suivant, next._prev, au nouveau noeud
    p._next._prev = newnode
    #on relie le suivant du précédent, p._next, au nouveau noeud
    p._next = newnode
    #s'il y a un noeud en-dessous, q n'est pas à None, on le relie au nouveau noeud
    #which is abov it
    if not( q is None ):
        q._abov = newnode
    #on retourne le nouveau noeud
    return newnode

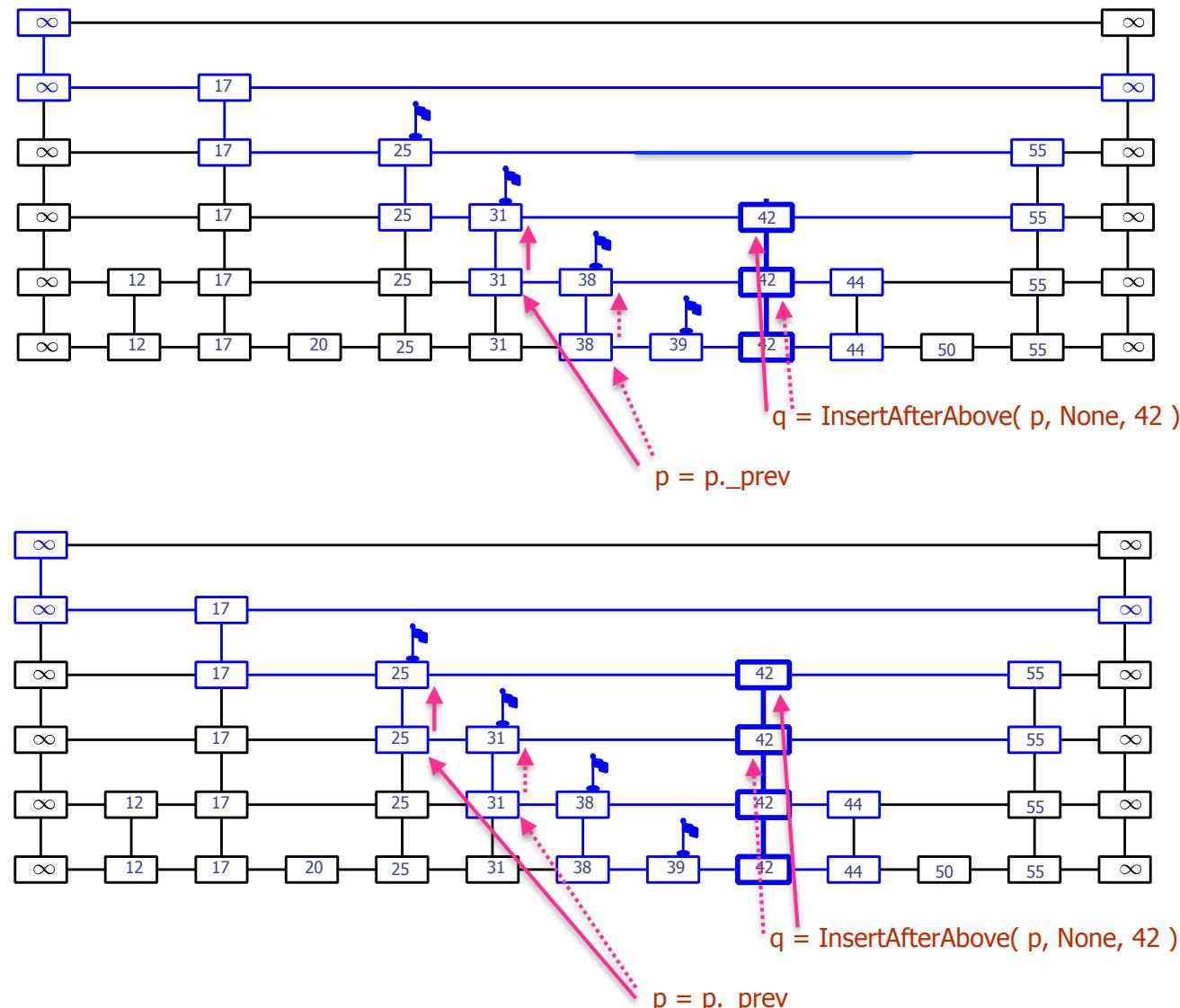
```



SkipInsert(42)



SkipInsert(42)

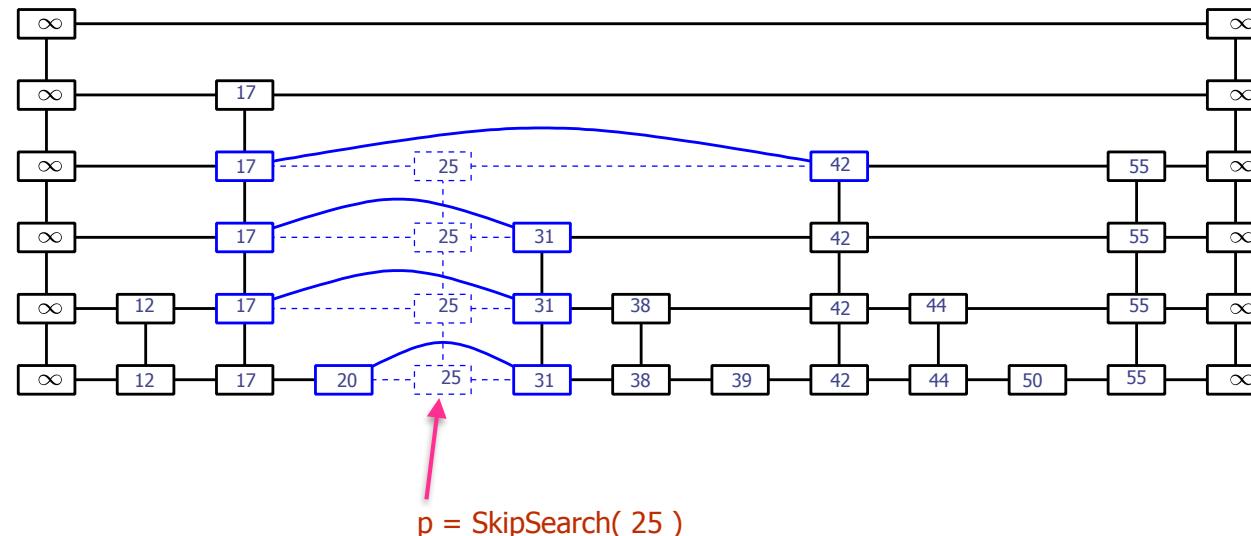


SkipInsert(42)

```

#supprime et retourne un élément, s'il existe
#sinon, retourne None
def SkipRemove( self, element ):
    #on commence par le chercher
    p = self.SkipSearch( element )
    #s'il existe
    if p._elem == element:
        #on retire le noeud de cet élément à chaque niveau
        tower = p
        while not( tower is None ):
            #comme la suppression dans une liste chaînée
            tower._prev._next = tower._next
            tower._next._prev = tower._prev
            #on monte de niveau
            tower = tower._above
        #on retourne
        return p
    return None

```



SkipRemove(25)

```

import sys
import time
import random
from Map import Map
from SkipList import SkipList

_MIN = -sys.maxsize - 1
_MAX = sys.maxsize

#Map implémenté avec une skipList
class SkipListMap( Map ):

    #on utilise une skipList avec des Items comme éléments dans les noeuds
    #on utilise les valeurs de +infini et -infini pour les sentinelles
    def __init__( self, _MIN_KEY = _MIN, _MAX_KEY = _MAX ):
        self._T = SkipList( Map._Item( _MIN_KEY, None ), Map._Item( _MAX_KEY, None ) )

    def __str__( self ):
        return str( self._T )

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

    #accès à l'élément k, par appel direct à SkipSearch
    def __getitem__( self, k ):
        p = self._T.SkipSearch( self._Item( k ) )
        if p._elem._key != k:
            raise KeyError( k )
        return p._elem._value

    #assignation de l'élément de clé k, appel direct à SkipInsert
    def __setitem__( self, k, v ):
        self._T.SkipInsert( self._Item( k, v ) )

    #suppression de l'élément de clé k, appel direct à SkipRemove
    def __delitem__( self, k ):
        p = self._T.SkipRemove( self._Item( k ) )
        if p is None:
            raise KeyError( k )
        return p._elem._value

    #itérateur sur les clé
    def __iter__( self ):
        for item in self._T:
            yield item._key

```

SortedListMap vs. *SkipListMap*

Data structure	Insertion (sec.)	Search (sec.)	Delete (sec.)
Sorted List	$O(n)$	$O(\log n)$	$O(n)$
50,000	0.6	0.4	0.5
500,000	15.0	5.6	11.9
1M	50.1	11.8	35.9
Skip List	$O(\log n)$	$O(\log n)$	$O(\log n)$
50,000	0.8	0.6	0.6
500,000	11.0	8.2	8.3
1M	26.2	17.7	17.6

Application de Map : Maxima Set

La vie est pleine de compromis!

Achetez une voiture rapide mais à bon prix ...

Nous pouvons modéliser un problème de compromis en utilisant des paires (clé, valeur), c'est-à-dire (coût, vitesse) pour des voitures.

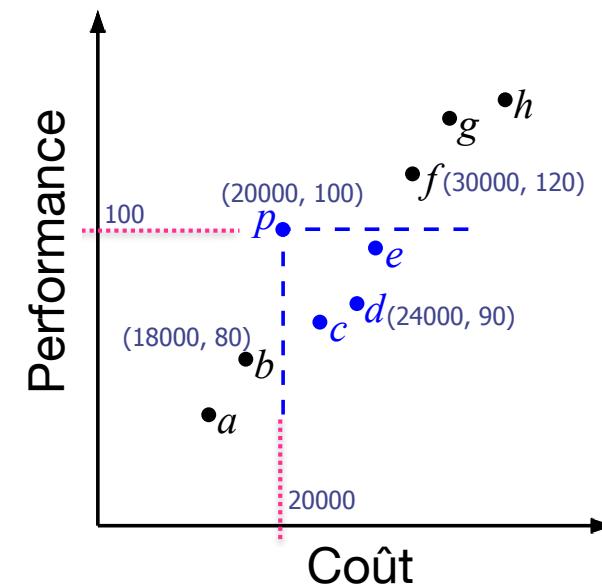
Certaines voitures sont mieux que d'autres:

(20000, 100) est mieux (moins chère et plus rapide) que (30000, 90)

Cependant, comment décider entre :
(20000, 100) et (30000, 120), (moins chère mais aussi moins rapide)

Étant donné les points **a** à **h**, lorsque le point **p** est trouvé, quels points peuvent être éliminés de la liste des voitures à considérer ?

Clairement, les voitures correspondantes aux points **c**, **d** et **e** puisqu'elles sont plus chères et moins performantes que **p**.



Maxima Set

```

from SortedListMap import SortedListMap
import time
import random

#classe pour un Maxima Set implantée avec SortedListMap
class SortedListMaximaSet():

    #constructeur
    def __init__( self ):
        self._M = SortedListMap()

    #prettyprint
    def __str__( self ):
        return str( self._M )

    #ajouter un point au Maxima Set
    def add( self, x, y ):
        #ne pas considérer (x,y) s'il existe un point
        #équivalent ou meilleur
        other = self._M.find_le( x )
        if other is not None and other[1] >= y:
            return
        #sinon, on l'ajoute à l'ensemble
        self._M[x] = y
        #éliminer les points dont le 1er critère est > x
        #et qui ne peuvent pas être justifiés un 2ème critère > y
        #on prend le premier point défavorisé par x
        other = self._M.find_gt( x )
        #on l'élimine s'il est également défavorisé par y
        while other is not None and ( other[1] <= y ):
            del self._M[other[0]]
            other = self._M.find_gt( x )

    SortedListMaximaSet unit testing...
    Add 4000000 (x,y) pairs, x in [9000..300000], y in [50...300] in 18.9579598903656 seconds.
    {(9000, 287)(9001, 290)(9004, 295)(9012, 296)(9019, 297)(9032, 300)} size = 6

```

```

SkipListMap import SkipListMap
rt time
rt random

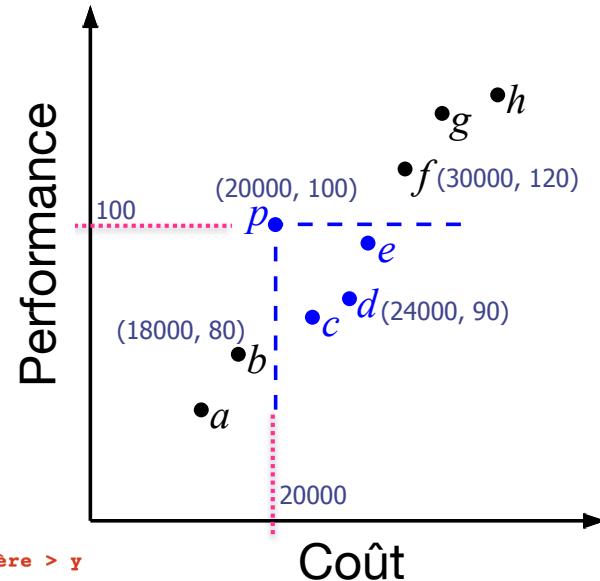
#classe pour un Maxima Set implémenté avec SkipListMap
class SkipListMaximaSet():

    #constructeur
    def __init__( self ):
        self._M = SkipListMap()

    #prettyprint
    def __str__( self ):
        return str( self._M )

    def add( self, x, y ):
        #ne pas considérer (x,y) s'il existe un point
        #équivalent ou meilleur
        other = self._M.find_le( x )
        if ( other is not None ) and ( other[1] >= y ):
            return
        #sinon, l'ajouter
        self._M[x] = y
        #éliminer les points dont le 1er critère est > x
        #et qui ne peuvent pas être justifiés un 2ème critère > y
        #on prend le premier point défavorisé par x
        other = self._M.find_gt( x )
        #on l'élimine s'il est également défavorisé par y
        while ( other is not None ) and ( other[1] <= y ):
            del self._M[other[0]]
            other = self._M.find_gt( x )

```



```

SkipListMaximaSet unit testing...
Add 4000000 (x,y) pairs, x in [9000..300000], y in [50...300] in 27.241148948669434 seconds.
level 0 [<9000,287>,<9001,290>,<9004,295>,<9012,296>,<9019,297>,<9032,300>]

```

```

#trouve et retourne l'élément avec la plus petit clé
#si la Map n'est pas vide
def find_min( self ):
    if len( self._T ) > 0:
        return ( self._T[0]._key, self._T[0]._value )
    else:
        return None

#trouve et retourne l'élément avec la plus grande clé
#si la Map n'est pas vide
def find_max( self ):
    if len( self._T ) > 0:
        return ( self._T[-1]._key, self._T[-1]._value )
    else:
        return None

#trouve et retourne l'élément avec la première clé >= k
#si il existe
def find_ge( self, k ):
    j = self._find_index( k, 0, len( self._T ) - 1 )
    if j < len( self._T ):
        return ( self._T[j]._key, self._T[j]._value )
    else:
        #j >= len(T) => non trouvé
        return None

#trouve et retourne l'élément avec la première clé <= k
#si il existe
def find_le( self, k ):
    #si la Map est vide, il n'existe pas
    if( len( self._T ) ) == 0:
        return None
    #sinon, on trouve j tel que clé de T[j] >= k
    j = self._find_index( k, 0, len( self._T ) - 1 )
    #si k est > que la plus grande clé
    #on retourne le dernier élément (celui de la plus grande clé)
    if j >= len( self._T ):
        return ( self._T[j-1]._key, self._T[j-1]._value )
    #si on a trouvé un clé à l'intérieur de la liste mais
    #que cette clé ne correspond pas à k => elle est > k
    #alors on retourne l'élément précédent
    if self._T[j]._key != k:
        j -= 1
    #si cet élément n'était pas l'élément de la plus petite clé
    #on le retourne
    if j >= 0:
        return ( self._T[j]._key, self._T[j]._value )
    else:
        #sinon, il n'existe pas d'élément avec une clé <= k
        return None

#trouve et retourne l'élément avec la première clé < k
#si il existe
def find_lt( self, k ):
    #si la Map est vide, il n'existe pas
    if( len( self._T ) ) == 0:
        return None
    #sinon, on trouve j tel que clé de T[j] >= k
    j = self._find_index( k, 0, len( self._T ) - 1 )
    #si j < len(T) et que la clé de T[j] > k
    #alors on prend l'élément précédent
    #si la clé de T[j] == k, on a un élément dont la clé est égale
    #et on veut l'élément de clé plus petite
    #si la clé de T[j] n'est pas égale à k, elle est la première > k
    #et on veut l'élément de clé plus petite
    if j < len( self._T ) and self._T[j]._key > k :
        j -= 1
    #si ce n'était pas la plus petite clé de la Map
    #alors on le retourne
    if j >= 0:
        return ( self._T[j]._key, self._T[j]._value )
    else:
        #sinon, l'élément de clé k est le plus petit de la Map
        #et, donc, il n'en existe pas de plus petit
        return None

#itérateur d'éléments de la Map avec start <= clé < stop
def find_range( self, start, stop ):
    #si start est None, on commence au début de la Map
    if start is None:
        j = 0
    #sinon, on trouve l'index de l'élément de clé >= start
    else:
        j = self._find_index( start, 0, len( self._T ) - 1 )
    #on itère sur les éléments j <= clé < stop
    #on teste j < len(T), puisque j pourrait correspondre
    #à une clé > la plus grande clé dans la Map
    while j < len( self._T ) and (stop is None or self._T[j]._key < stop):
        yield ( self._T[j]._key, self._T[j]._value )
        j += 1

```

```

#trouve et retourne l'élément avec la plus petit clé
#si la Map n'est pas vide
def find_min( self ):
    if len( self._T ) > 0:
        theItem = self._T.Min()
        return ( theItem._key, theItem._value )
    else:
        return None

#trouve et retourne l'élément avec la plus grande clé
#si la Map n'est pas vide
def find_max( self ):
    if len( self._T ) > 0:
        theItem = self._T.Max()
        return ( theItem._key, theItem._value )
    else:
        return None

#retourne ( key, value ), key >= k
def find_ge( self, k ):
    #SkipSearch s'arrête sur key <= k
    p = self._T.SkipSearch( Map._Item( k ) )
    #si on arrête sur la sentinelle à droite
    #il n'existe pas d'élément dont la clé est >= k
    if p._next is None:
        return None
    #sinon, si l'élément de clé k n'existe pas,
    #le plus grand est le suivant
    if p._elem._key < k:
        p = p._next
    return ( p._elem._key, p._elem._value )

#retourne ( key, value ), key <= k
def find_le( self, k ):
    #SkipSearch s'arrête sur key <= k
    p = self._T.SkipSearch( Map._Item( k ) )
    #si on arrête sur la sentinelle à gauche
    #il n'existe pas d'élément dont la clé est <= k
    if p._prev is None:
        return None
    #sinon, on s'est arrêté dessus
    return ( p._elem._key, p._elem._value )

#retourne ( key, value ), key > k
def find_gt( self, k ):
    #SkipSearch s'arrête sur key <= k
    p = self._T.SkipSearch( Map._Item( k ) )
    #si on arrête sur le dernier élément ou la sentinelle à droite
    #il n'existe pas d'élément dont la clé est > k
    if p._next is None or p._next._next is None:
        return None
    #le plus grand est le suivant
    #s'il l'élément de clé k existe, le > est le suivant
    #s'il n'existe pas, le > est aussi le suivant
    #puisque'on s'est arrêté sur le premier <
    p = p._next
    return ( p._elem._key, p._elem._value )

#retourne ( key, value ), key < k
def find_lt( self, k ):
    #SkipSearch s'arrête sur key <= k
    p = self._T.SkipSearch( Map._Item( k ) )
    #si on arrête sur le premier élément ou la sentinelle à gauche
    #il n'existe pas d'élément dont la clé est < k
    if p._prev is None or p._prev._prev is None:
        return None
    #sinon, soit on s'est arrêté sur l'élément de clé k
    #et on doit prendre son précédent (le premier < devant k)
    #ou si l'élément de clé k n'existe pas, on est sur le
    #premier élément de clé < k
    if p._elem._key == k:
        p = p._prev
    return ( p._elem._key, p._elem._value )

#itérateur des éléments de clé entre start jusqu'au précédent stop
def find_range( self, start, stop ):
    #si start est None, on prend start = Min
    if start is None:
        start, v = self.find_min()
    #on cherche l'élément de clé start
    #on arrête sur ce dernier ou sur le premier élément <
    p = self._T.SkipSearch( Map._Item( start ) )
    #si l'élément start n'existe pas, on est sur l'élément <
    #donc, on prend l'élément suivant
    if p._elem._key < start:
        p = p._next
    #tant qu'il y a des éléments dont la clé est < stop
    #on les rapporte dans l'itérateur
    while not( p._next is None ) and ( p._elem._key < stop ):
        yield ( p._elem._key, p._elem._value )
        p = p._next

```