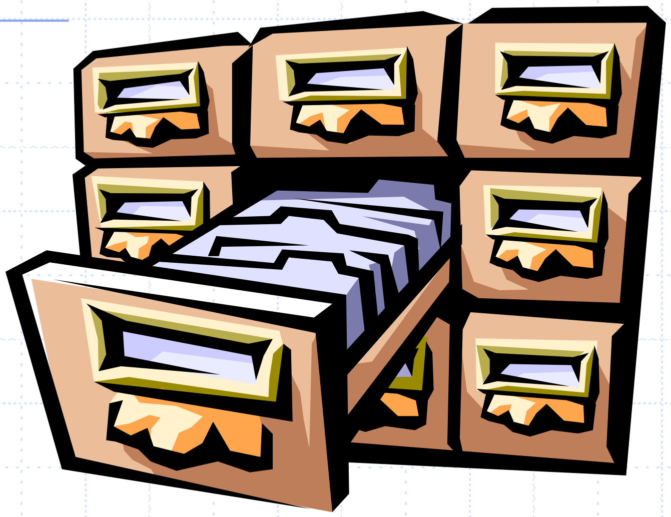
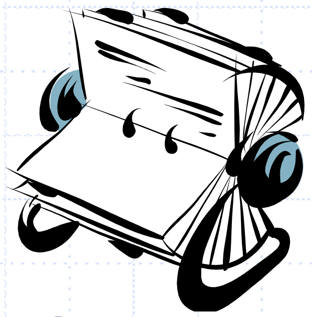
Maps and Dictionaries



Maps



- A **map** is a searchable collection of items that are key-value pairs
- The main operations of a map are for searching, inserting, and deleting items
- Multiple items with the same key are not allowed
- Applications:
  - address book
  - student-record database

Dictionaries

- Python's **dict** class is arguably the most significant data structure in the language.
  - It represents an abstraction known as a **dictionary** in which unique **keys** are mapped to associated **values**.
- Here, we use the term "dictionary" when specifically discussing Python's dict class, and the term "map" when discussing the more general notion of the abstract data type.

The Map ADT (Using **dict** Syntax)

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.

More Map Operations

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).

A Few More Map Operations

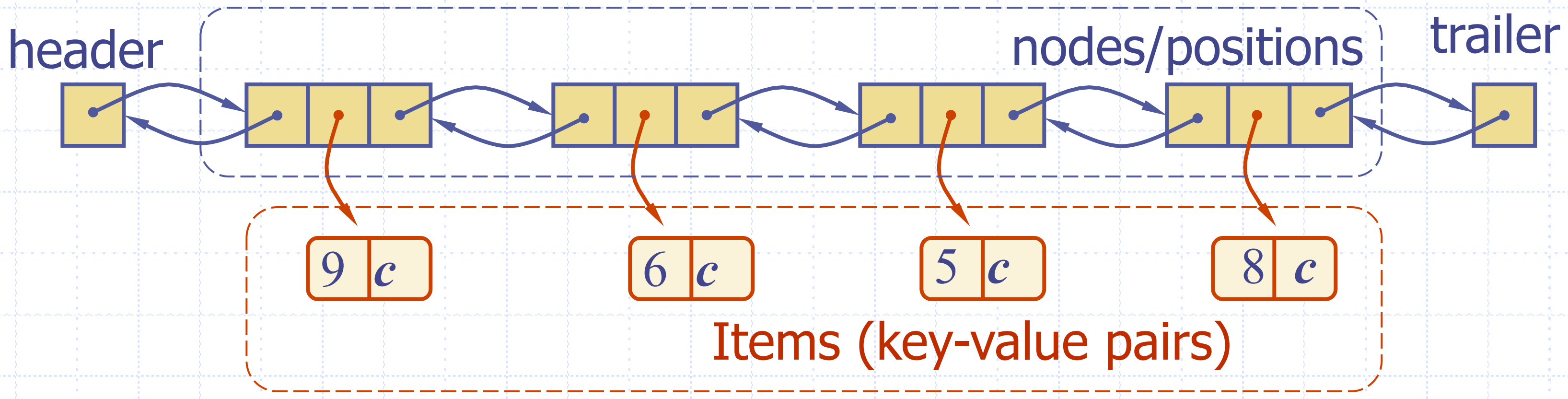
M.popitem():	Remove an arbitrary key-value pair from the map, and return a (k,v) tuple representing the removed pair. If map is empty, raise a KeyError.
M.clear():	Remove all key-value pairs from the map.
M.keys():	Return a set-like view of all keys of M.
M.values():	Return a set-like view of all values 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.

Example

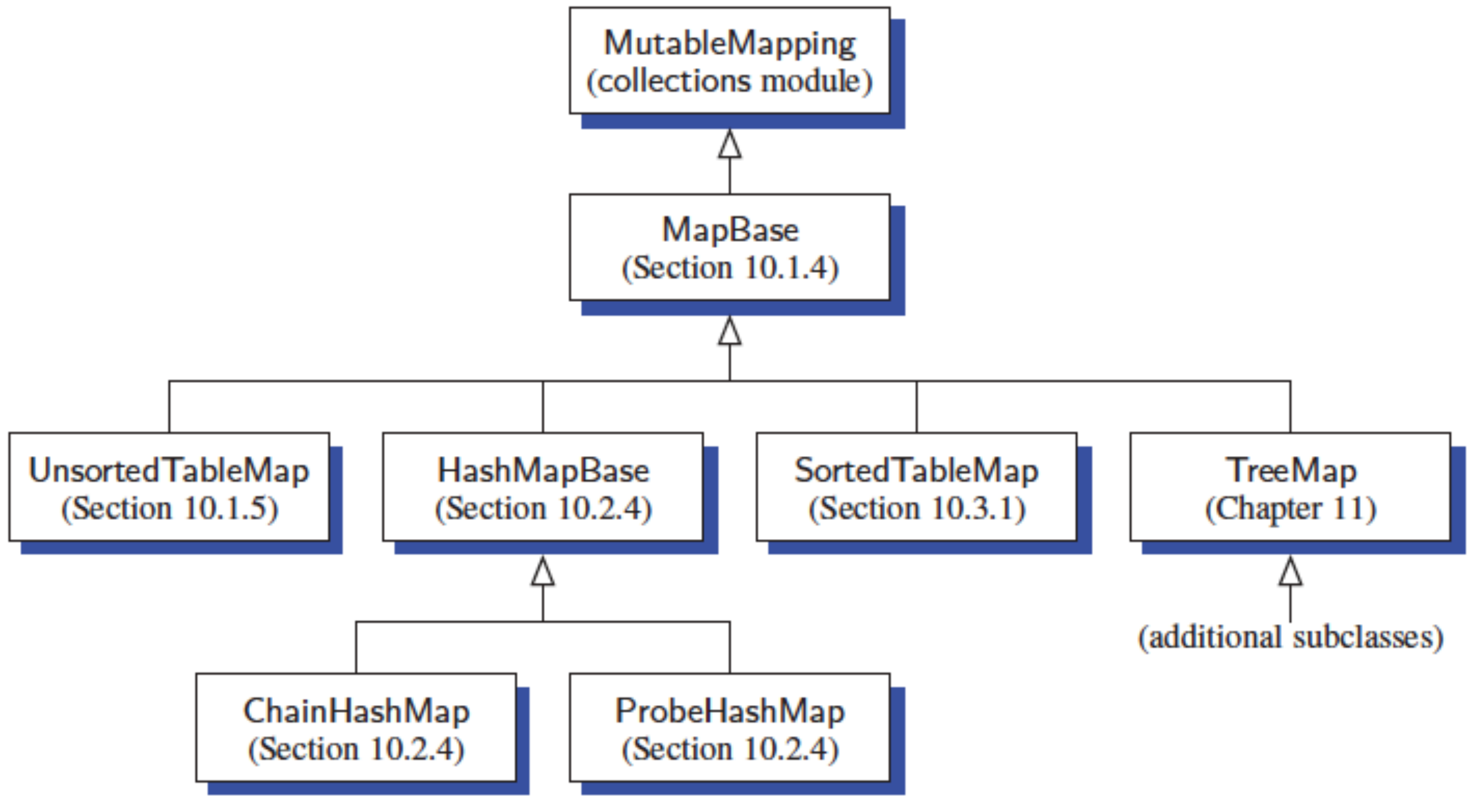
Operation	Return Value	Map
len(M)	0	{ }
M['K'] = 2	–	{'K': 2}
M['B'] = 4	–	{'K': 2, 'B': 4}
M['U'] = 2	–	{'K': 2, 'B': 4, 'U': 2}
M['V'] = 8	–	{'K': 2, 'B': 4, 'U': 2, 'V': 8}
M['K'] = 9	–	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['B']	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M['X']	KeyError	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F')	None	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('F', 5)	5	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
M.get('K', 5)	9	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
len(M)	4	{'K': 9, 'B': 4, 'U': 2, 'V': 8}
del M['V']	–	{'K': 9, 'B': 4, 'U': 2}
M.pop('K')	9	{'B': 4, 'U': 2}
M.keys()	'B', 'U'	{'B': 4, 'U': 2}
M.values()	4, 2	{'B': 4, 'U': 2}
M.items()	('B', 4), ('U', 2)	{'B': 4, 'U': 2}
M.setdefault('B', 1)	4	{'B': 4, 'U': 2}
M.setdefault('A', 1)	1	{'A': 1, 'B': 4, 'U': 2}
M.popitem()	('B', 4)	{'A': 1, 'U': 2}

A Simple List-Based Map

- We can efficiently implement a map using an unsorted list
  - We store the items of the map in a list S (based on a doubly-linked list), in arbitrary order



Our MapBase Class



The Map Abstract Class

```
import collections
class Map( collections.MutableMapping ):

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

        def __init__( self, k, v ):
            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
```

An Unsorted List Implementation

```
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
```

Performance of a List-Based Map

- Performance:
  - Inserting an item takes $O(1)$ time since we can insert the new item at the beginning or at the end of the unsorted list
    - ***HOWEVER: since inserting in a map invokes first a search of the key, the insertion operation takes O(n)***
  - Searching for or removing an item takes $O(n)$ time, since in the worst case (the item is not found) we traverse the entire listto look for an item with the given key
- The unsorted list implementation is effective only for maps of small size or for maps in which insertions are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

Unsorted List Implementation (Performances 50,000 keys)

```
djmaya2-iro-61:Maps major$ python UnsortedListMap.py
UnsortedListMap unit testing...
Insertion of 50000 keys in  46.3809711933136 seconds.
Access to 50000 keys in  101.96852087974548 seconds.
End of testing.
djmaya2-iro-61:Maps major$
```

A Sorted List Implementation…

```
class SortedListMap( Map ):

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

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

    def __getitem__( self, k ):
        j = self._find_index( k, 0, len( self._T ) - 1 )
        if j == len( self._T ) or self._T[j]._key != k:
            return False
        return self._T[j]._value

    def __setitem__( self, k, v ):
        j = self._find_index( k, 0, len( self._T ) - 1 )
        if j < len( self._T ) and self._T[j]._key == k:
            self._T[j]._value = v
        else:
            self._T.insert( j, self._Item( k, v ) )

    def __delitem__( self, k ):
        j = self._find_index( k, 0, len( self._T ) - 1 )
        if j == len( self._T ) or self._T[j]._key != k:
            return False
        self._T.pop( j )
```

A Sorted List Implementation...

```
def _find_index( self, k, low, high ):
    """
     Binary search
     Return the index of the leftmost item with key >= k
         return j such that:
             T[low:j] have key < k
             T[j:high+1] have key >= k
    """
    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 )
```

A Sorted List Implementation…

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

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

def find_min( self ):
     if len( self._T ) > 0:
         return (self._T[0]._key,self._T[0]._value)
     else:
         return None

def find_max( self ):
    if len( self._T ) > 0:
         return (self._T[-1]._key,self._T[-1]._value)
    else:
         return None
```

A Sorted List Implementation…

```
def find_ge( self, k ):
    #return (key,value) where key >= 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:
        return None

def find_le( self, k ):
    #return (key,value) where key <= k
    j = self._find_index( k, 0, len( self._T ) - 1 )
    if j > 0:
        return (self._T[j-1]._key,self._T[j-1]._value)
    else:
        return None

def find_gt( self, k ):
    #return (key,value) where key > k
    j = self._find_index( k, 0, len( self._T ) - 1 )
    if j < len( self._T ) and self._T[j]._key == k:
        j += 1
    if j < len( self._T ):
        return (self._T[j]._key,self._T[j]._value)
    else:
        return None

def find_lt( self, k ):
    #return (key,value) where key < k
    j = self._find_index( k, 0, len( self._T ) - 1 )
    if j > 0:
        return (self._T[j-1]._key,self._T[j-1]._value)
    else:
        return None
```

Performance of a Sorted List Map

- Performance:
  - Inserting and removing an item takes $O(n)$ time since we need to shift the elements, and this despite the fast binary search to find a key in O(log n) time.
- The sorted list implementation is effective for maps of large size and for maps in which searching is the most common operation.

A Sorted List Implementation

```
def find_range( self, start, stop ):
    #iterate (key,value) where start <= key < stop
    if start is None:
        j = 0
    else:
        j = self._find_index( start, 0, len( self._T ) - 1 )
    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
```

Sorted List Implementation (Performances 50,000 keys)

```
djmaya2-iro-61:Maps major$ python SortedListMap.py
UnsortedListMap unit testing...
Insertion of 50000 keys in  0.864854097366333 seconds.
Access to 50000 keys in  1.08042311668396 seconds.
End of testing.
djmaya2-iro-61:Maps major$
```

500,000 keys:

```
djmaya2-iro-61:Maps major$ python SortedListMap.py
UnsortedListMap unit testing...
Insertion of 500000 keys in  25.5028610229492 seconds.
Access to 500000 keys in  12.1945929527271 seconds.
End of testing.
djmaya2-iro-61:Maps major$
```

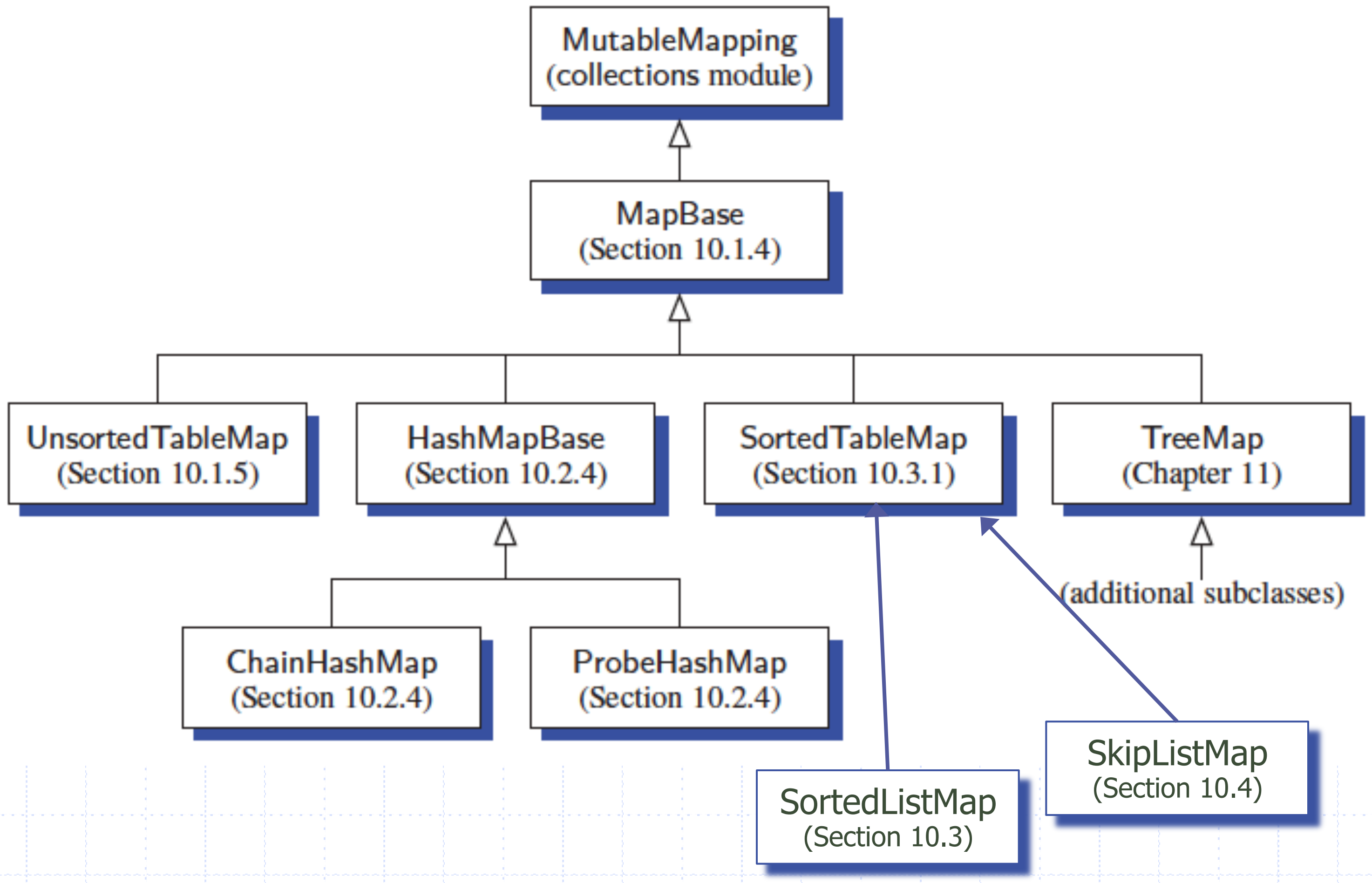
Sorted Implementation

A sorted list implementation is nice!

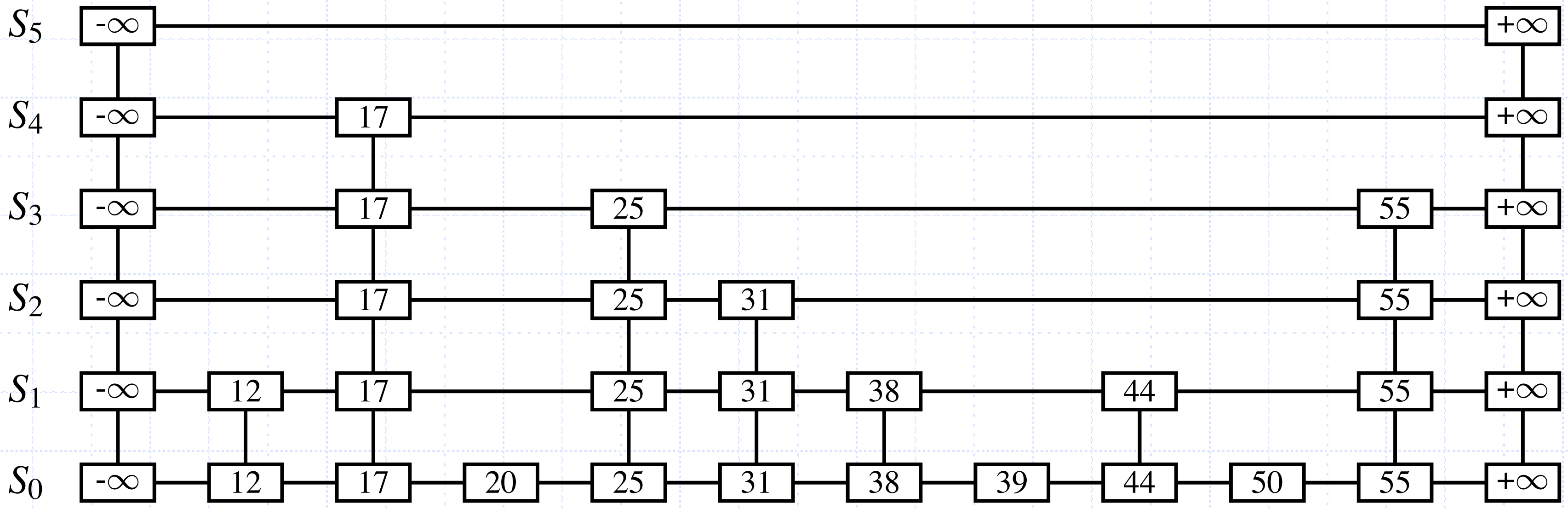
- The binary search in O(log n)
- However, the update methods are in O(n) time

Can we do better?

SkipLists implement the SortedMap



SkipList of 10 elements



SkipListNode

```
class SkipListNode:

    __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

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

SkipList Constructor

```
from SkipListNode import SkipListNode
from Coin import Coin

_FACE       = True
_TAILS      = False

class SkipList():

    def __init__( self, _MIN_VALUE = -999999999, _MAX_VALUE = 999999999 ):
        self._MIN_VALUE = _MIN_VALUE
        self._MAX_VALUE = _MAX_VALUE
        self._coin = Coin()
        self._height = 1
        self._count = 0
        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
        self._start = sentinel_ul
```

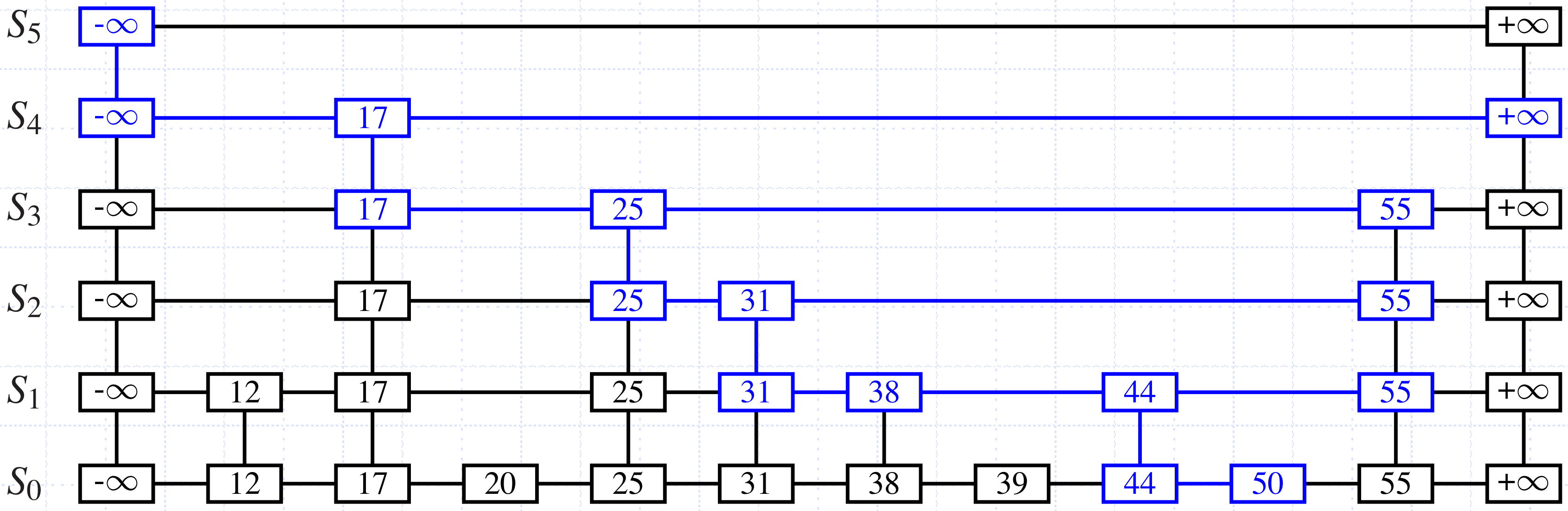
SkipList len, str, and iter

```
#return the number of elements in List
def __len__( self ):
      return self._count

#convert a SkipList into a string:
# elements listed between brackets
# separated by commas.
def __str__( self ):
    tower = self._start
      pp = "SkipList of height " + str( self._height ) + ":\n"
      for level in range( self._height, -1, -1 ):
        p = tower
        pp += "level " + str( level ) + " [" + str( p._elem )
        p = p._next
        while not( p is None ):
                pp +=  "," + str( p._elem )
                p = p._next
        tower = tower._belo
        pp += "]\n"
      return pp

def __iter__( self ):
      tower = self._start
      while not( tower._belo is None ):
        tower = tower._belo
      tower = tower._next
      while not( tower._next is None ):
                yield tower._elem
                tower = tower._next
```

SkipList Min & Max

```
def Min( self ):
    if self._count == 0:
        return False
    tower = self._start
    while not( tower._belo is None ):
        tower = tower._belo
    return tower._next._elem

def Max( self ):
    if self._count == 0:
        return False
    tower = self._start
    while not( tower._belo is None ):
        tower = tower._belo
    tower = tower._next
    while not( tower._next is None ):
            tower = tower._next
    return tower._prev._elem()
```

SkipSearch…

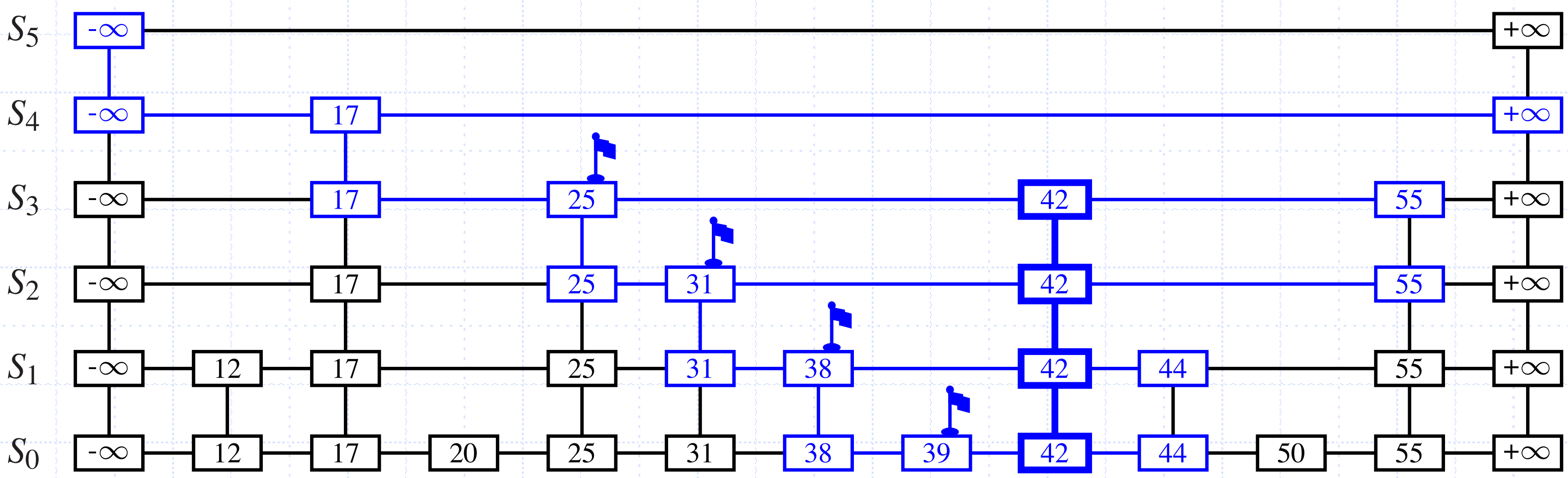


S_5 -∞ +∞
S_4 -∞ 17 +∞
S_3 -∞ 17 25 55 +∞
S_2 -∞ 17 25 31 55 +∞
S_1 -∞ 12 17 25 31 38 44 55 +∞
S_0 -∞ 12 17 20 25 31 38 39 44 50 55 +∞

SkipSearch

```
#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
```

SkipInsert…



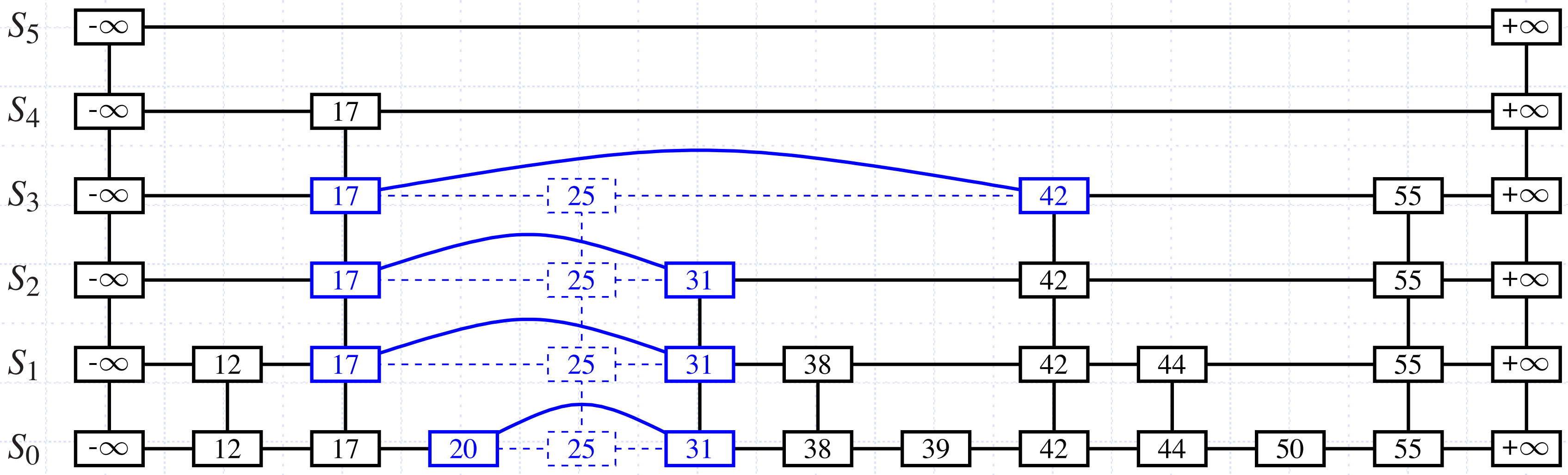
SkipInsert...

```
#insert element
def SkipInsert( self, element ):
    p = self.SkipSearch( element )
    if p._elem == element:
        p._elem = element
        return p
    q = self.insertAfterAbove( p, None, element )
    i = 0
    coin_flip = self._coin.flip()
    while coin_flip == _FACE:
        i += 1
        if i >= self._height:
            self.increaseHeight()
        while p._abov is None:
            p = p._prev
        p = p._abov
        q = self.insertAfterAbove( p, q, element )
        coin_flip = self._coin.flip()
    self._count += 1
    return q
```

SkipInsert

```
def increaseHeight( self ):
    old_sentinel_l = self._start
    old_sentinel_r = self._start._next
    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
    old_sentinel_l._abov = new_sentinel_l
    old_sentinel_r._abov = new_sentinel_r
    self._height += 1
      self._start = new_sentinel_l

def insertAfterAbove( self, p, q, element ):
    newnode = SkipListNode( element, p, p._next, q, None )
    p._next._prev = newnode
    p._next = newnode
    if not( q is None ):
        q._abov = newnode
    return newnode
```

SkipRemove…



SkipRemove

```
#remove element
def SkipRemove( self, element ):
    p = self.SkipSearch( element )
    if p._elem == element:
        tower = p
        while not( tower is None ):
            tower._prev._next = tower._next
            tower._next._prev = tower._prev
            tower = tower._abov
        return p
    return False
```

SkipListMap...

```
class SkipListMap( Map ):

    def __init__( self, _MIN_VALUE, _MAX_VALUE ):
        self._T = SkipList( Map._Item( _MIN_VALUE, None ), Map._Item( _MAX_VALUE, None ) )

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

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

    def __getitem__( self, k ):
        p = self._T.SkipSearch( self._Item( k ) )
        if p._elem._key != k:
            return False
        return p._elem._value

    def __setitem__( self, k, v ):
        self._T.SkipInsert( self._Item( k, v ) )

    def __delitem__( self, k ):
        p = self._T.SkipRemove( self._Item( k ) )
        if p is None:
            return False
        return p._elem._value
```

SkipListMap...

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

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

def pop( self, k ):
    p = self._T.SkipRemove( self._Item( k ) )
    if p is None:
        return False
    return p._elem._value

def find_min( self ):
    if len( self._T ) > 0:
        theItem = self._T.Min()
        return (theItem._key, theItem._value)
    else:
        return None

def find_max( self ):
    if len( self._T ) > 0:
        theItem = self._T.Max()
        return (theItem._key, theItem._value)
    else:
        return None
```

SkipListMap…

```
def find_ge( self, k ):
    #return (key,value) where key >= k
    p = self._T.SkipSearch( Map._Item( k ) )
    if p._next is None:
        return None
    if p._elem._key < k:
        p = p._next
    return (p._elem._key,p._elem._value)

def find_le( self, k ):
    #return (key,value) where key <= k
    p = self._T.SkipSearch( Map._Item( k ) )
    if p._prev is None:
        return None
    return (p._elem._key,p._elem._value)

def find_gt( self, k ):
    #return (key,value) where key > k
    p = self._T.SkipSearch( Map._Item( k ) )
    if p._next is None or p._next._next is None:
        return None
    p = p._next
    return (p._elem._key,p._elem._value)

def find_lt( self, k ):
    #return (key,value) where key < k
    p = self._T.SkipSearch( Map._Item( k ) )
    if p._prev is None or p._prev._prev is None:
        return None
    if p._elem._key == k:
        p = p._prev
    return (p._elem._key,p._elem._value)
```

SkipListMap

```
def find_range( self, start, stop ):
        #iterate (key,value) where start <= key < stop
        if start is None:
            start = self._T.Min()
        p = self._T.SkipSearch( Map._Item( start ) )
        if p._next is None:
            return None
        if p._elem._key < start:
            p = p._next
        while not( p._next is None ) and ( p._elem._key < stop ):
                yield (p._elem._key,p._elem._value)
                p = p._next
```

Sort vs. Skip List Implementation (Performances)

Data structure	Insertion (sec.)	Search (sec.)
Sorted List		
50,000	0.86	1.08
500,000	25.50	12.19
Skip List		
50,000	1.36	1.53
500,000	17.09	18.30

Maxima Set

Life is full of trade-offs!
Buy a fast but cheap car…

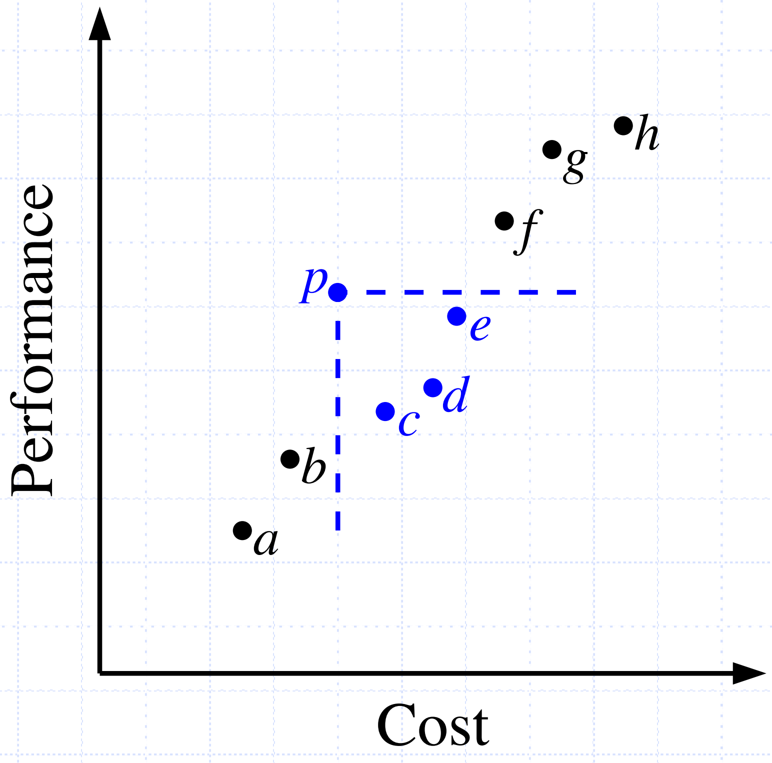
We can model a trade-off problem using (key, value) pairs, i.e. (cost, speed) for cars.

Some cars are strickly better than others:

(20000, 100) is better than (30000, 90)

However, we can't decide for:

(20000, 100) and (30000, 120)



SortedListMaximaSets

```
class SortedListMaximaSet():

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

    def best( self, x ):
        #return (X,Y) with cost <= c
        return self._M.find_le( x )

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

    def add( self, x, y ):
        other = self._M.find_le( x )
        if other is not None and other[1] >= y:
            return
        self._M[x] = y

         other = self._M.find_gt( x )
         while other is not None and other[1] <= y:
            del self._M[other[0]]
            other = self._M.find_gt( x )
```

SkipListMaximaSets

```
class SkipListMaximaSet():

    def __init__( self ):
        self._M = SkipListMap( -999999, 999999 )

    def best( self, x ):
        #return (X,Y) with cost <= c
        return self._M.find_le( x )

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

    def add( self, x, y ):
        other = self._M.find_le( x )
        if ( other is not None ) and ( other[1] >= y ):
            return
        self._M[x] = y

        other = self._M.find_gt( x )
        while ( other is not None ) and ( other[1] <= y ):
            del self._M[other[0]]
            other = self._M.find_gt( x )
```

Sorted vs. Skip List Maxima Sets (Performances 2,000,000 points)

```
djmaya2-iro-61:Maps major$ python SortedListMaximaSet.py
SortedListMaximaSet unit testing...
python SkipListAdd 2000000 XY pairs in  16.913034200668335 seconds.
{(15000, 185)(15004, 197)(15032, 200)}
End of testing.
djmaya2-iro-61:Maps major$ python SkipListMaximaSet.py
SkipListMaximaSet unit testing...
Add 2000000 XY pairs in  27.957228183746338 seconds.
SkipList of height 11:
level 11 [<-999999,None>,<999999,None>]
level 10 [<-999999,None>,<999999,None>]
level 9 [<-999999,None>,<999999,None>]
level 8 [<-999999,None>,<999999,None>]
level 7 [<-999999,None>,<999999,None>]
level 6 [<-999999,None>,<999999,None>]
level 5 [<-999999,None>,<999999,None>]
level 4 [<-999999,None>,<999999,None>]
level 3 [<-999999,None>,<999999,None>]
level 2 [<-999999,None>,<15004,197>,<999999,None>]
level 1 [<-999999,None>,<15001,119>,<15004,197>,<15006,196>,<999999,None>]
level 0 [<-999999,None>,<15000,185>,<15001,194>,<15004,197>,<15006,200>,<999999,None>]

End of testing.
djmaya2-iro-61:Maps major$
```