

# Map, hash table and skip list

## 1 Maps and dictionary

A dictionary or map is a collection of pairs (*key*, *values*) that is searchable. The main operations of a map are inserting an item, deleting an item, and finding an item. The map do not allow for multiple items to have the same key. Given a map  $M$  it is possible to access the value of the item with key  $k$  with  $M[k]$ . We can insert an item using  $M[k] = v$ . We can delete an item with `del M[k]`. `len(M)` return the number of items in the dictionary. The default iteration `__iter__` for a map generate the sequences of *keys* of the map, i.e. it is equivalent to `for k in M: k in M` return True if the key is in the map. `M.get(k, d=None)` return the value corresponding to key  $k$  if it exist else it returns None. `M.setdefault(k, d)` return  $M[k]$  if it exist else set  $M[k]=d$  and return  $d$ . `M.pop(k, d)` remove the item with key  $k$  and return its value; if  $k \notin M$  return  $d$  or raise a `KeyError` if  $d$  is None. `M.popitem()` remove an arbitrary pair  $(k, v)$  from  $M$  and return it. `M.clear` empty the map. `M.keys()` and `M.values()` return respectively the set of keys and the set of values in the map. `M.items()` return a set of pairs in the map. `M.update(D)` will set  $M[k]=v$  for each pair  $(k, v) \in D$ . `M1==M2` return `True` if all  $M1$  and  $M2$  have identical pair-value associations.

**MutableMapping abstract base class.** The class `MutableMapping` provide a concrete implementation of all the method describe in above except for the methods `__getitem__`, `__setitem__`, `__iter__`, `__len__`, and `__delitem__`. This means that a class that inherit from `MutableMapping` should implement these methods, else a `NotImplementedError` should be raise.

**The Map class.** Before implementing the class we need an `_Item` to represent the objects in the map.

```
class Item:
    def __init__(self, key, value=None):
        self._key = key
        self._value = value

    def key(self):
        return self._key

    def value(self):
        return self._value
```

Now we can implement the class Map. The class Map needs a `is_empty()` method that return True if the map is empty, a `get(k)` method that return the value corresponding to key  $k$ , and a `setdefault(k, d)` method that returns the value corresponding to the key  $k$  if it is in the map, else it set  $M[k]=d$  and return  $d$ .

```
class Map(MutableMapping, Item):
    def is_empty(self):
        return len(self) == 0
```

```

def get(self, k, d=None):
    try:
        return self[k]
    except KeyError:
        return d

def setdefault(self, k, d=None):
    if self.get(k) is not None:
        return self.get(k)
    else:
        self[k] = d
        return d

```

## 1.1 Implementing Map with a list.

We can implement Map with an unsorted list. We stock the element of a Map is a list  $S$  in an arbitrary order. We need to implement 7 methods for the class Map: `__setitem__` set a key to a particular value and return nothing, `__getitem__`, `__len__`, `__iter__` that provide an iterator on the keys, `__items__` that provide an iterator on the items, `__contains__`, and `__delitem__`.

```

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

    def __getitem__(self, k):
        for item in self._T:
            if k == item._key:
                return item._values
        raise KeyError(k)

    def __setitem__(self, k, v):
        for item in self._T:
            if k == item._key:
                item._value = v
                return
        self._T.append(Item(k, v))


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

    def __contains__(self, k):
        try:
            self[k]
        except KeyError:
            return False
        return True

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

    def __items__(self):
        for item in self._T:

```

A rectangular box with a black border, containing the text 'lookup\_table.png' in a monospaced font.

```
        yield (item.key(), item.value())

    def __delitem__(self, k):
        for i, item in enumerate(self._T):
            if k == item.key():
                self._T.pop(i)
                return
        raise KeyError(k)
```

**Performance of the Map implementation with an unsorted list** Inserting can be done in  $O(1)$  since we can insert an item at the end of the list using `append`, but because we need if the key is already in the Map, the operation takes  $O(n)$ . Searching and deleting takes  $O(n)$ -time since we need to loop in all the items of the Map in the worst case. The implementation with the unsorted list is practical only for Map of small size.

## 2 Hash Tables

A map support the abstraction of using keys to access values with the syntax  $M[K]$ . This is similar to the representation of a lookup table. As a mental warm-up, we consider restricted case where the Map  $M$  have  $n$  elements with integer keys  $0, 2, \dots, N-1$  for  $N \geq n$ . For example, consider the Map  $\{1: D, 3: Z, 6: C, 7: Q\}$ .

## 3 Array based sequence

### 3.1 Dynamic Array

- Accessor methods

```
__len__()
__getitem__(k)
```

- Methods

```
_make_array(c)
append(e)
_resize(c)
```

```
class DynamicArray:

    def _make_array(self, c):
        """Return a new array with capacity `c`."""
        return (c * ctypes.py_object)()

    def __init__(self):
        """Create an empty array."""
```

```

self._n = 0
self._capacity = 1
self._array = self._make_array(self._capacity)

def __len__(self):
    """Return the number of element stored in the array."""
    return self._n

def __getitem__(self, k):
    """Return the element at index `k`."""
    if not 0 <= k < self._n:
        raise IndexError('invalid index')
    return self._array[k]

def __setitem__(self, k, e):
    """Return the element at index `k`."""
    self._array[k] = e

def append(self, e):
    """Add element `e` at the end of the array"""
    if self._n == self._capacity:
        self._resize(2 * self._capacity)
    self._array[self._n] = e
    self._n += 1

def _resize(self, c):
    """Resize internal array to capacity `c`."""
    B = self._make_array(c)
    for i in range(self._n):
        B[i] = self[i]
    self._capacity = c
    self._array = B

def remove(self, k):
    obj_to_remove = self[k]
    for i in range(k, self._n-1):
        self[i] = self[i+1]
    self[self._n-1] = None
    self._n -= 1
    return obj_to_remove

def pop(self):
    obj = self[self._n-1]
    self.remove(self._n-1)
    if self._n <= self._capacity / 4:
        self._resize(self._capacity // 2)
    return obj

def find(self, obj):
    """Return the index of `obj` if `obj` is in the list else `None`"""

```

```

    for k in range(self._n):
        if self[k] == obj:
            return k
    return None

def insert(self, obj, k):
    if not 0 <= k < self._n:
        raise IndexError('invalid index')
    self.append(None)
    for i in range(self._n-1, k, -1):
        self[i] = self[i-1]
    self[k] = obj

```

## 4 Stack

A **Stack** is a collection of element that are added and removed according to the *last-in, first-out* principle. Formally, as stack is an abstract data type characterized by two methods: the method `push` and the method `pop`. Let  $S$  be an instance of a stack. The method `push( $e$ )` adds element  $e$  at the top of the stack (usually at the end?), e.g. if  $S = [1]$  then after executing `S.push(2)` we have  $S = [1, 2]$ . The method `pop()` remove and return the element at the top of the stack. An instance of a stack also have 3 *accessor methods*: `top()` return the element at the top of the stack, `is_empty()` return `True` if the stack does not contain any element, and `len()` return the number of elements in the stack.

### 4.1 Implementing a Stack using a Python List

```

class ListStack:
    def __init__(self):
        self._data = [] # nonpublic list instance

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

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

    def top(self):
        if self.is_empty():
            return None
        return self._data[-1]

    def pop(self):
        if self.is_empty():
            return None
        return self._data.pop()

    def push(self, e):
        self._data.append(e)

```

## 4.2 implementing a Stack using a Dynamic Array

```
from dynamic_array import DynamicArray
class ArrayStack:
    def __init__(self):
        self._data = DynamicArray()

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

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

    def top(self):
        if self.is_empty():
            return None
        return self._data[-1]

    def pop(self):
        if self.is_empty():
            return None
        return self._data.pop()

    def push(self, e):
        self._data.append(e)
```

## 4.3 Example: Evaluation of arithmetic expression

# 5 Queue

A **Queue** is a collection of object that are inserted and removed according to the *first-in, first-out* principle. We usually say that elements enter a queue at the back and are removed from the front.

## 5.1 The ADT Queue

- Methods

`enqueue(e)` Add element *e* to the end of the queue.

`dequeue()` Remove and return the first element of the queue.

- Accessor methods

`__len__()`

`is_empty()`

`first()` return a reference to the element at the front of the queue.

```
class Queue:

    DEFAULT_CAPACITY = 10

    def __init__(self, capacity=DEFAULT_CAPACITY):
```

```

self._data = [None] * capacity
self._capacity = capacity
self._size = 0
self._first = 0

def __len__(self):
    return self._size

def is_empty(self):
    return self._size == 0

def first(self):
    if self.is_empty():
        return None
    return self._data[self._first]

def enqueue(self, obj):
    if self._size == self._capacity:
        self._resize(2 * self._capacity)
    avail_idx = (self._first + self._size) % self._capacity
    self._data[avail_idx] = obj
    self._size += 1

def dequeue(self):
    if self.is_empty():
        return None
    obj = self._data[self._first]
    self._data[self._first] = None
    self._first = (self._first + 1) % self._capacity
    self._size -= 1
    return obj

def _resize(self, c):
    old_data = self._data
    self._data = [None] * c
    walk = self._first
    for i in range(self._size):
        self._data[i] = old_data[walk]
        walk = (self._first + 1) % self._capacity
    self._front = 0
    self._capacity = c

```

An inefficient way to implement a queue would be create an adapter using the Python `list`. The method `enqueue(e)` could be implemented with the method `append(e)` and the method `dequeue()` could be implemented as follow:

```

class Queue:
    ...
    def dequeue(self):
        if self.is_empty(): return None
        first_obj = self._data[0]
        for i in range(len(self._data)):
            self._data[i] = self._data[i+1]
        self._data[len(self._data)] = None

```

```
self._size -= 1
return first_obj
```

However, this method is tragically inefficient as its time complexity is  $\Theta(n)$ . Another approach would be to keep an instance variable `_first` to refer to the index of the next element in the queue. When the method `dequeue()` is called, the element at index `_first` is replaced by `None` and `_first` is incremented by one.

## 5.2 Using an array circularity

Suppose the current queue is given by

A	B	C	D	E	F	None	None	None	None
---	---	---	---	---	---	------	------	------	------

with a capacity of 10, a size of 6, and index 0 as the first position. Let say we dequeue one time, thus we remove `A`, replace it by `None` and increment `_first` by one. Now, let's say we enqueue four times for the elements `G`, `H`, `I`, and `J`. The resulting queue will look like this

None	B	C	D	E	F	G	H	I	J
------	---	---	---	---	---	---	---	---	---

The size is now equal to 9, the first index is 1, and the capacity still 10. Finally, let say we want to enqueue `K`. One option would be to increase the size of the array and insert it at the end. This option could lead to a huge waste since decrease and increase often maintaining a modest average size. Another option would be to insert `K` at position 0. The arithmetic expression to determine the index of insertion that way is

```
_first + _size % _capacity
```

and when we dequeue, instead of incrementing the first index by one, we have to consider the circularity

```
_first = (_first + 1) % _capacity
```

## 6 Linked list

### 6.1 Singly linked list

Une liste simplement chaînée est une structure de données concrète constituée d'une séquence de noeuds à partir d'une référence vers le noeud de tête, où chaque noeud contient deux références : vers un élément et vers le noeud suivant. On gardera aussi une référence sur le dernier noeud et un entier pour le nombre d'éléments dans la liste.

#### 6.1.1 The Node

A node contains an element and a pointer to the next node.

```
class Node:
    def __init__(self, element, next=None):
        self.element = element
        self.next = next
```

Listing 1: Singly linked Node

Once you have the Node class, you can implement any linked list as follows:



```

from node import Node

if __name__ == '__main__':
    node1 = Node('one')
    node2 = Node('two')
    node3 = Node('three')

    node1.next = node2
    node2.next = node3

```

Listing 2: Singly linked Node

### 6.1.2 Interface of a singly linked list

Now let's write the interface for the class `SinglyLinkedList`. We define three private variable in the constructor. The instance variables `_head` and `_last` refer to the first and the last element of the list, respectively. The instance variable `_size` is a reference to the last element of the list.

- Methods

- `append(e)` Add element  $e$  to the end of the list.
  - `insert(e)` Add element  $e$  at the beginning of the list
  - `remove(k)` Remove the  $k$ th element of the list in  $O(n)$

- Accessor methods

- `_len_()` return the number of element stored in the list
  - `is_empty()` return `True` if the list is empty
  - `first()` return a reference to the element at the head of the list.
  - `last()` return a reference to the element at the tail of the list.
  - `find(e)` return the index of element  $e$  in  $O(n)$  or `None`

We need an `append`, an `insert`, and a `remove` method. The method `append` add an element at the end of the linked list. The method `insert` add an element at the beginning of the linked list. The method `remove` takes an argument `k` and remove the element at position  $k$  in the list.

The constructor of the singly linked list is

```

class SinglyLinkedList
    def __init__(self):
        self._head = None
        self._tail = None
        self._size = 0

```

### 6.1.3 Accessor Methods

```

class SinglyLinkedList
    ...
    def __len__(self):
        return self._size
    def first(self):
        return self._head.element
    def last(self):
        return self._tail.element
    def find(self, k):
        pass

```

The `find()` method is more tricky. To be able to reach the  $k$ th element of a list, we need to move from the head to the tail by going from one node to the other. We will know we reach the tail when a node will refer to None as the next element. This process is called *traversing* the linked list.

```

class SinglyLinkedList
    ...
    def find(self, e):
        if self.is_empty():
            return None
        curr = self._head._next
        for i in range(self._size):
            if element == curr:
                return i
            curr = curr._next
        return None

```

#### 6.1.4 Insert Methods

To insert an element at the beginning of the linked list, we first need to check if the list is empty. If not, we need to create a new node with the new element and an instance next that refers to the current head. After that, we can set the head instance to refer to the new node.

```

class SinglyLinkedList
    ...
    def insert(self, e):
        new_head = Node(e, self._head)
        if self.is_empty():
            self._tail = new_head
        self._head = new_head
        self._size += 1

```

#### 6.1.5 Append Methods

```

class SinglyLinkedList
    ...
    def append(self, e):
        new_tail = Node(e, None)
        if self.is_empty():
            self._head = new_tail
            self._tail = new_tail
        else:

```

```

        self._tail._next = new_tail
        self._tail = new_tail
    self._size += 1

```

### 6.1.6 Remove Methods

```

class SinglyLinkedList
...
def remove(self, k):
    if self.is_empty():
        raise EmptyError('the list is empty')
    if not 0 <= k < self._size:
        raise IndexError('index invalid for list of size {}'.format(self._size))
    prev_node = None
    curr_node = self._head
    for i in range(k):
        prev_node = curr_node          # curr_node is index `k-1`
        curr_node = curr_node._next    # curr_node is index `k`
    if prev_node is None:              # index `k` is 0
        self._head = curr_node._next  # remove current head
    else:
        prev_node._next = curr_node._next # remove current node
    self._size -= 1
    if self._size == 0:
        self._tail = None              # remove reference tail <- removed node
    if prev_node._next is None:        # tail was removed
        self._tail = prev_node
    return curr_node._element

```

## 7 The Doubly linked List

The node of a doubly linked list have 3 instances: `element` is a reference to the value stored, `prev` is a reference to the previous element, and `next` is a reference to the next element.

```

class Node:
    def __init__(self, element, prev, next):
        self.element = element
        self.prev = prev
        self.next = next

```

A doubly linked list have 2 special nodes instance at the beginning and the end of the list: the **header** node have a reference to the first element of the list and the **trailer** node have a reference to the last element of the list stored in is `prev` instance.

```

class DoublyLinkedList:
    class _Node:
        ...
    def __init__(self):
        self._header = self._Node(None, None, None)
        self._trailer = self._Node(None, None, None)
        self._header.next = self._trailer

```

```
self._trailer.prev = self._header
self._size = 0
```

## 7.1 Methods

- Methods

```
append(e)  Add element e to the end of the list in  $O(1)$ .
insert(e)  Add element e at the beginning of the list in  $O(1)$ 
remove(k)  Remove the kth element of the list in  $O(?)$ 
insert_between(e, prev, next)
```

- Accessor methods

```
__len__()  return the number of element stored in the list
is_empty() return True if the list is empty
first()    return a reference to the element at the head of the list in  $O(1)$ .
last()     return a reference to the element at the beginning of the list in  $O(1)$ .
find(e)    return the index of element e in  $O(n)$  or None
```

```
class DoublyLinkedList:
    ...
    def __len__(self):
        return self._size
    def is_empty(self):
        return self._size == 0
    def first(self):
        if self.is_empty():
            raise EmptyError('the list is empty')
        return self._header.next.element
    def last(self):
        if self.is_empty():
            raise EmptyError('the list is empty')
        return self._trailer.prev.element
    def find(self, e):
        if self.is_empty():
            raise EmptyError('the list is empty')
        curr_node = self._header.next
        for i in range(self._size):
            if element == curr_node:
                return i
            curr_node = curr_node._next
        return None
```

```
class DoublyLinkedList:
    ...
    def append(e):
        new_node = self._Node(e, self._trailer.prev, self._trailer)
        self._trailer.prev.next = new_node
```

```

        self._trailer.prev = new_node
        self._size += 1

    def insert(self, element):
        new_node = self._Node(e, self._header, self._header.next)
        self._header.next.prev = new_node
        self._header.next = new_node
        self._siz += 1

    def remove(self, k):
        if self.is_empty():
            return EmptyError('list is empty')
        if not 0 <= k < self._size:
            return IndexError('index out of bound')
        curr_node = self._header.next
        for i in range(k):
            curr_node = curr_node.next
        curr_node.prev.next = curr_node.next
        curr_node.next.prev = curr_node.prev
        self._size -= 1
        curr_node.next = None
        return curr_node.element

    def insert_between(self, e, k1, k2):
        curr = self._header.next
        for i in range(k1):
            curr = curr.next
        new = self._Node(e, curr, curr.next)
        curr.next.prev = new
        curr.next = new
        self._size += 1

```

## 8 Double Ended Queue

## 9 The Positional List

## 10 Tree

```

from ListQueue import ListQueue

class DoublyLinkedList:
    ...
    def __len__(self):
        return self._size
    def is_empty(self):
        return self._size == 0
    def first(self):
        if self.is_empty():
            raise EmptyError('the list is empty')
        return self._header.next.element
    def last(self):

```

```

if self.is_empty():
    raise EmptyError('the list is empty')
return self._trailer.prev.element
def find(self, e):
    if self.is_empty():
        raise EmptyError('the list is empty')
    curr_node = self._header.next
    for i in range(self._size):
        if element == curr_node:
            return i
    curr_node = curr_node._next
    return None

```

```

def _link_leaves(self):
    Q = ListQueue()
    found_first_leaf = False
    prev_leaf = None

    Q.enqueue(self._root)

    while not Q.is_empty():
        p = Q.dequeue()
        if prev_leaf is not None and p._is_leaf():
            prev_leaf._right_leaf = p
            prev_leaf = p
        if not found_first_leaf and p._is_leaf():
            self._first_leaf = p
            prev_leaf = self._first_leaf
        for c in p._children():
            Q.enqueue(c)
    self._last_leaf = p
    self._last_leaf._right_leaf = self._first_leaf

```

```

def find_int(S, k, i, j):
    if S[i] + S[j]

```

```

from node import Node
from list import List

class SinglyLinkedList(List):

    def __init__( self ):
        self._head = None
        self._last = None
        self._size = 0

    def __len__(self):
        # return the lenght of the list
        pass

    def __str__(self):
        # return a string representation of the list
        pass

    def is_empty(self):
        # return True if the list is empty else False
        pass

    def append(self, element):
        # add an element at the end of the list in O(1)
        pass

    def insert(self, element):
        # add an element at the beginning of the list in O(1)
        pass

    def remove( self, k ):
        # remove the element at index k in O(n)
        pass

    def find(self, element):
        # return the index where element is located
        # or None if the element is not in the list
        pass

    def last( self ):
        # return the last element if the list is not empty
        pass

    def first( self ):
        # return the first element if the list is not empty
        pass

```

Listing 3: Interface for the singly linked list