Scientific Programming Practical 16

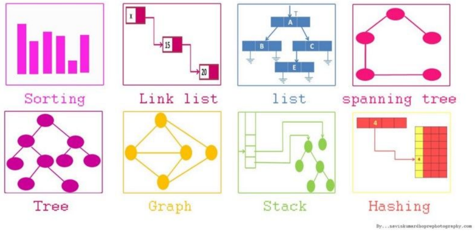
Introduction

Luca Bianco - Academic Year 2018-19
luca.bianco@fmach.it

Data structures: motivation

The way we arrange data within our programs depends on the operations we want to perform on them. To be as effective as possible, **we should pick the data structure that gives the most efficient access to the data** according to the operations we intend to perform on the data.

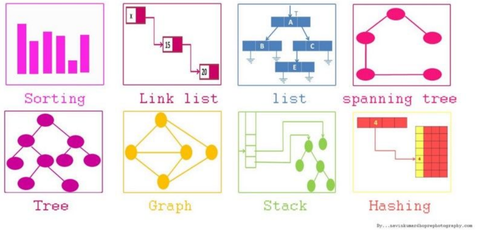
To be able to decide what data type suits better our needs, **we need to know the different features of the various datatypes** that are available (or that we can implement).



Abstract data types (ADT)

Abstract Data Types are a mathematical description of ways to store the data and of operations that can be performed on the data (in abstract terms, without focusing on implementation).

Abstract data types basically **provide the specifications of what the data types should look like/behave**.



Example: ADT Sequence

A **sequence** is a **dynamic data structure** (no limit on the number of elements) **that contains (possibly repeated) sorted elements** (sorted not by the value of the elements, but based on their position within the structure).

Operations allowed are ***remove*** elements **by their index**, ***access* directly some elements** like the first or the last (***head*** and ***tail***) **or given their index** (position), **sequentially access all the elements moving forward** (***next***) **or backwards** (***previous***) in the structure.

% Return **True** if the sequence is empty
boolean isEmpty()

% Returns the position of the first element
Pos head()

% Returns the position of the last element
Pos tail()

% Returns the position of the successor of p
Pos next(Pos p)

% Returns the position of the predecessor of p
Pos prev(Pos p)

% Inserts element v of type OBJECT in position p.
% Returns the position of the new element
Pos insert(Pos p, OBJECT v)

% Removes the element contained in position p.
% Returns the position of the successor of p, which % becomes successor of the predecessor of p
Pos remove(Pos p)

% Reads the element contained in position p
OBJECT read(Pos p)

% Writes the element v of type OBJECT in position p
write(Pos p, OBJECT v)

Sets:

Sets are dynamic data structures that contain **non-repeated** elements in **no specific order**.
Sets support: ***insert***, ***delete*** and ***contains*** to add, remove or test the presence of an element in the set.
They have a ***minimum*** and ***maximum*** to retrieve the minimum and maximum element (based on values) and it should be possible to **iterate through the elements** in the set (<u>in no specific order</u>) with something like `for el in set:`.
Finally, some **operations are defined on two sets** like: ***union***, ***intersection***, ***difference***.

% Returns the size of the set
int len()

% Returns **True** if x belongs to the set; Python: `x in S`
boolean contains(OBJECT x)

% Inserts x in the set, if not already present
add(OBJECT x)

% Removes x from the set, if present
discard(OBJECT x)

% Returns a new set which is the union of A and B
SET union(SET A, SET B)

% Returns a new set which is the intersection of A and B
SET intersection(SET A, SET B)

% Returns a new set which is the difference of A and B
SET difference(SET A, SET B)

Python sets (no import needed!):

```python
#empty set
a = set()
print(a)
a.add("Luca")
a.add("Alberto")
a.add("David")

print(a)
#adding the same element twice...
a.add("Luca")
#..has no effect
print(a)
```

```
set()
{'David', 'Alberto', 'Luca'}
{'David', 'Alberto', 'Luca'}
```

Python sets (no import needed!):

```python
#a set from a list of values
myL = [121,5,4,1,1,4,2,121]
print("\nList:{}".format(myL))
S = set(myL)
print("Set:{}".format(S))

#accessing elements:
for el in S:
    print("\telement: {}".format(el))

print("44 in S? {}".format(44 in S))
print("121 in S? {}".format(121 in S))

#from strings
S1 = set("abracadabra")
S2 = set("AbbadabE")
print("\nS1:{}".format(S1))
print("S2:{}".format(S2))
print("\nIntersection(S1,S2): {}".format(S1 & S2))
print("\nUnion(S1,S2): {}".format(S1 | S2))
print("\nIn S1 but not in S2: {}".format(S1 - S2))
print("In S2 but not in S1: {}".format(S2 - S1))
print("\nIn S1 or S2 but not in both: {}".format(S1 ^ S2))
```

```
List:[121, 5, 4, 1, 1, 4, 2, 121]
Set:{121, 2, 4, 5, 1}
        element: 121
        element: 2
        element: 4
        element: 5
        element: 1
44 in S? False
121 in S? True

S1:{'a', 'r', 'd', 'c', 'b'}
S2:{'A', 'a', 'd', 'E', 'b'}

Intersection(S1,S2): {'a', 'd', 'b'}

Union(S1,S2): {'a', 'A', 'E', 'b', 'c', 'd', 'r'}

In S1 but not in S2: {'r', 'c'}
In S2 but not in S1: {'A', 'E'}

In S1 or S2 but not in both: {'A', 'E', 'c', 'r'}
```

Queue (FIFO): ADT

Queues (also called **FIFO queues**): **first in first out queues** are linear dynamic data structures that **add at the back of the queue and remove elements from the beginning**.

QUEUE

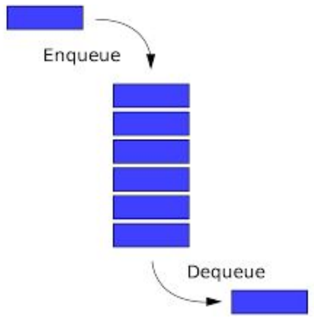
% Returns **True** if queue is empty
boolean isEmpty()

% Returns the size of the queue
int size()

% Inserts v at the end of the queue
enqueue(OBJECT v)

% Extracts q from the beginning of the queue
OBJECT dequeue()

% Reads the element at the top of the queue
OBJECT top()



Queue (FIFO)

```python
class MyQueue:

    def __init__(self):
        self.__data = list()

    def isEmpty(self):
        return len(self.__data) == 0

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

    def enqueue(self, element):
        self.__data.insert(0,element)

    def dequeue(self):
        el = None
        if len(self.__data) > 0:
            el = self.__data.pop()
        return el

    def top(self):
        if len(self.__data) > 0:
            return self.__data[-1]
```

insert at the beginning

remove from the end

QUEUE

% Returns **True** if queue is empty
boolean isEmpty()

% Returns the size of the queue
int size()

% Inserts v at the end of the queue
enqueue(OBJECT v)

% Extracts q from the beginning of the queue
OBJECT dequeue()

% Reads the element at the top of the queue
OBJECT top()

Queue (FIFO)

```python
class MyQueue:

    def __init__(self):
        self.__data = list()

    def isEmpty(self):
        return len(self.__data) == 0

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

    def enqueue(self, element):
        self.__data.insert(0,element)

    def dequeue(self):
        el = None
        if len(self.__data) > 0:
            el = self.__data.pop()
        return el

    def top(self):
        if len(self.__data) > 0:
            return self.__data[-1]
```

```python
if __name__ == "__main__":
    import time

    Q = MyQueue()
    Q.enqueue([1,2,3])
    Q.enqueue([2,3,4])
    Q.enqueue([3,4,4])
    print("Size of Q: {}".format(len(Q)))
    Q.enqueue([1,1,1])
    Q.enqueue([1,2,3])
    print("TOP is now: {}\n".format(Q.top()))
    while not Q.isEmpty():
        el = Q.dequeue()
        print("Removing el {} from queue".format(el))

    start_t = time.time()
    for i in range(400000):
        Q.enqueue(i)
    print("\nQueue has size: {}".format(len(Q)))
    #comment the next 3 lines and see what happens
    while not Q.isEmpty():
        el = Q.dequeue()
    print("\nQueue has size: {}".format(len(Q)))
    end_t = time.time()
    print("\nElapsed time: {:.2f}s".format(end_t - start_t))
```

```
Size of Q: 3
TOP is now: [1, 2, 3]

Removing el [1, 2, 3] from queue
Removing el [2, 3, 4] from queue
Removing el [3, 4, 4] from queue
Removing el [1, 1, 1] from queue
Removing el [1, 2, 3] from queue

Queue has size: 400000

Queue has size: 0

Elapsed time: 43.49s
```

Queue (FIFO). Quicker len, quicker enqueue

```
class MyQueue:

    def __init__(self):
        self.__data = list()
        self.__length = 0

    def isEmpty(self):
        return len(self.__data) == 0

    def __len__(self):
        return self.__length

    ## Add at the end not at the beginning
    def enqueue(self, element):
        self.__data.append(element)
        self.__length += 1

    def dequeue(self):
        el = None
        if len(self.__data) > 0:
            el = self.__data.pop(0)
            self.__length -= 1
        return el

    def top(self):
        if len(self.__data) > 0:
            return self.__data[-1]
```

Queue (FIFO). Quicker len, quicker enqueue

```python
if __name__ == "__main__":
    import time

    Q = MyQueue()
    Q.enqueue([1,2,3])
    Q.enqueue([2,3,4])
    Q.enqueue([3,4,4])
    print("Size of Q: {}".format(len(Q)))
    Q.enqueue([1,1,1])
    Q.enqueue([1,2,3])
    print("TOP is now: {}\n".format(Q.top()))
    while not Q.isEmpty():
        el = Q.dequeue()
        print("Removing el {} from queue".format(el))

    start_t = time.time()
    for i in range(400000):
        Q.enqueue(i)
    print("\nQueue has size: {}".format(len(Q)))
    #comment the next 3 lines and see what happens
    while not Q.isEmpty():
        el = Q.dequeue()
    print("\nQueue has size: {}".format(len(Q)))
    end_t = time.time()
    print("\nElapsed time: {:.2f}s".format(end_t - start_t))
```

try commenting this out

```
Size of Q: 3
TOP is now: [1, 2, 3]

Removing el [1, 2, 3] from queue
Removing el [2, 3, 4] from queue
Removing el [3, 4, 4] from queue
Removing el [1, 1, 1] from queue
Removing el [1, 2, 3] from queue

Queue has size: 400000

Queue has size: 0

Elapsed time: 41.48s
```

only slightly better

Queue (FIFO). Quicker len, quicker enqueue

```python
if __name__ == "__main__":
    import time

    Q = MyQueue()
    Q.enqueue([1,2,3])
    Q.enqueue([2,3,4])
    Q.enqueue([3,4,4])
    print("Size of Q: {}".format(len(Q)))
    Q.enqueue([1,1,1])
    Q.enqueue([1,2,3])
    print("TOP is now: {}\n".format(Q.top()))
    while not Q.isEmpty():
        el = Q.dequeue()
        print("Removing el {} from queue".format(el))

    start_t = time.time()
    for i in range(400000):
        Q.enqueue(i)
    print("\nQueue has size: {}".format(len(Q)))
    #comment the next 3 lines and see what happens
    #while not Q.isEmpty():
    #    el = Q.dequeue()
    print("\nQueue has size: {}".format(len(Q)))
    end_t = time.time()
    print("\nElapsed time: {:.2f}s".format(end_t - start_t))
```

```
Size of Q: 3
TOP is now: [1, 2, 3]

Removing el [1, 2, 3] from queue
Removing el [2, 3, 4] from queue
Removing el [3, 4, 4] from queue
Removing el [1, 1, 1] from queue
Removing el [1, 2, 3] from queue

Queue has size: 400000

Queue has size: 400000

Elapsed time: 0.14s
```

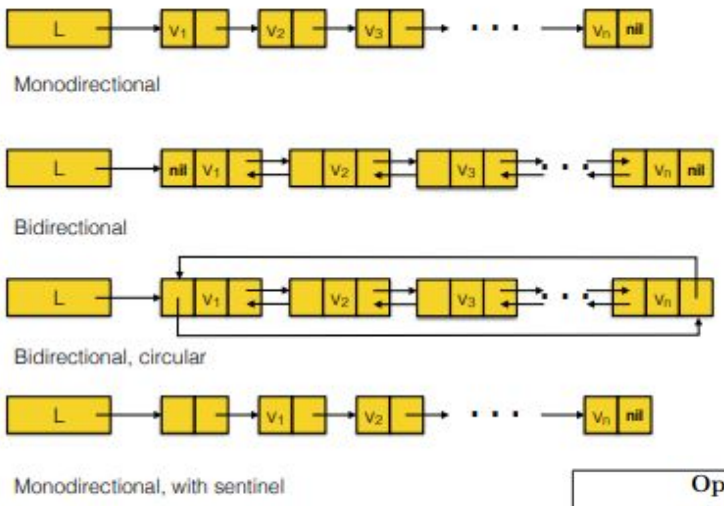
collections.deque

```
from collections import deque
import time

Q = deque()
start_t = time.time()
for i in range(400000):
    #add to the right
    Q.append(i)
print("Q has {} elements".format(len(Q)))
while len(Q) > 0:
    #remove from the left
    Q.popleft()
print("Q has {} elements".format(len(Q)))
end_t = time.time()
print("\nElapsed time: {:.2f}s".format(end_t - start_t))
```

```
Q has 400000 elements
Q has 0 elements

Elapsed time: 0.13s
```

Linked lists

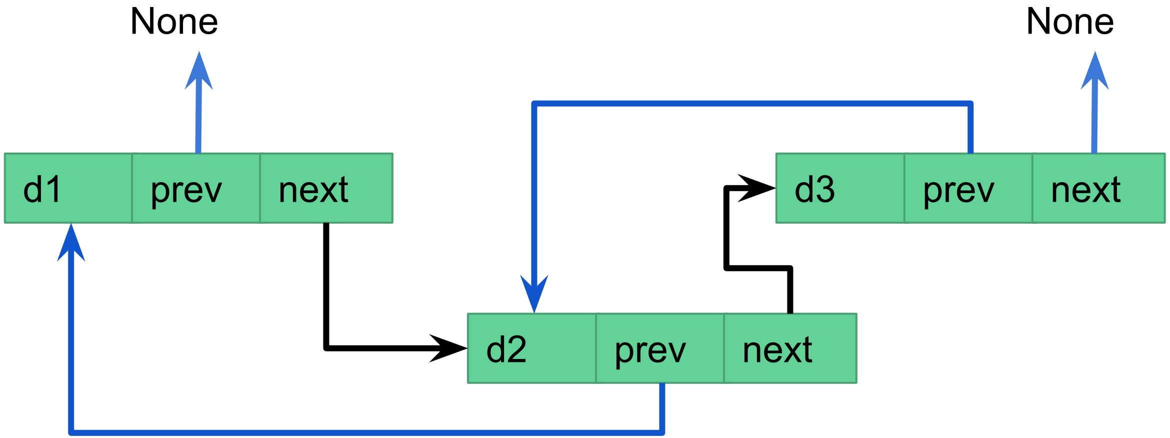
Linked lists are dynamic collections of **objects and pointers** (either 1 or 2) that **point to the next** element in the list **or to both the next and previous** element in the list.



Operator		Worst case	Worst case amortized
`L.copy()`	Copy	$O(n)$	$O(n)$
`L.append(x)`	Append	$O(n)$	$O(1)$
`L.insert(i,x)`	Insert	$O(n)$	$O(n)$
`L.remove(x)`	Remove	$O(n)$	$O(n)$
`L[i]`	Index	$O(1)$	$O(1)$
`for x in L`	Iterator	$O(n)$	$O(n)$
`L[i:i+k]`	Slicing	$O(k)$	$O(k)$
`L.extend(s)`	Extend	$O(k)$	$O(n+k)$
`x in L`	Contains	$O(n)$	$O(n)$
`min(L), max(L)`	Min, Max	$O(n)$	$O(n)$
`len(L)`	Get length	$O(1)$	$O(1)$

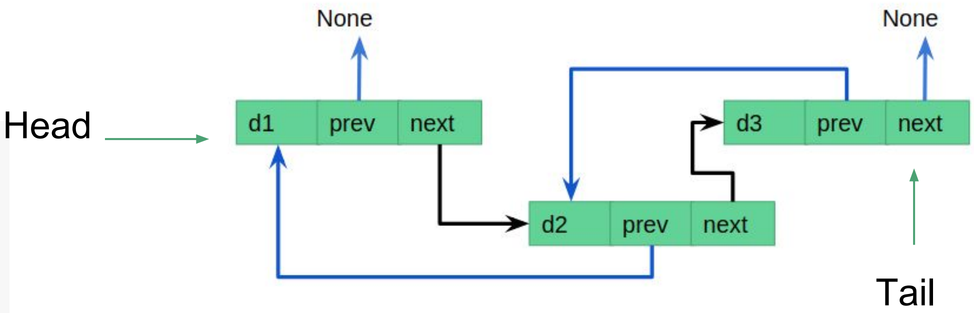
Linked lists: nodes

```python
class Node:
    def __init__(self, data):
        self.__data = data
        self.__prevEl = None
        self.__nextEl = None

    def getData(self):
        return self.__data

    def setData(self, newdata):
        self.__data = newdata

    def setNext(self, node):
        self.__nextEl = node

    def getNext(self):
        return self.__nextEl

    def setPrev(self,node):
        self.__prevEl = node
    def getPrev(self):
        return self.__prevEl

    def __str__(self):
        return str(self.__data)
    #for sorting
    def __lt__(self, other):
        return self.__data < other.__data
```



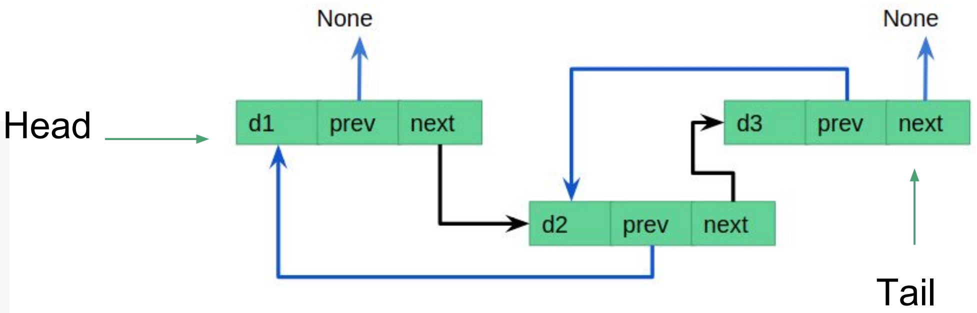
Linked lists: list

```python
class BiLinkList:
    def __init__(self):
        self.__head = None
        self.__tail = None
        self.__len = 0
        self.__minEl = None
        self.__maxEl = None

    def __len__(self):
        return self.__len
    def min(self):
        return self.__minEl
    def max(self):
        return self.__maxEl

    def append(self,node):
        if type(node) != Node:
            raise TypeError("node is not of type Node")
        else:
            if self.__head == None:
                self.__head = node
                self.__tail = node
            else:
                node.setPrev(self.__tail)
                self.__tail.setNext(node)
                self.__tail = node
            self.__len += 1
            #This assumes that nodes can be sorted
            if self.__minEl == None or self.__minEl > node:
                self.__minEl = node
            if self.__maxEl == None or self.__maxEl < node:
                self.__maxEl = node
```

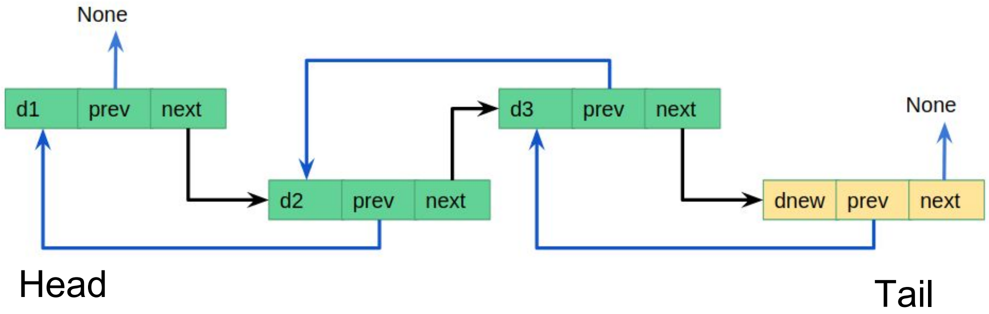


Linked lists: list

```python
class BiLinkList:
    def __init__(self):
        self.__head = None
        self.__tail = None
        self.__len = 0
        self.__minEl = None
        self.__maxEl = None

    def __len__(self):
        return self.__len
    def min(self):
        return self.__minEl
    def max(self):
        return self.__maxEl

    def append(self,node):
        if type(node) != Node:
            raise TypeError("node is not of type Node")
        else:
            if self.__head == None:
                self.__head = node
                self.__tail = node
            else:
                node.setPrev(self.__tail)
                self.__tail.setNext(node)
                self.__tail = node
            self.__len += 1
            #This assumes that nodes can be sorted
            if self.__minEl == None or self.__minEl > node:
                self.__minEl = node
            if self.__maxEl == None or self.__maxEl < node:
                self.__maxEl = node
```

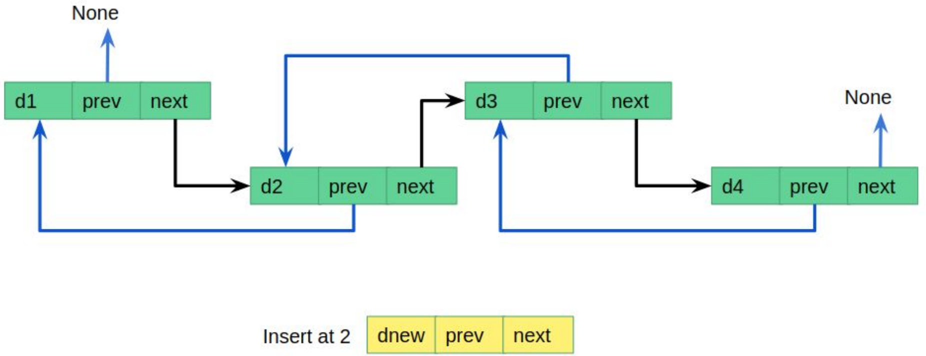


Append

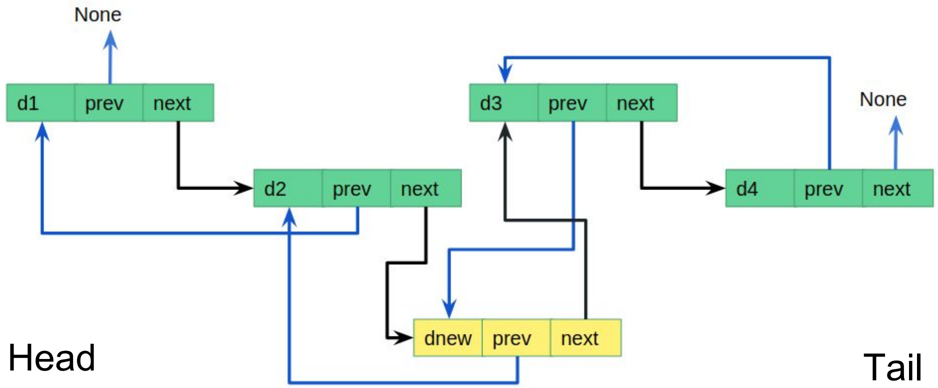


Linked lists: insert at position

```python
def insert(self, node, i):
    # to avoid index problems, if i is out of bounds
    # we insert at beginning or end
    if i > self.__len:
        i = self.__len #I know that it is after tail!
    if i < 0:
        i = 0
    cnt = 0
    cur_el = self.__head
    while cnt < i:
        cur_el = cur_el.getNext()
        cnt += 1
    #add node before cur_el
    if cur_el == self.__head:
        #add before current head
        node.setNext(self.__head)
        self.__head.setPrev(node)
        self.__head = node
    else:
        if cur_el == None:
            #add after tail
            self.__tail.setNext(node)
            node.setPrev(self.__tail)
            self.__tail = node
        else:
            p = cur_el.getPrev()
            p.setNext(node)
            node.setPrev(p)
            node.setNext(cur_el)
            cur_el.setPrev(node)

    self.__len += 1
    #This assumes that nodes can be sorted
    if self.__minEl == None or self.__minEl > node:
        self.__minEl = node
    if self.__maxEl == None or self.__maxEl < node:
        self.__maxEl = node
```



Insert at 2
First loop until you reach 2 (cur = cur.getNext())



Iterating through a collection: yield

To be able to iterate through the elements of our **custom** collection with something like **for x in mycollection.iterate()**, we need to **generate one element at a time until all elements have been obtained**.

We need to define an *iterator.*

In python we can use `yield`

```
class myClass:
    def __init__(self, x,y,z):
        self.__x = x
        self.__y = y
        self.__z = z

    def add(self,x,y,z):
        self.__x.extend(x)
        self.__y.extend(y)
        self.__z.extend(z)

    def iterator(self):
        for i in range(len(self.__x)):
            yield (self.__x[i], self.__y[i], self.__z[i])


C = myClass([0,1,2,3,4], [0,1,0,1,0], [4,3,2,1,0])
C.add([27,44],[14,4],[27,1])
for el in C.iterator():
    print("Element: {}".format(el))
```

```
Element: (0, 0, 4)
Element: (1, 1, 3)
Element: (2, 0, 2)
Element: (3, 1, 1)
Element: (4, 0, 0)
Element: (27, 14, 27)
Element: (44, 4, 1)
```

http://qcbprolab.readthedocs.io/en/latest/practical16.html

Exercises

1. Write a simple MySet class that implements the abstract data type **set**. Use a dictionary as internal data structure (hint: you can put the element as key of the dictionary and the value as 1). For simplicity, the object is constructed by passing to it a list of elements (e.g. S = mySet([1,2,3]).

The ADT of the set structure is (i.e. the methods to implement):

% Returns the size of the set
int len()

% Returns **True** if x belongs to the set; Python: `x in S`
boolean contains(OBJECT x)

% Inserts x in the set, if not already present
add(OBJECT x)

% Removes x from the set, if present
discard(OBJECT x)

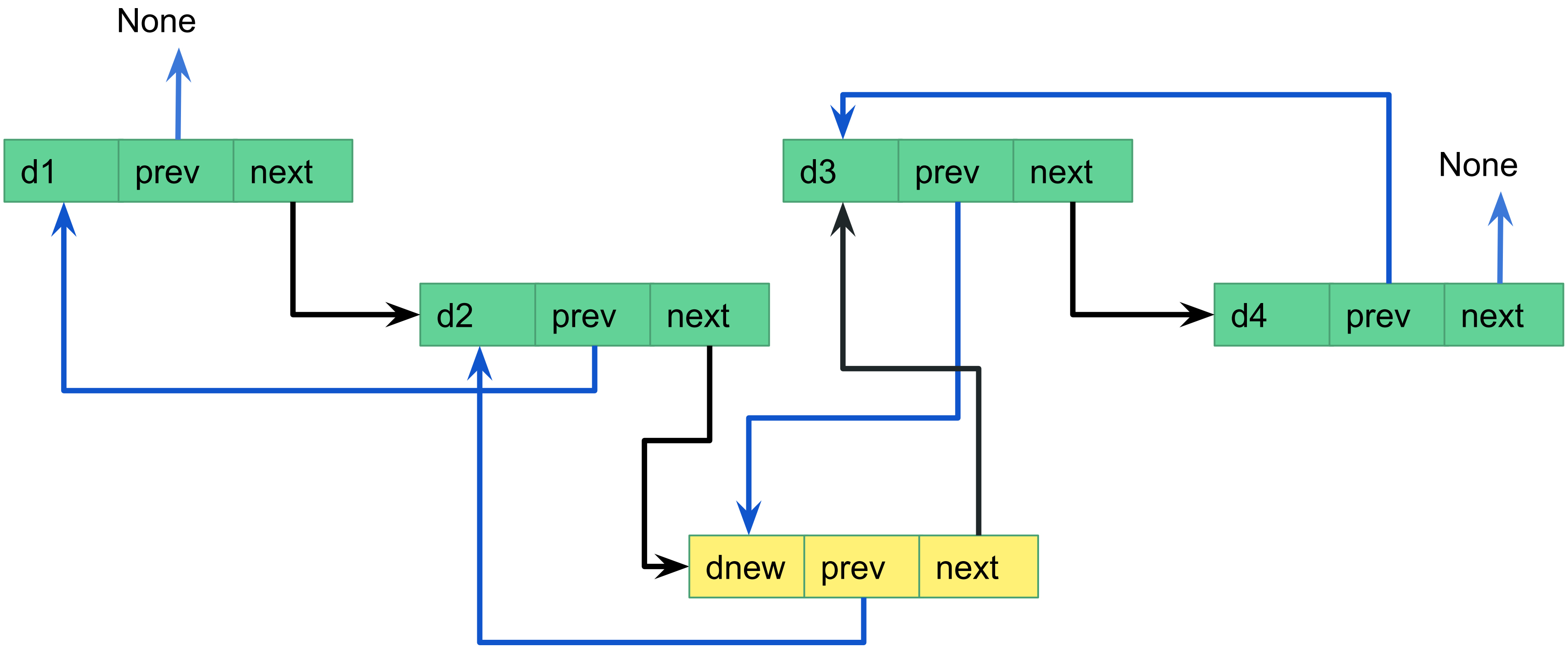
% Returns a new set which is the union of A and B
SET union(SET A, SET B)

% Returns a new set which is the intersection of A and B
SET intersection(SET A, SET B)

% Returns a new set which is the difference of A and B
SET difference(SET A, SET B)

Implement a **iterator** method that **yields** the next elements. Implement a special method "**contains**" to test if an element is present with **el in S**.

Test the code with:


None
d1
prev
next
d2
prev
next
d3
prev
next
None
d4
prev
next
dnew
prev
next