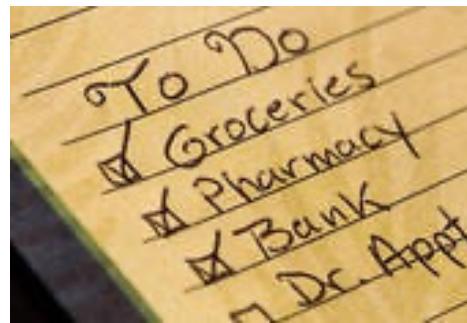


# Linked list

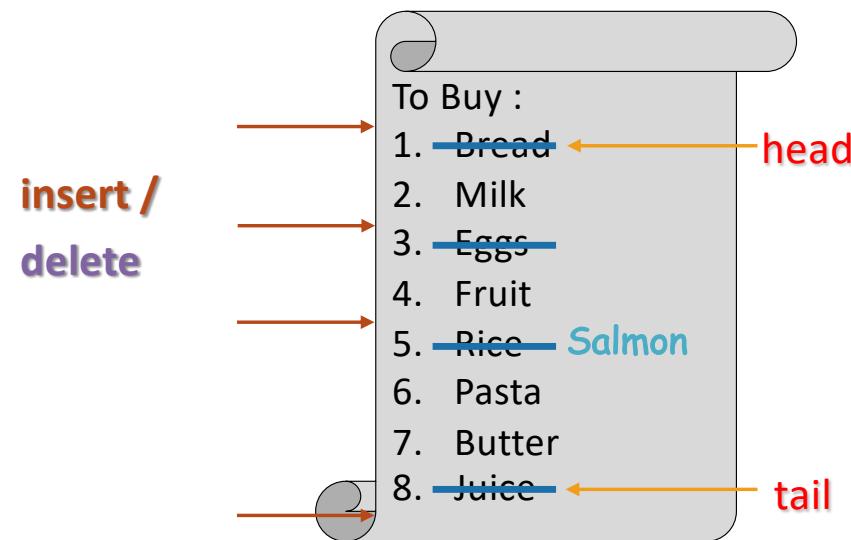
Kiatnarong Tongprasert

## List



# List

## List



Already bought  
Eggs, Rice, Bread, Juice

Oh I forgot  
Salmon

## Ordered List – Unordered List

To Buy :

1. Bread
2. Milk
3. Eggs
4. Fruit
5. Rice
6. Pasta
7. Butter
8. Juice

Unordered List

Scores :

- |          |    |
|----------|----|
| 1. Bruce | 2  |
| 2. Tom   | 3  |
| 3. Ben   | 5  |
| 4. Max   | 7  |
| 5. Tim   | 7  |
| 6. Marry | 8  |
| 7. Ron   | 9  |
| 8. Harry | 10 |

Ordered List  
Ascending Order

$$98n^5 - 4n^4 + n^3 - 8n^2 + 5n + 7$$

Ordered List  
Decending Order

## Logical Abstract Data Type & Implementation

Logical ADT : ขึ้นกับ application

1. Data : ของมีลำดับ มีปลาย หัว head และ/หรือ ท้าย tail      Data Implementation ?

Python List

2. Methods : ขึ้นกับ list เป็น ordered list หรือไม่

## Unordered List / Ordered List

- `List()` สร้าง empty list
- `isEmpty()` returns boolean ว่า empty list หรือไม่
- `size()` returns จำนวนของใน list
- `search(item)` returns ว่ามี item ใน list หรือไม่
- `index(item)` returns index ของ item กำหนดให้ item อยู่ใน list

# unordered list

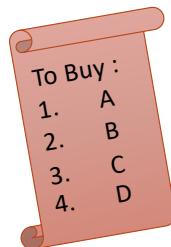
- `append(item)` adds item ท้าย list ไม่ Returns คิดว่า item ไม่มีอยู่ก่อนใน list
- `insert(pos,item)` adds item ที่ index pos ไม่ Returns คิดว่า item ไม่มีอยู่ก่อนใน list

# ordered list

- `add(item)` adds item เข้า list ตามลำดับ ไม่ Returns

- `remove(item)` removes & return item คิดว่า item มีอยู่ใน list
- `pop()` removes & return item ตัวสุดท้าย list\_size >= 1
- `pop(pos)` removes & return item ตัวที่ index = pos list\_size >= 1

## List Implementation : Sequential (Implicit) Array, Python List

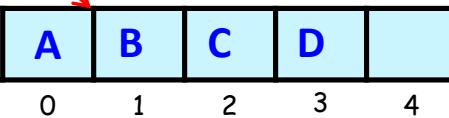


A B C D

i

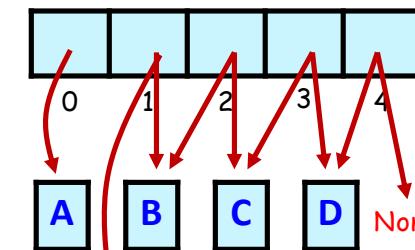
insert i ? : shift out

head



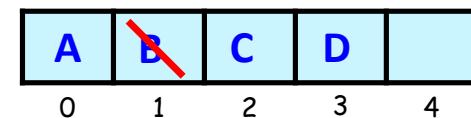
Problem : fix positions

head

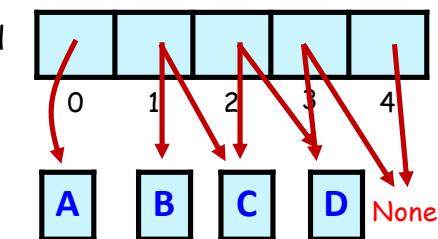


delete : shift in

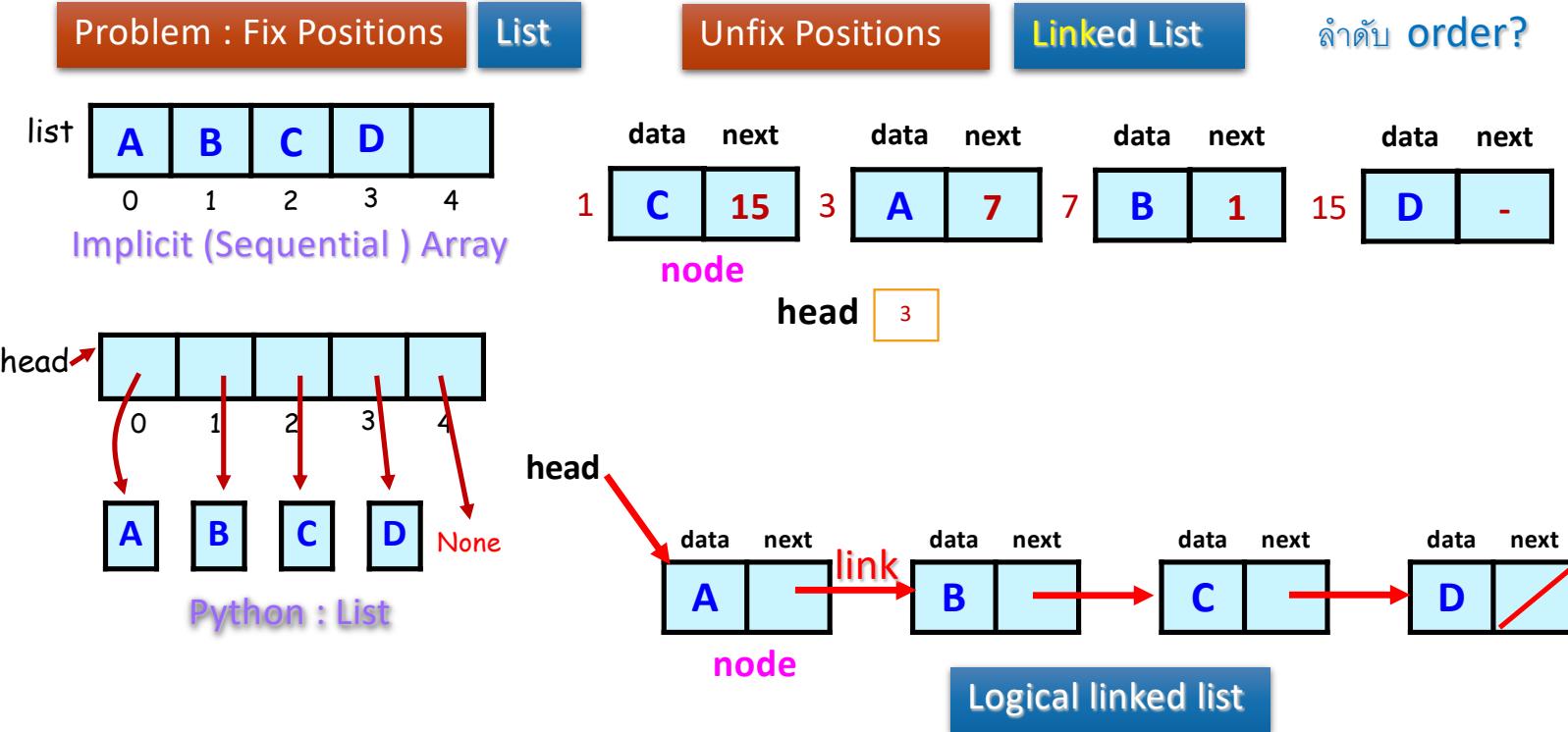
head



head



## Linked List



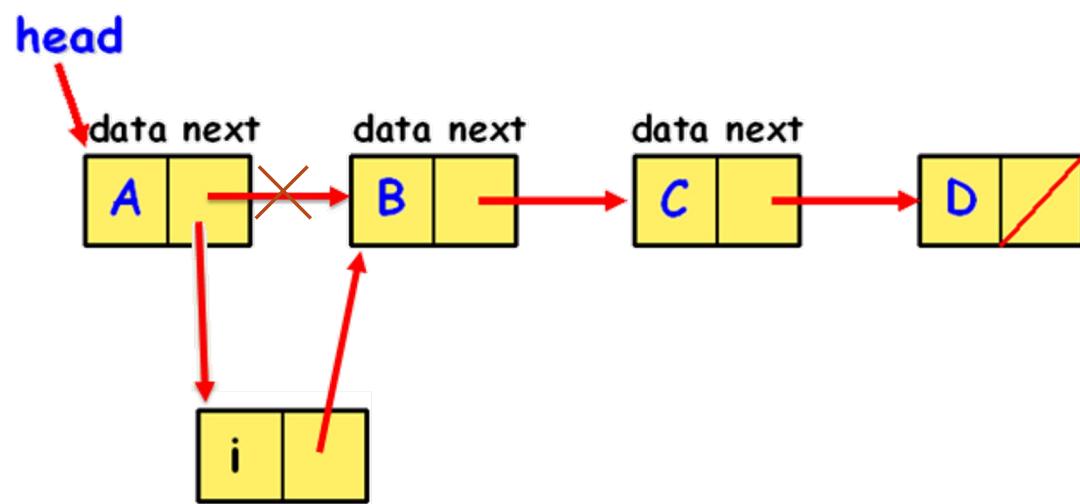
Logical คือในความคิดของเรา

เช่น link แทนด้วยลูกศร แทนการเชื่อมโยงกัน

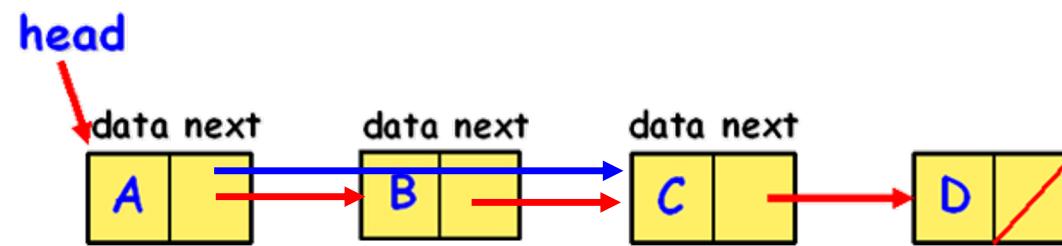
physical (implementation) โครงสร้างที่ใช้ในการสร้างจริง

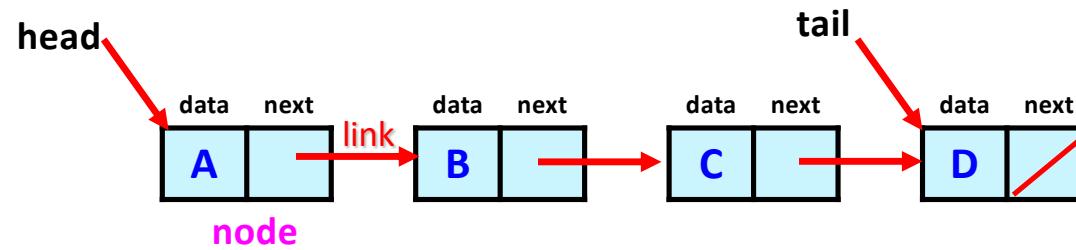
เช่น link อาจใช้ pointer หรือ index ของ array

## Solve Inserting Expensive Shifting Problem



## Solve Deleting Expensive Shifting Problem





## Linked List

Data :

- 1. data
  - 2. next (link)
  - 3. head
  - 4. tail ?
- } node class
- } list class



## Node Class / List Class

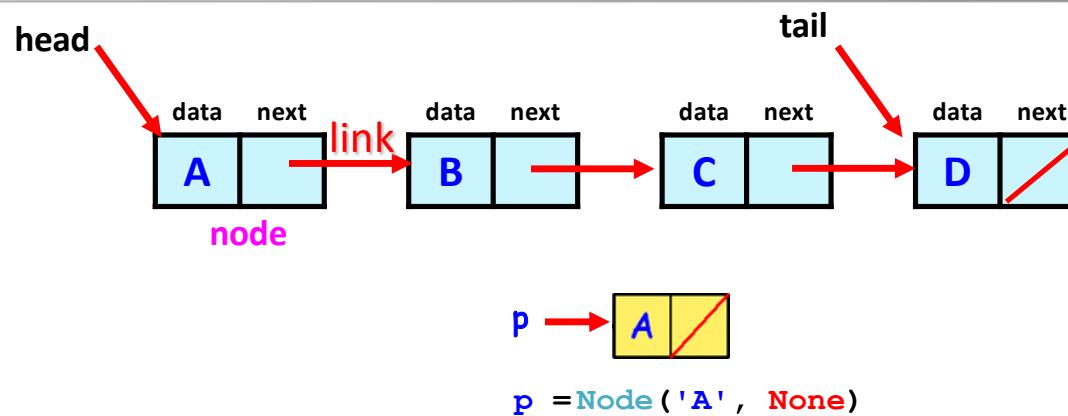
### 1. Data :

`__init__()` : **constructor** ให้ค่าตั้งต้น



2 underscores    2 underscores

## Node Class



```
class Node:

    def __init__(self, data, next = None):

        self.data = data

        if next is None:
            self.next = None
        else:
            self.next = next

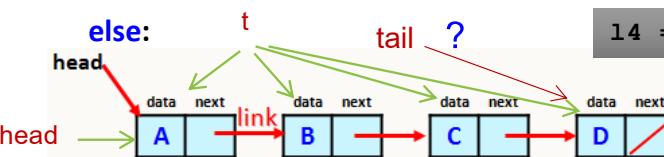
    def __str__(self):
        return str(self.data)
```

## List Class

```
class list:  
    """ unordered singly linked list  
    with head  
    """  
  
    def __init__(self):      l1 = list()  
        self.head = None
```

```
    """ unordered singly linked list  
    with head & tail  
    """  
  
    def __init__(self):      l2 = list()  
        self.head = self.tail = None
```

```
def __init__(self, head = None):  
  
    """ unordered singly linked list  
    can set default list  
    with head, tail & size  
    """  
    if head == None:          head tail → None  
        l3 = list()  
        self.head = self.tail = None  
        self.size = 0  
  
    else:                    t  
        head → A  
        t = A  
        link → B  
        tail → ?  
        l4 = list(head)  
        self.head = head  
        t = self.head  
        self.size = 1  
        while t.next != None: # locating tail & find size  
            t = t.next  
            self.size += 1  
        self.tail = t
```





## Methods

1. **`__init__()`** : ให้ค่าตั้งต้น

2. **`size()`**:

3. **`isEmpty()`**:

4. **`append ()`** : add at the end

5. **`__str__()`**:

6. **`addHead()`** : ให้ค่าตั้งต้น

7. **`remove(item)`**:

8. **`removeTail()`**:

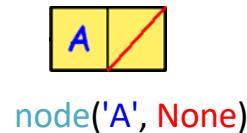
9. **`removeHead()`** :

10. **`isIn(item)`**: / **`search(item)`**

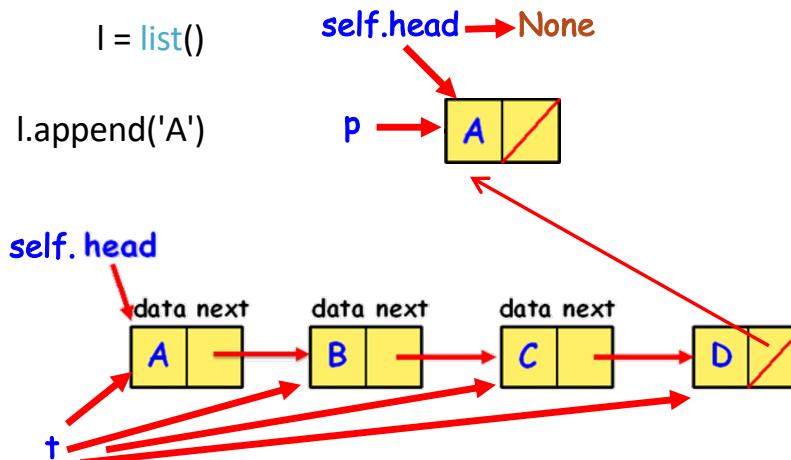
11. . . .

## Creating a List

```
class list:  
  
    """ unordered singly linked list  
    with head """  
  
    def __init__(self):  
        self.head = None  
        self.size = 0  
  
    def append(self, data):  
  
        """ add at the end of list"""  
        p = Node(data)  
        if self.head == None: # null list  
            self.head = p  
        else:  
            t = self.head  
            while t.next != None :  
                t = t.next  
            t.next = p  
            self.size += 1
```



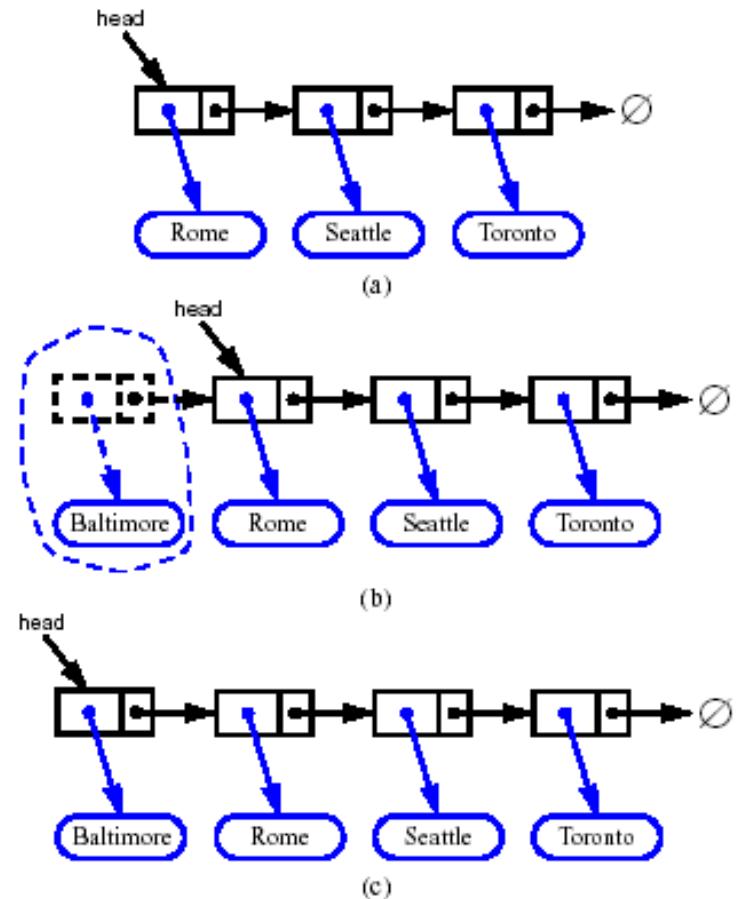
```
class Node:  
    def __init__(self, data, next = None):  
        self.data = data  
        if next == None:  
            self.next = None  
        else:  
            self.next = next  
        self.size = 0
```



# Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node

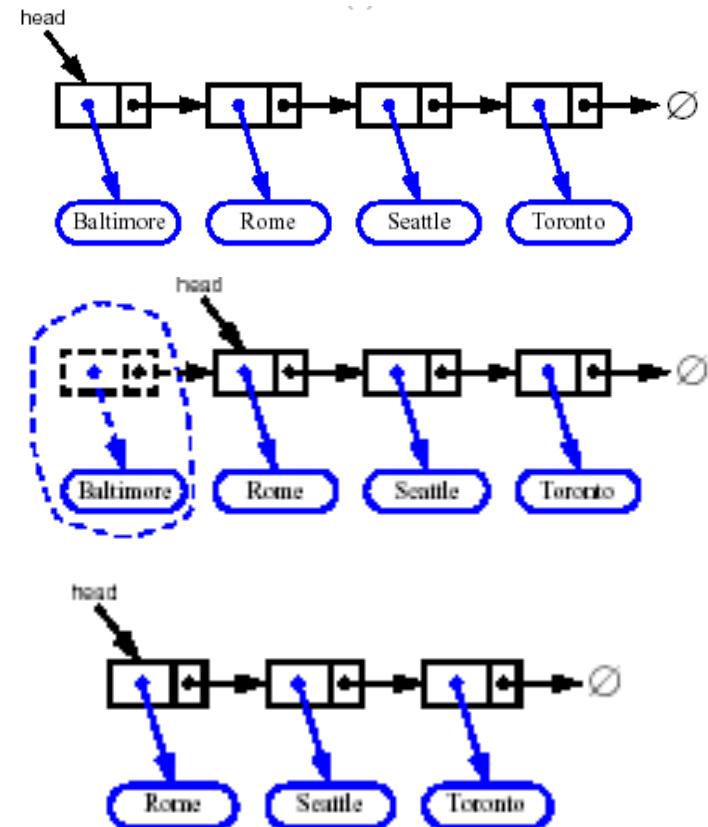
```
class list:  
    def add_head(self, data):  
        p = Node(data)  
        p.next = self.head  
        self.head = p  
        self.size += 1
```



# Removing at the Head

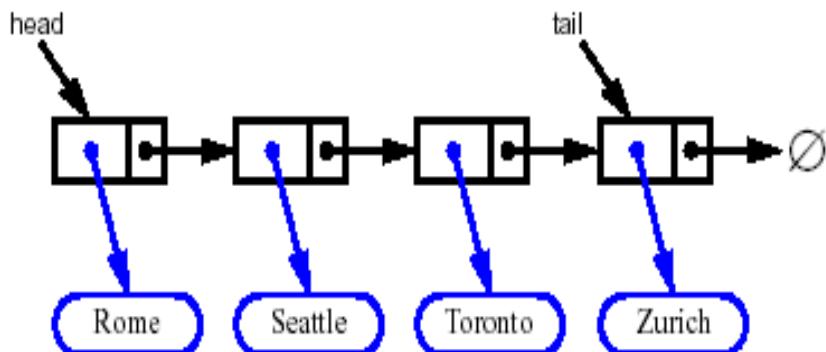
1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node

```
class list:  
    def removeHead(self):  
        if self.head == None : return  
        if self.head.next == None :  
            p = self.head  
            self.head = None  
        else :  
            p = self.head  
            self.head = self.head.next  
        self.size -= 1  
        return p.data
```



# Removing at the Tail

- ◆ Removing at the tail of a singly linked list is not efficient!
- ◆ There is no constant-time way to update the tail to point to the previous node

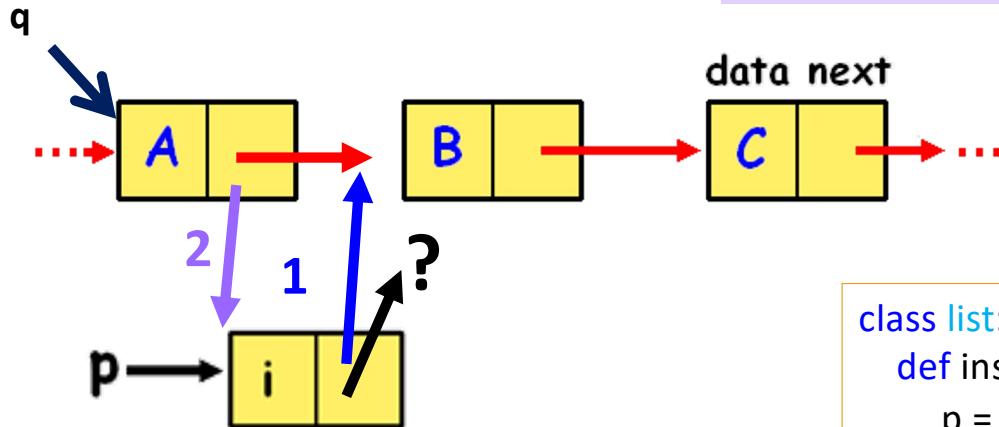


```
class list:  
    def removeTail(self):  
        if self.head == None : return  
        if self.head.next == None :  
            self.head = None  
            self.size -= 1  
            return  
        else :  
            p = self.head  
            while p.next.next != None :  
                p = p.next  
            p.next = p.next.next  
            self.size -= 1
```

## Insert After

`insertAfter(q, 'i')`

insert node data  
after a node pointed by q



```
p = node('i')  
p.next = q.next  
q.next = p
```

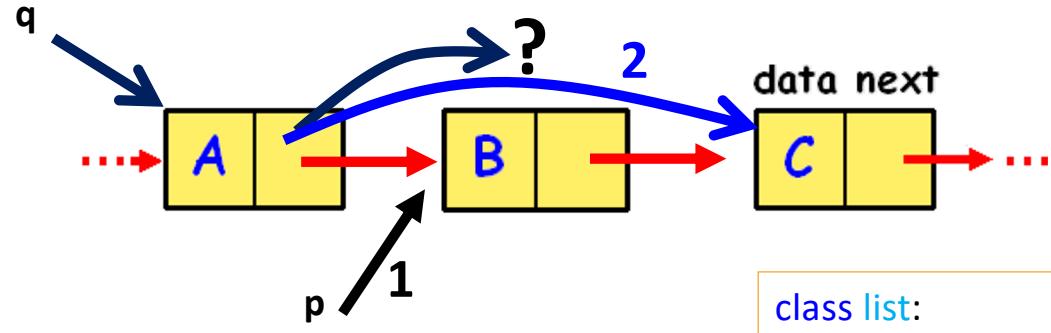
Why insert after ?  
Can you insert before ?

```
class list:  
    def insertAfter(self, i, data):  
        p = Node(data)  
        q = self.head  
        count = 0  
        while q != None :  
            if count == i:  
                p.next = q.next  
                q.next = p  
                return  
            q = q.next  
            count += 1
```

## Delete After

**deleteAfter (q)**

delete a node  
after a node pointed by q



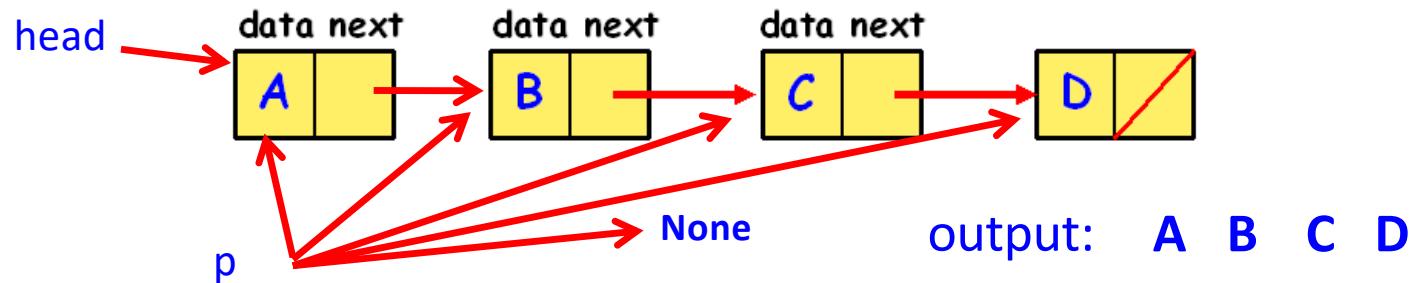
`q.next = p.next`

class list:

```
def deleteAfter(self, i):
    q = self.head
    count = 0
    while curr_node != None and curr_node.next != None:
        if count == i:
            p = q.next
            q.next = p.next
            p = None
        return
    q = q.next
    count += 1
```

## print list

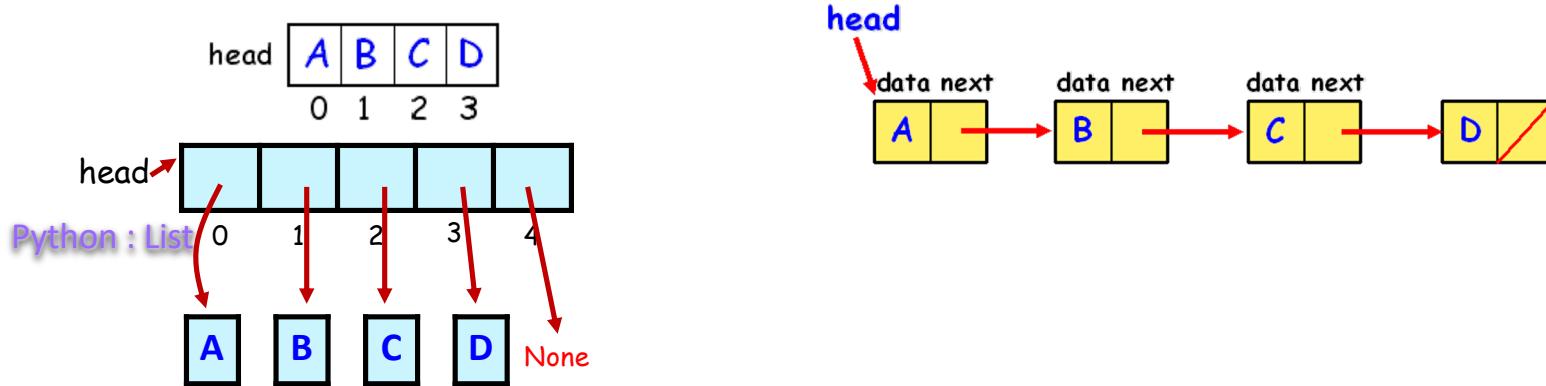
Design how to call.



```
p = head  
while p != None:  
    print(p.data)  
    p = p.next
```

```
class list:  
    def printList(self):  
        p = self.head  
        while p != None:  
            print(p.data, end=" -> ")  
            p = p.next  
        print("None")
```

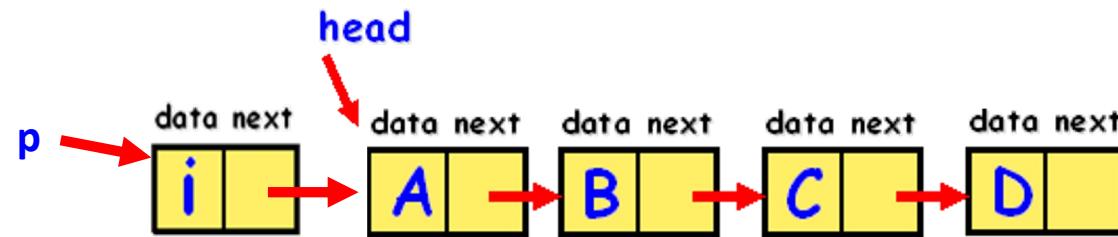
## Linked List VS Sequential Array



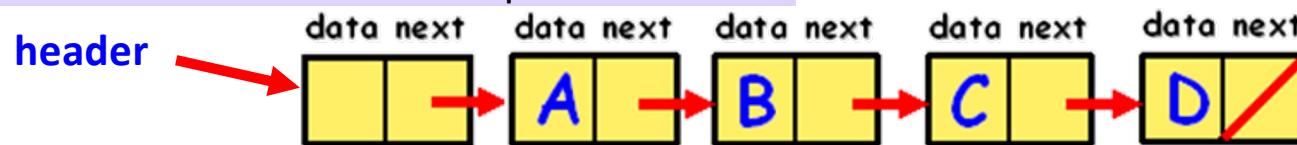
Sequential Array	Linked List
<ul style="list-style-type: none"> <li>• Insertion / Deletion Shifting Problem.</li> <li>• Random Access.</li> <li>• C array : Automatic Allocation Python List array : Dynamic Allocation</li> <li>• Lifetime : C-array, Python List <ul style="list-style-type: none"> <li>• from defined until its scope finishes.</li> </ul> </li> <li>• Only keeps data.</li> </ul>	<ul style="list-style-type: none"> <li>• Solved.</li> <li>• Sequential Access.</li> <li>• Node : Dynamic Allocation.</li> <li>• Node Lifetime : from allocated (C : malloc()/new, python: instantiate obj) until C: deallocated by free()/delete, Python : no reference.</li> <li>• Need spaces for links.</li> </ul>

## Header Node

To insert & delete at 1<sup>st</sup> position change head ie. make special case.



“Header Node” solves the problem.



Empty List has a header node.



## Header Node

```
class list:  
  
    """ unordered singly linked list  
    with head """  
  
    def __init__(self):  
        self.head = None  
        self.size = 0  
  
    def append(self, data):  
  
        """ add at the end of list"""  
        p = Node(data)  
        if self.head == None: # null list  
            self.head = p  
        else:  
            t = self.head  
            while t.next != None :  
                t = t.next  
            t.next = p  
        self.size += 1
```

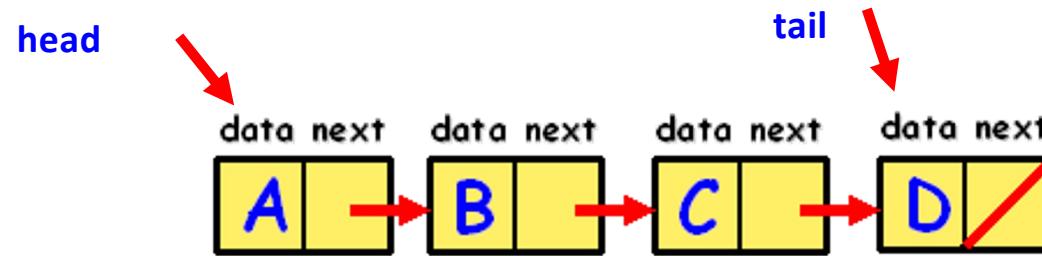
```
class list:  
  
    """ unordered singly linked list  
    with head """  
  
    def __init__(self):  
        self.header = Node()  
        self.size += 1  
  
    def append(self, data):  
  
        """ add at the end of list"""  
        p = Node(data)  
        t = self.header.next  
        while t.next != None :  
            t = t.next  
        t.next = p  
        self.size += 1
```

## Header Node

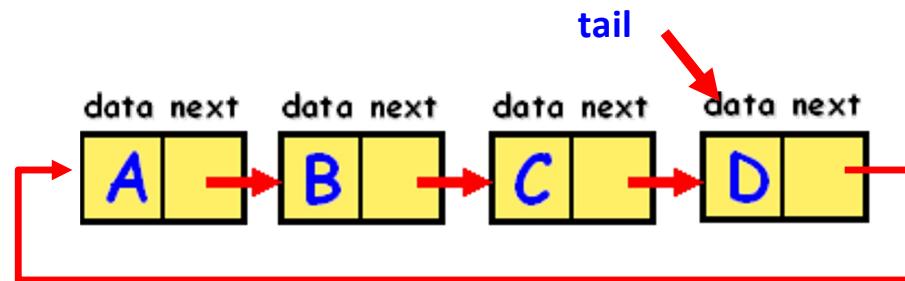
```
class list:  
    def removeTail(self):  
        if self.head == None : return  
        if self.head.next == None :  
            self.head = None  
            self.size -= 1  
            return  
        else :  
            p = self.head  
            while p.next.next != None :  
                p = p.next  
            p.next = p.next.next  
            self.size -= 1
```

```
class list:  
    def removeTail(self):  
        if self.head == None : return  
        if self.head.next == None :  
            self.head = None  
            self.size -= 1  
            return  
        p = self.head.next  
        while p.next.next != None :  
            p = p.next  
        p.next = p.next.next  
        self.size -= 1
```

## Head & Tail Nodes

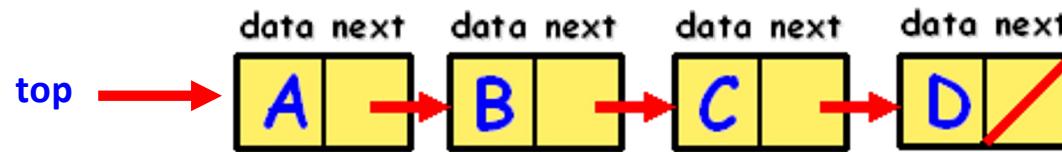


## Circular List

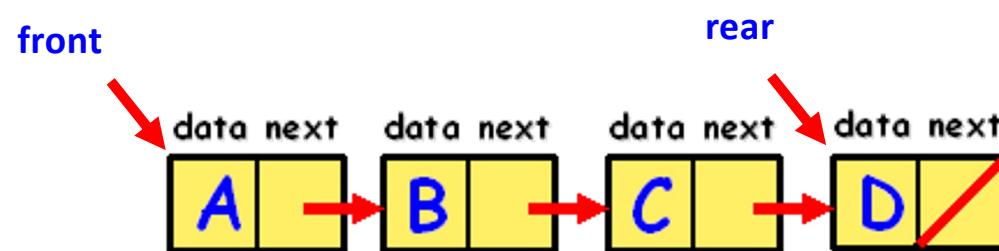


Why ptr to tail ? Why not ptr to head?

## Linked Stack



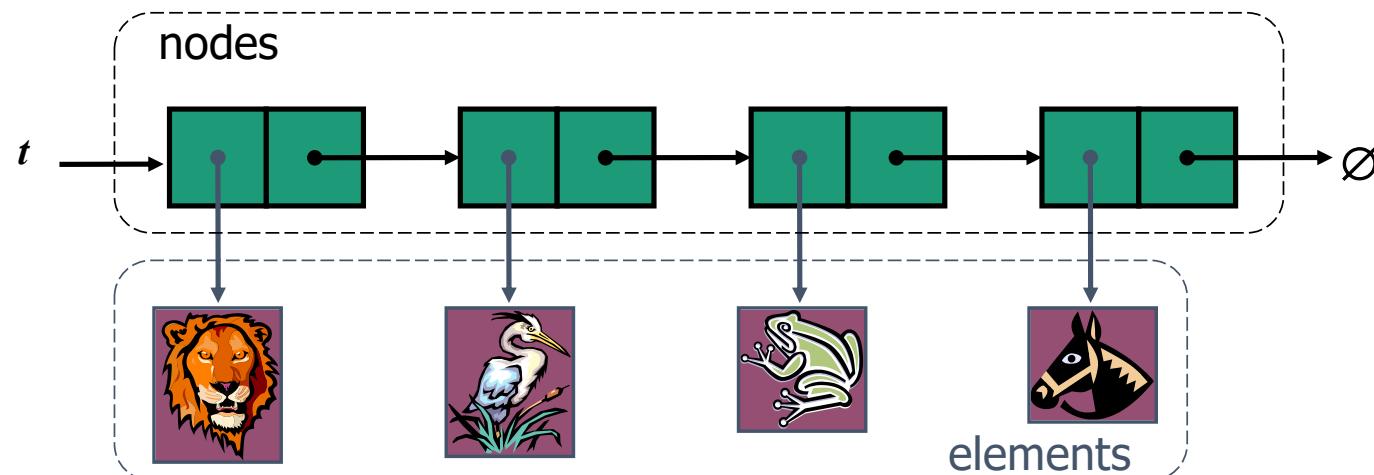
## Linked Queue



Can switch front & rear ?

# Stack as a Linked List

- ◆ We can implement a stack with a singly linked list
- ◆ The top element is stored at the first node of the list
- ◆ The space used is  $O(n)$  and each operation of the Stack ADT takes  $O(1)$  time



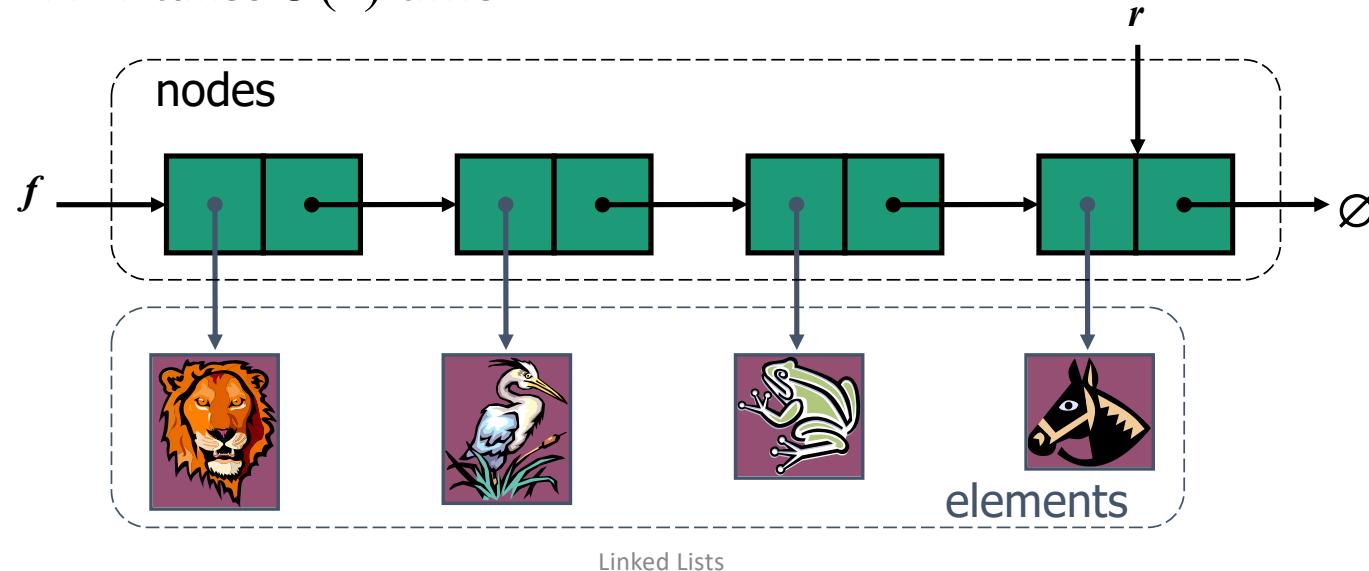
# Linked-List Stack in Python

```
1 class LinkedStack:
2     """LIFO Stack implementation using a singly linked list for storage."""
3
4     #----- nested _Node class -----
5     class _Node:
6         """Lightweight, nonpublic class for storing a singly linked node."""
7         __slots__ = '_element', '_next'      # streamline memory usage
8
9         def __init__(self, element, next):    # initialize node's fields
10            self._element = element          # reference to user's element
11            self._next = next                # reference to next node
12
13     #----- stack methods -----
14     def __init__(self):
15         """Create an empty stack."""
16         self._head = None                 # reference to the head node
17         self._size = 0                    # number of stack elements
18
19     def __len__(self):
20         """Return the number of elements in the stack."""
21         return self._size
22
```

```
23     def is_empty(self):
24         """Return True if the stack is empty."""
25         return self._size == 0
26
27     def push(self, e):
28         """Add element e to the top of the stack."""
29         self._head = self._Node(e, self._head)    # create and link a new node
30         self._size += 1
31
32     def top(self):
33         """Return (but do not remove) the element at the top of the stack.
34
35         Raise Empty exception if the stack is empty.
36         """
37
38         if self.is_empty():
39             raise Empty('Stack is empty')
40             return self._head._element           # top of stack is at head of list
41
42
43     def pop(self):
44         """Remove and return the element from the top of the stack (i.e., LIFO).
45
46         Raise Empty exception if the stack is empty.
47         """
48
49         if self.is_empty():
50             raise Empty('Stack is empty')
51             answer = self._head._element
52             self._head = self._head._next        # bypass the former top node
53             self._size -= 1
54             return answer
```

# Queue as a Linked List

- ◆ We can implement a queue with a singly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- ◆ The space used is  $O(n)$  and each operation of the Queue ADT takes  $O(1)$  time



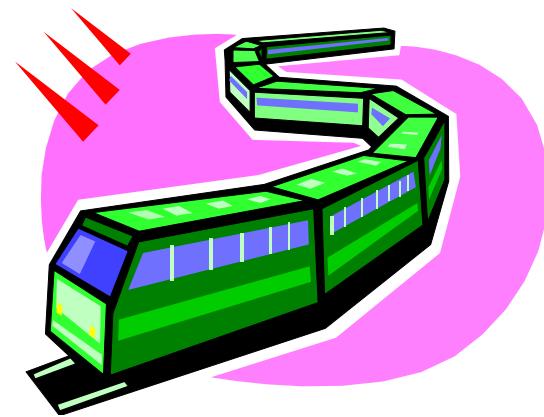
# Linked-List Queue in Python

```
1 class LinkedQueue:
2     """FIFO queue implementation using a singly linked list for storage."""
3
4     class _Node:
5         """Lightweight, nonpublic class for storing a singly linked node."""
6         (omitted here; identical to that of LinkedStack._Node)
7
8     def __init__(self):
9         """Create an empty queue."""
10        self._head = None
11        self._tail = None
12        self._size = 0                      # number of queue elements
13
14    def __len__(self):
15        """Return the number of elements in the queue."""
16        return self._size
17
18    def is_empty(self):
19        """Return True if the queue is empty."""
20        return self._size == 0
21
22    def first(self):
23        """Return (but do not remove) the element at the front of the queue."""
24        if self.is_empty():
25            raise Empty('Queue is empty')
26        return self._head._element          # front aligned with head of list
```

# Linked-List Queue in Python

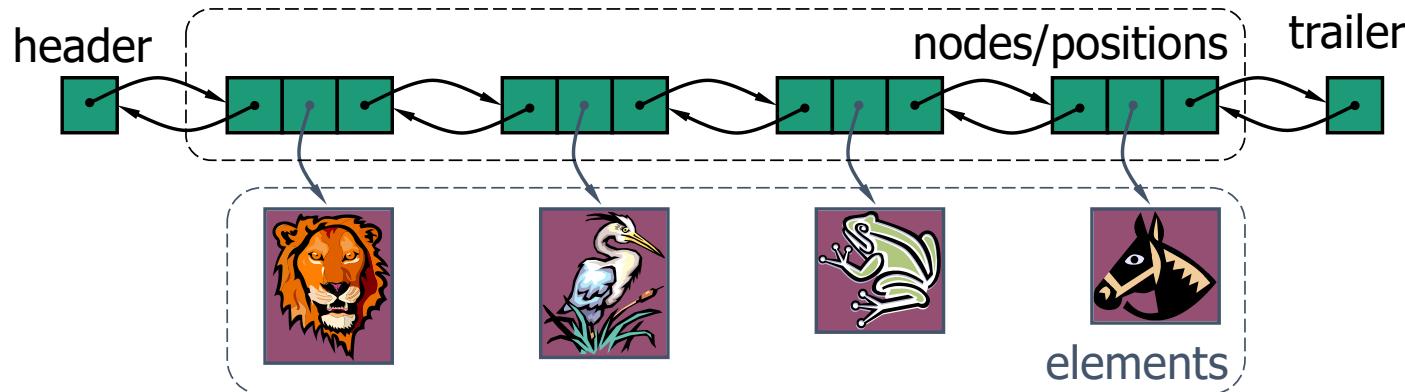
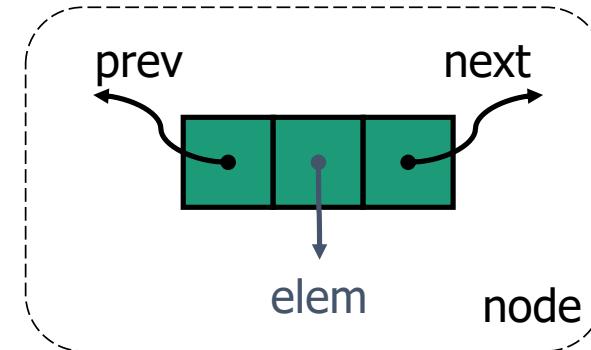
```
27 def dequeue(self):
28     """Remove and return the first element of the queue (i.e., FIFO).
29
30     Raise Empty exception if the queue is empty.
31     """
32     if self.is_empty():
33         raise Empty('Queue is empty')
34     answer = self._head._element
35     self._head = self._head._next
36     self._size -= 1
37     if self.is_empty():          # special case as queue is empty
38         self._tail = None        # removed head had been the tail
39     return answer
40
41 def enqueue(self, e):
42     """Add an element to the back of queue."""
43     newest = self._Node(e, None)      # node will be new tail node
44     if self.is_empty():
45         self._head = newest          # special case: previously empty
46     else:
47         self._tail._next = newest
48     self._tail = newest            # update reference to tail node
49     self._size += 1
```

# Doubly-Linked Lists



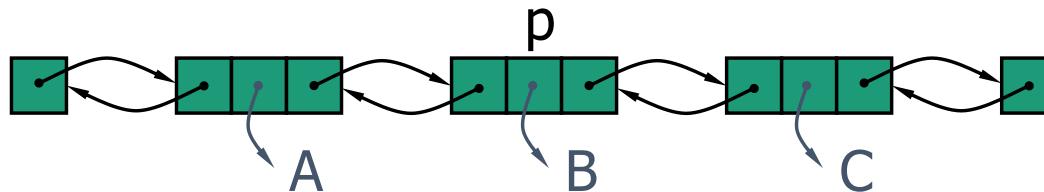
# Doubly Linked List

- A doubly linked list provides a natural implementation of the Node List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes



# Insertion

- Insert a new node,  $q$ , between  $p$  and its successor.

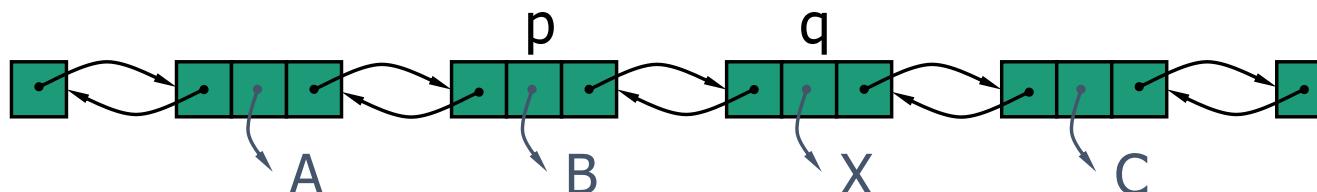
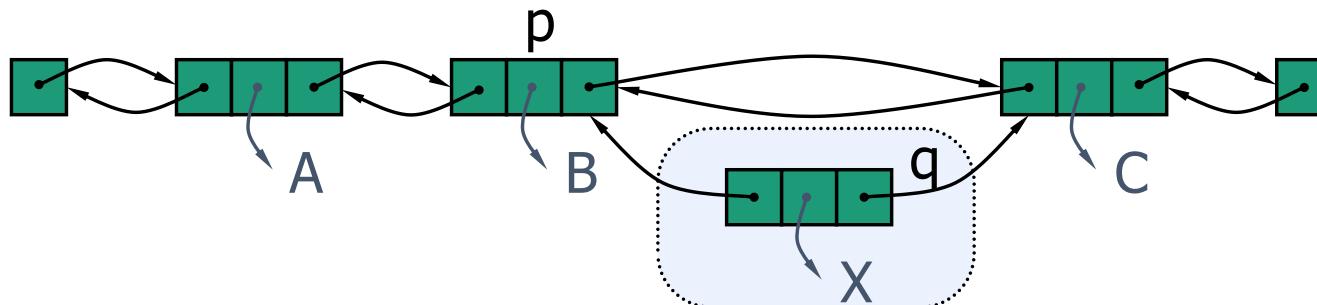


$q = \text{node}('X')$

$q.\text{next} = p.\text{next}$

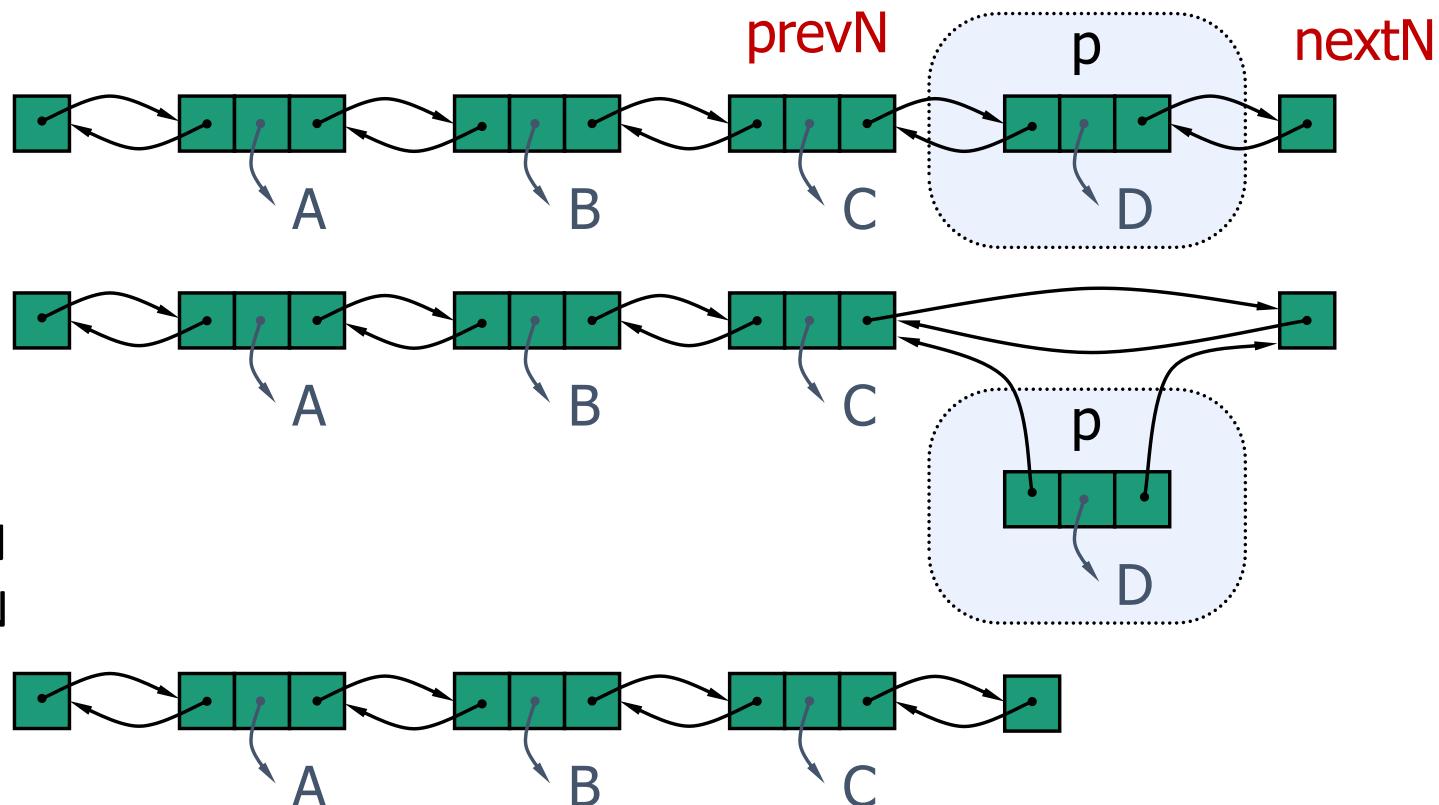
$q.\text{prev} = p$

$p.\text{next} = q$



# Deletion

- Remove a node,  $p$ , from a doubly-linked list.



$\text{prevN} = p.\text{prev}$

$\text{nextN} = p.\text{next}$

$\text{prevN}.\text{next} = \text{nextN}$

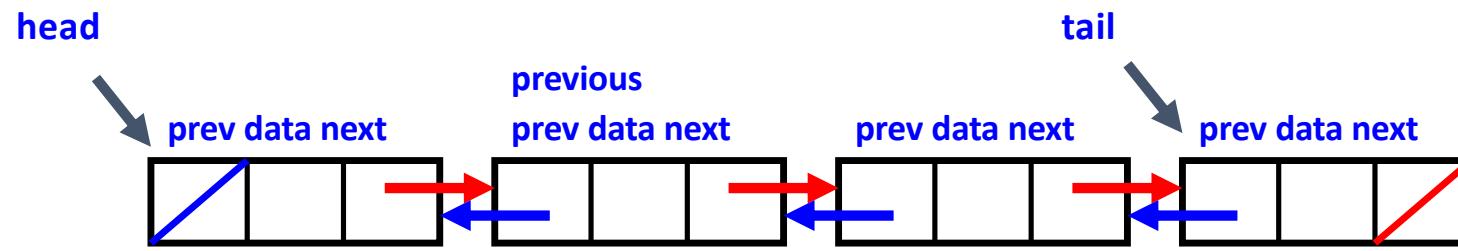
$\text{nextN}.\text{prev} = \text{prevN}$

# Doubly-Linked List with header node

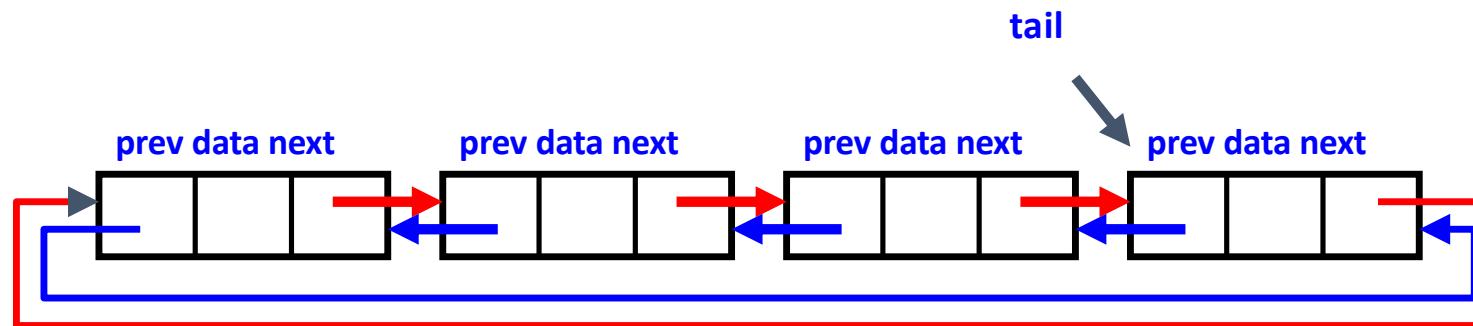
```
1 class _DoublyLinkedListBase:  
2     """A base class providing a doubly linked list representation."""  
3  
4 class _Node:  
5     """Lightweight, nonpublic class for storing a doubly linked node."""  
6     (omitted here; see previous code fragment)  
7  
8 def __init__(self):  
9     """Create an empty list."""  
10    self._header = self._Node(None, None, None)  
11    self._trailer = self._Node(None, None, None)  
12    self._header._next = self._trailer           # trailer is after header  
13    self._trailer._prev = self._header          # header is before trailer  
14    self._size = 0                             # number of elements  
15  
16 def __len__(self):  
17     """Return the number of elements in the list."""  
18     return self._size  
19  
20 def is_empty(self):  
21     """Return True if list is empty."""  
22     return self._size == 0  
23
```

```
class _Node:  
    """Lightweight, nonpublic class for storing a doubly linked node."""  
    __slots__ = '_element', '_prev', '_next' # streamline memory  
  
    def __init__(self, element, prev, next):      # initialize node's fields  
        self._element = element                   # user's element  
        self._prev = prev                        # previous node reference  
        self._next = next                        # next node reference  
  
24    def _insert_between(self, e, predecessor, successor):  
25        """Add element e between two existing nodes and return new node."""  
26        newest = self._Node(e, predecessor, successor) # linked to neighbors  
27        predecessor._next = newest  
28        successor._prev = newest  
29        self._size += 1  
30        return newest  
31  
32    def _delete_node(self, node):  
33        """Delete nonsentinel node from the list and return its element."""  
34        predecessor = node._prev  
35        successor = node._next  
36        predecessor._next = successor  
37        successor._prev = predecessor  
38        self._size -= 1  
39        element = node._element                # record deleted element  
40        node._prev = node._next = node._element = None # deprecate node  
41        return element                         # return deleted element
```

## Doubly Linked List



## Doubly Circular List



## Applications

- Polynomial Expression
- Multilists

## Polynomial expression

$$A = 5x^3 + 4x^2 - 7x + 10$$

+

$$B = x^3 + 2x - 8$$

---

$$C = 6x^3 + 4x^2 - 5x + 2$$

How about ... ?

$$5x^{85} + 7x + 1$$

+

$$x^{76} - 8$$

---

What data structure will you use?

Array?

Array?

Sparse  $\rightarrow$  Linked List

( has lots of 0 data )

## Multilists

1. class หนึ่งๆ มีโครงสร้างแบบ 2. นร. คนหนึ่งๆ ลง class ได้บ้าง

