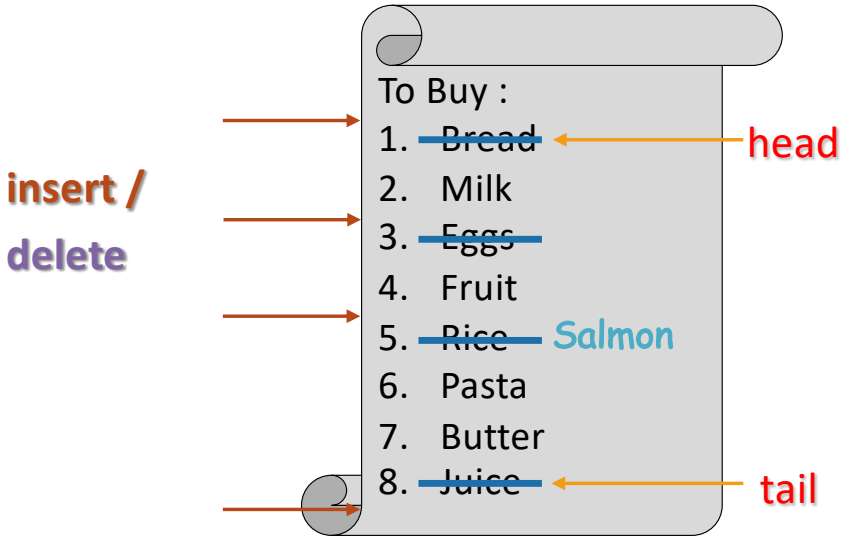
Linked list

Kiatnarong Tongprasert

List

http://www.gograph.com/vector-clip-art/queue.html

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

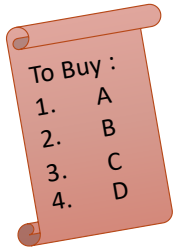
- append(item) adds item ท้าย list ไม่ Returns คิดว่า item ไม่มีอยู่ก่อนใน list
- insert(pos,item) adds item ที่ index pos ไม่ Returns คิดว่า item ไม่มีอยู่ก่อนใน list

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

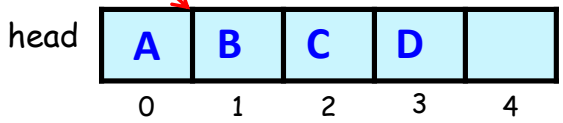
Problem : fix positions

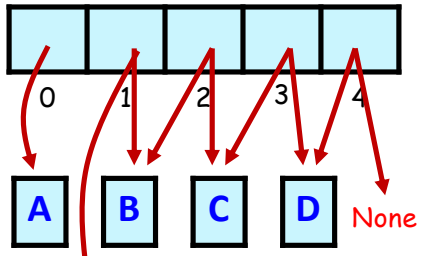
i

Sequential Array

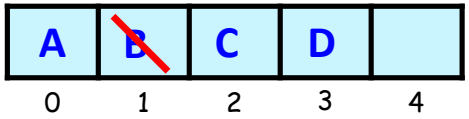
Python : List

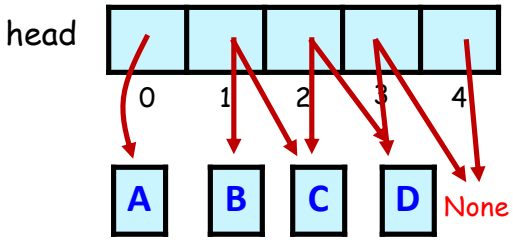
insert i ? : shift out



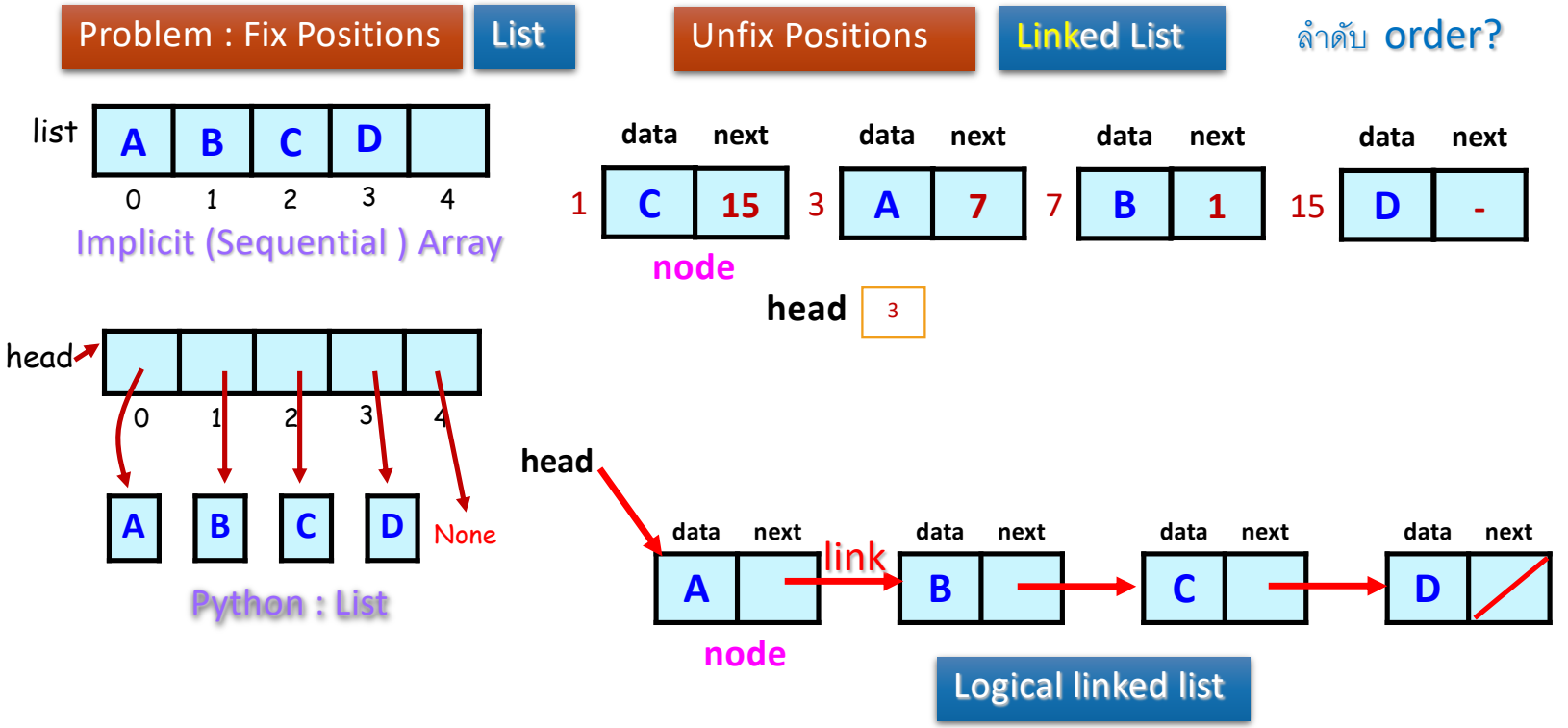


delete : shift in





Linked List



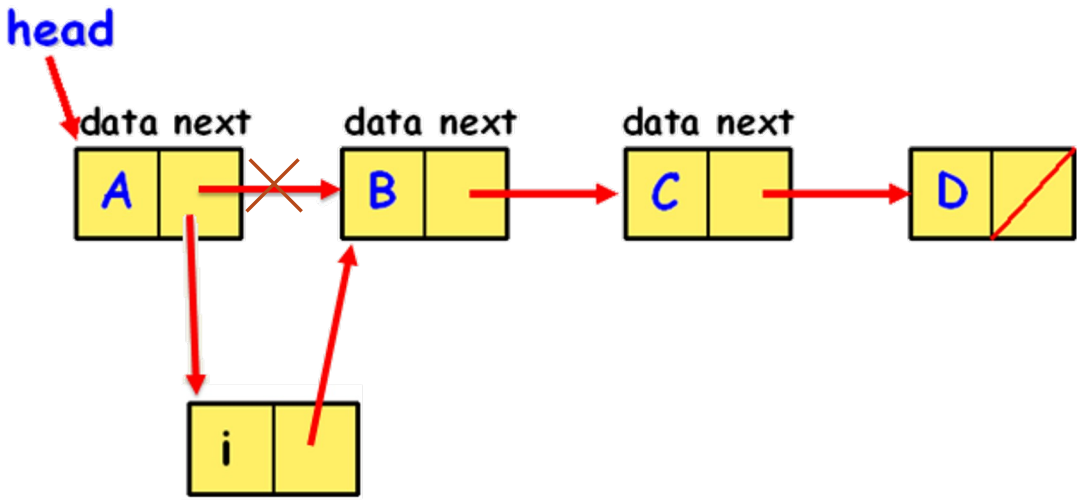
Logical คือในความคิดของเรา

เช่น link แทนด้วยลูกศร แทนการเชื่อมโยงกัน

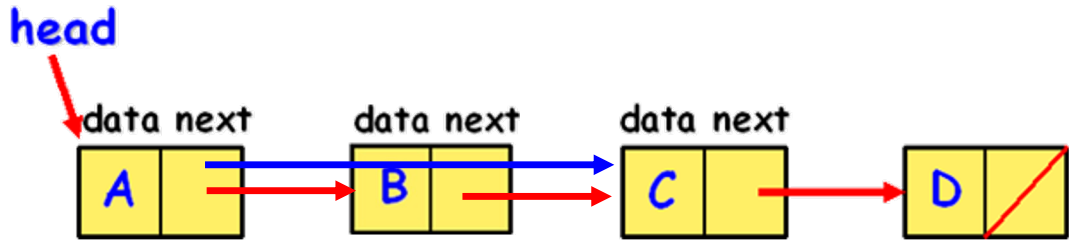
physical (implementation) โครงสร้างที่ใช้ในการสร้างจริง

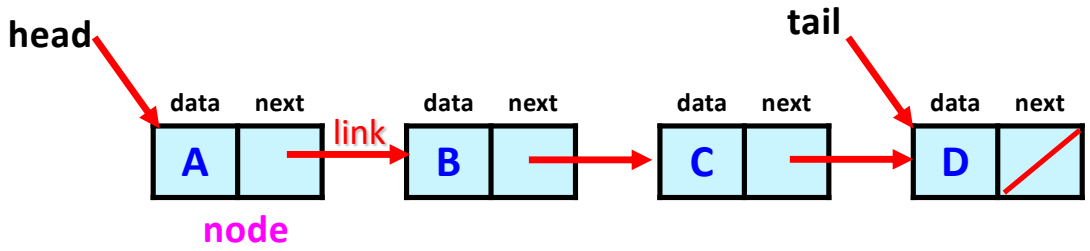
เช่น link อาจใช้ pointer หรือ index ของ array

Solve Inserting Expensive Shifting Problem



Solve Deleting Expensive Shifting Problem





Linked List

Data :

1. data
2. next (link)

} node class

3. head
4. tail ?

} list class

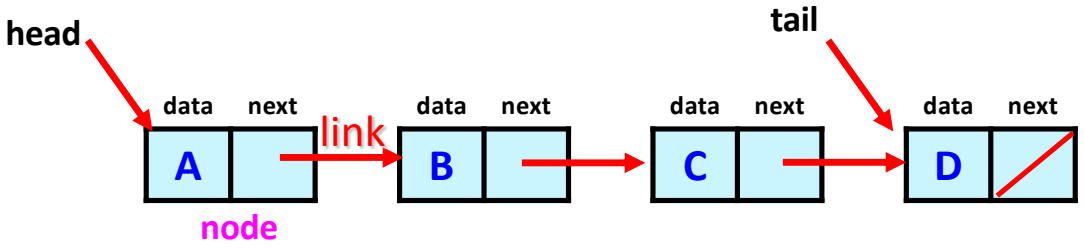
Node Class / List Class

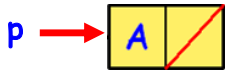
1. Data :

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

2 underscores 2 underscores

Node Class






```
p =node('A', None)
```

```
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):
        self.head = None
```

```
    """ unordered singly linked list
        with head & tail
    """
    def __init__(self):
        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:
        self.head = self.tail = None
        self.size = 0
    else:
        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
```

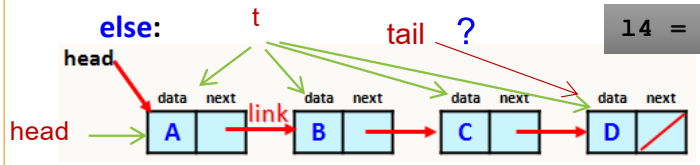
l1 = list()

l2 = list()

l3 = list()

head
tail
None

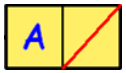
l4 = list(head)



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. . . .

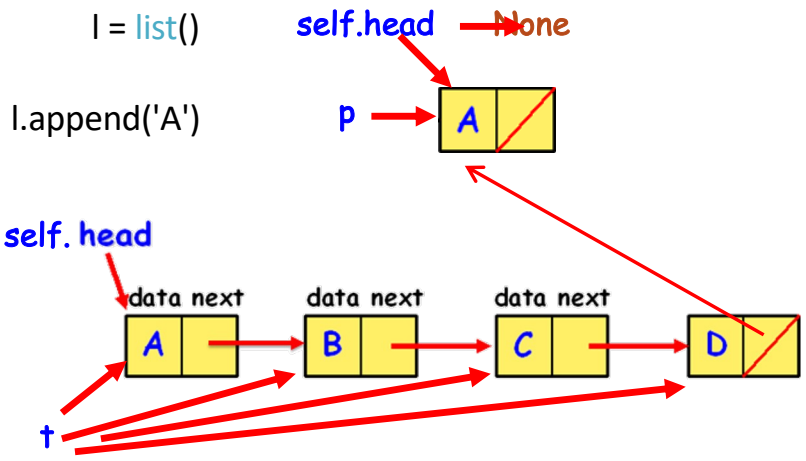
Creating a List



node('A', None)

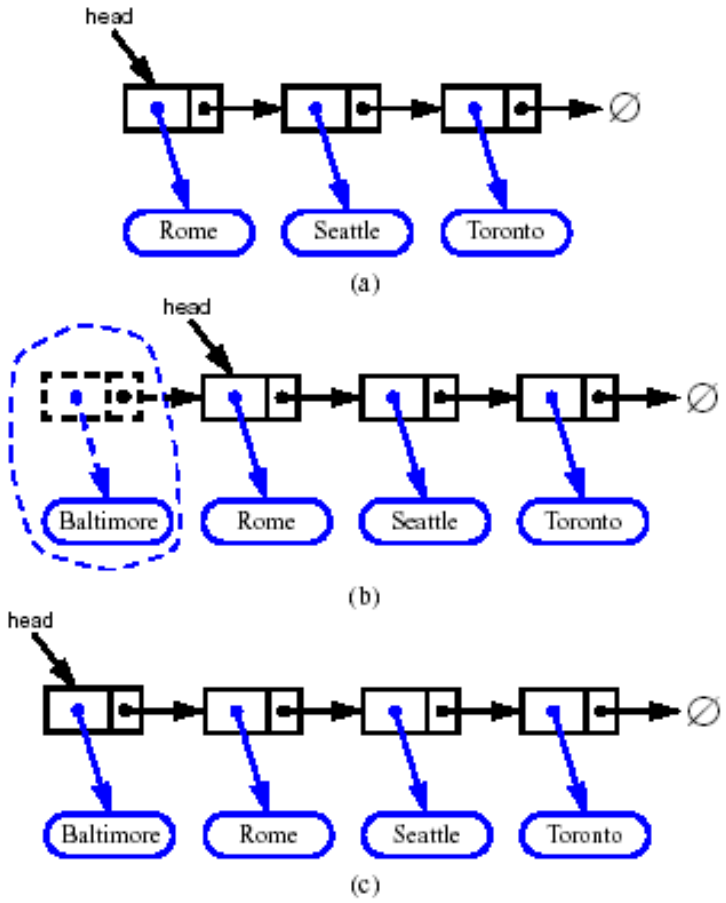
```
class node:
   def __init__(self, data, next = None):
      self.data = data
      if next == None:
         self.next = None
      else:
         self.next = next
```

```
class list:

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



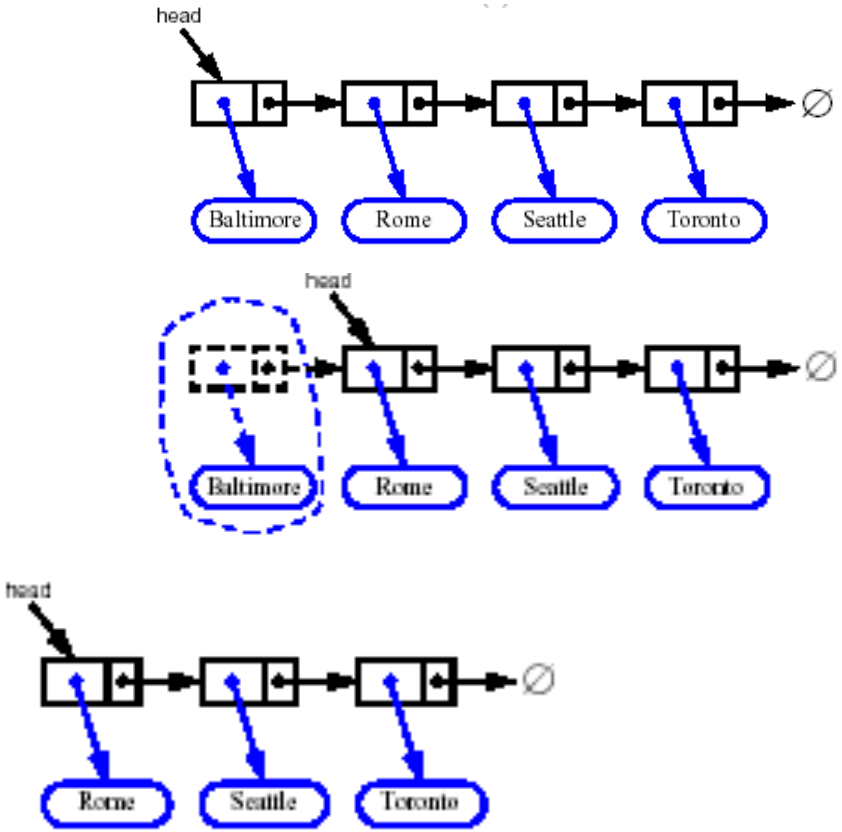
Inserting at the Head

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



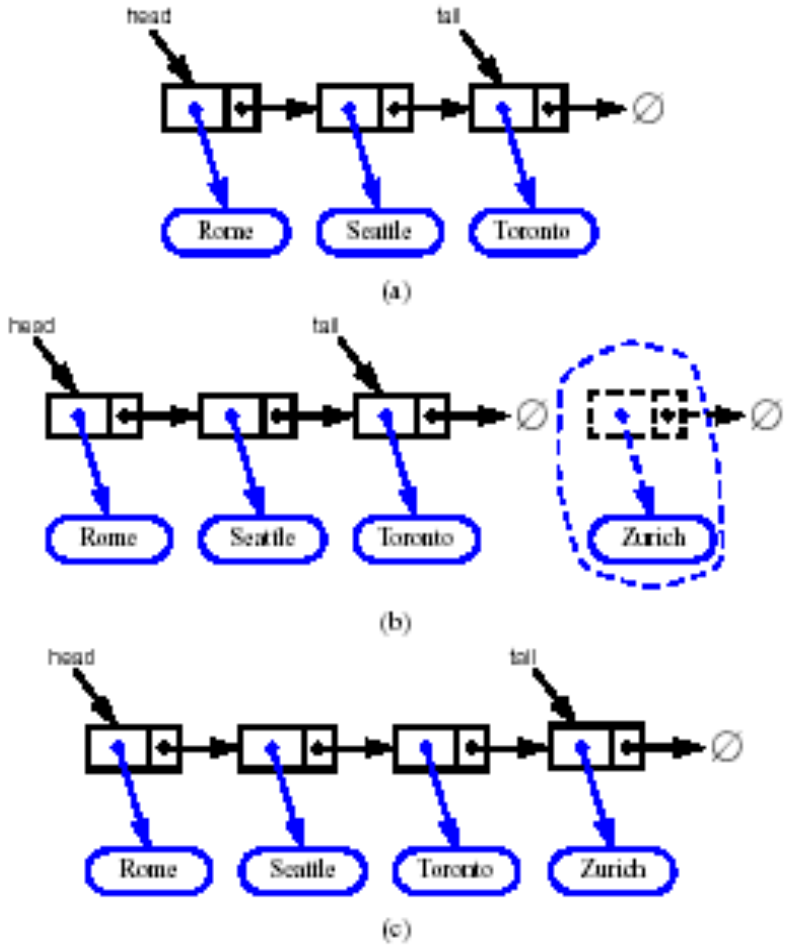
Removing at the Head

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



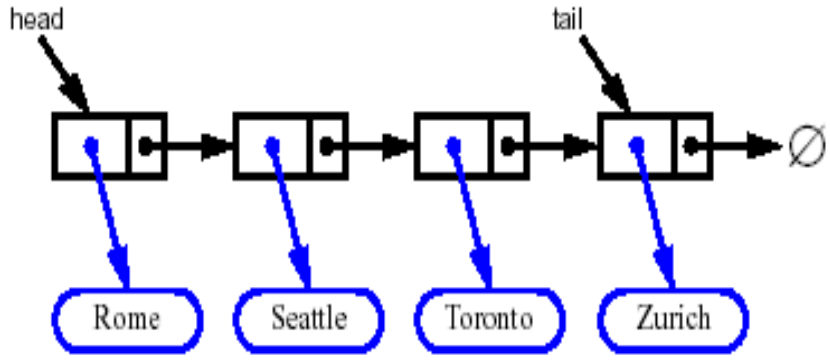
Inserting at the Tail

1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node

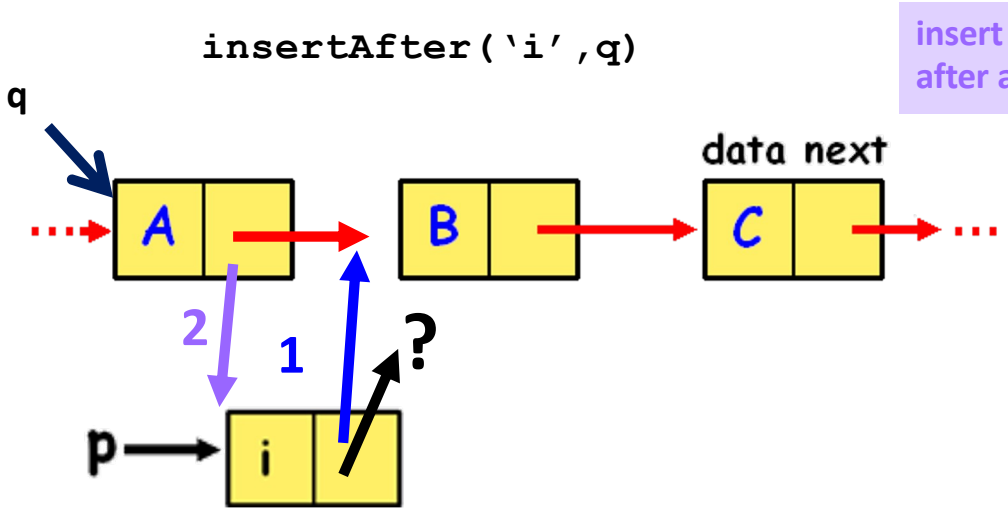


Removing at the Tail

- 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



Insert After



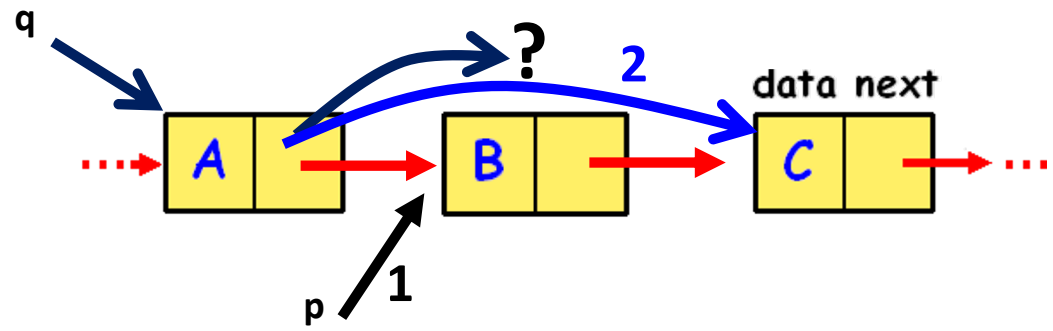
insert node data
after a node pointed by q

Why insert after ?
Can you insert before ?

Delete After

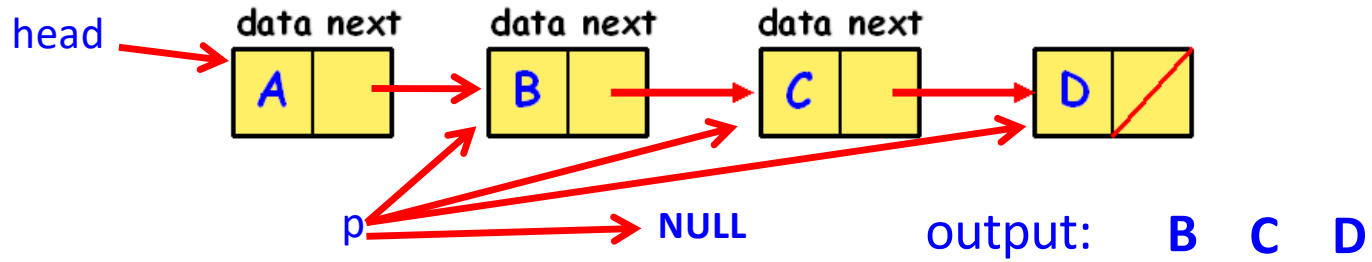
`deleteAfter(q)`

delete a node
after a node pointed by q



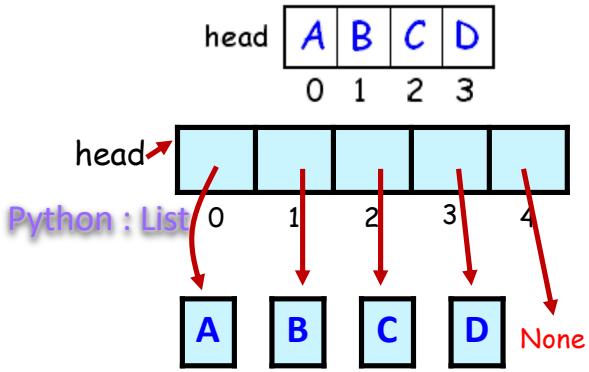
print list

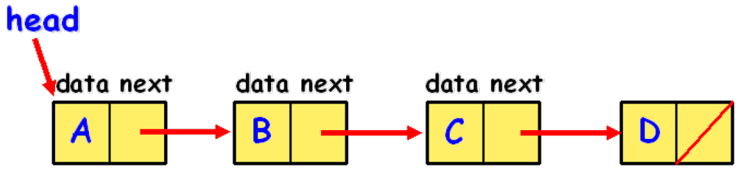
Design how to call.




```
          p is not None:
while p != None:
    print(p.data)
    p = p.next
}
```

Linked List VS Sequential Array

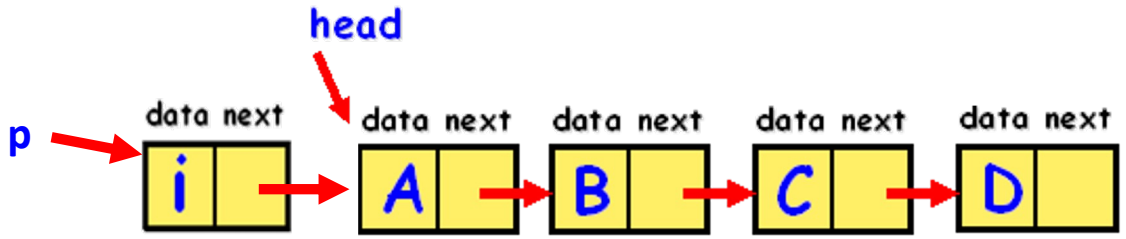




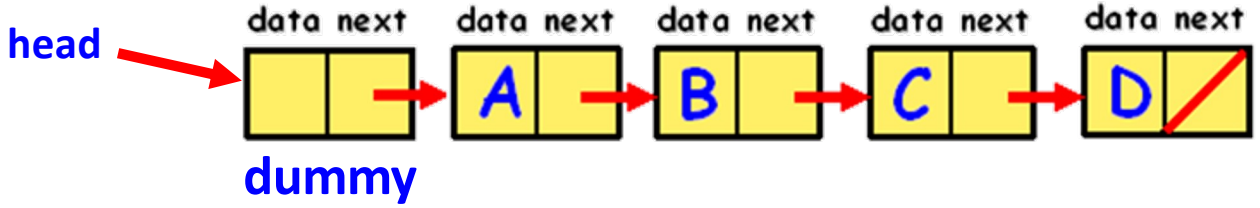
Sequential Array	Linked List
• Insertion / Deletion Shifting Problem. • Random Access. • C array : Automatic Allocation. Python List array : Dynamic Allocation • Lifetime : C-array, Python List • from defined until its scope finishes. • Only keeps data.	• Solved. • Sequential Access. • Node : Dynamic Allocation. • Node Lifetime : from allocated (C : malloc()/new, python: instantiate obj) until C: deallocated by free()/delete, Python : no reference. • Need spaces for linkes.

Dummy Node

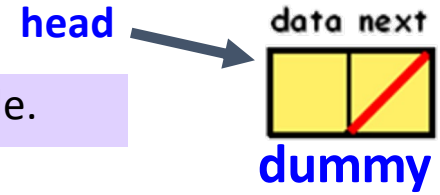
To insert & delete at 1st position change head ie. make special case.



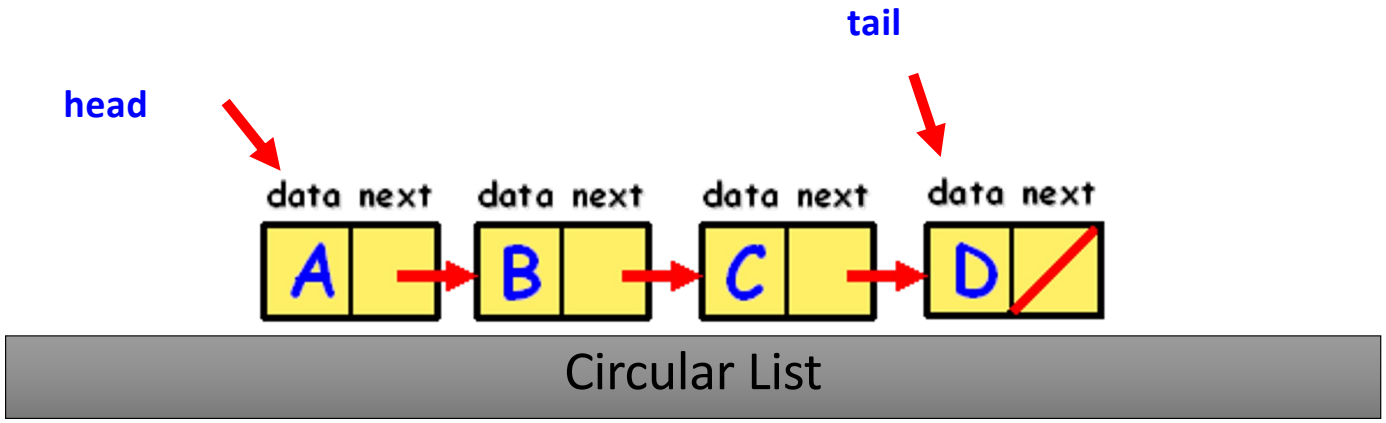
"Dummy Node" solves the problem.

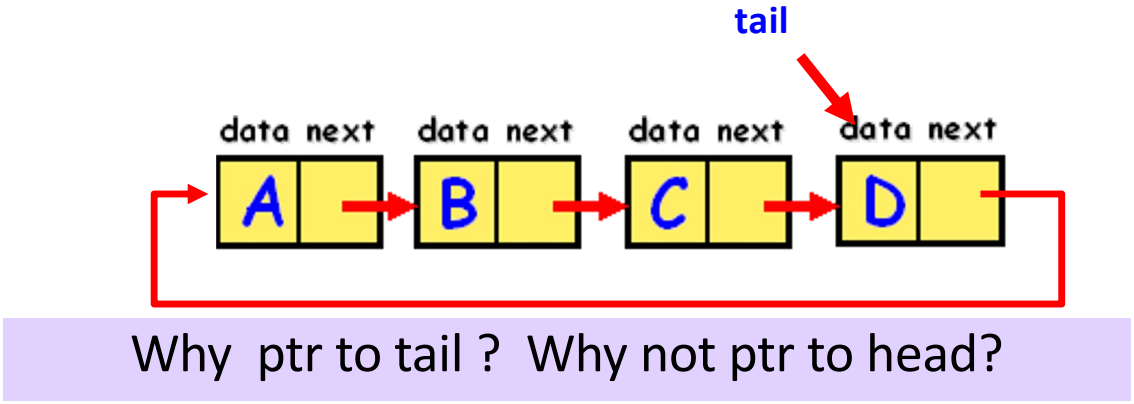


Empty List has a dummy node.



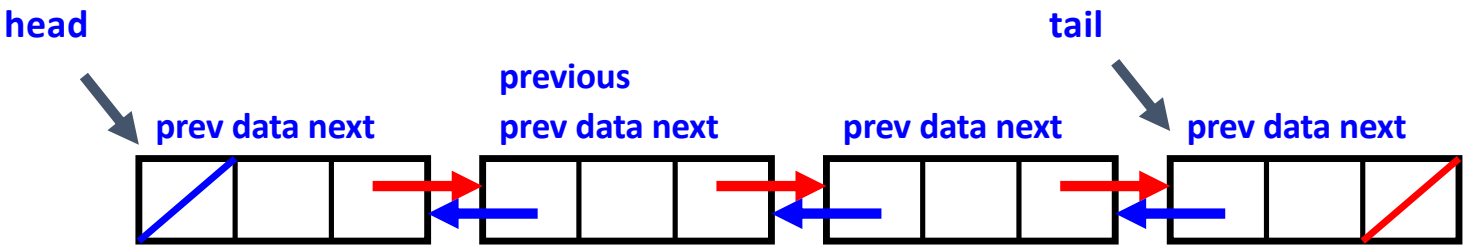
Head & Tail Nodes



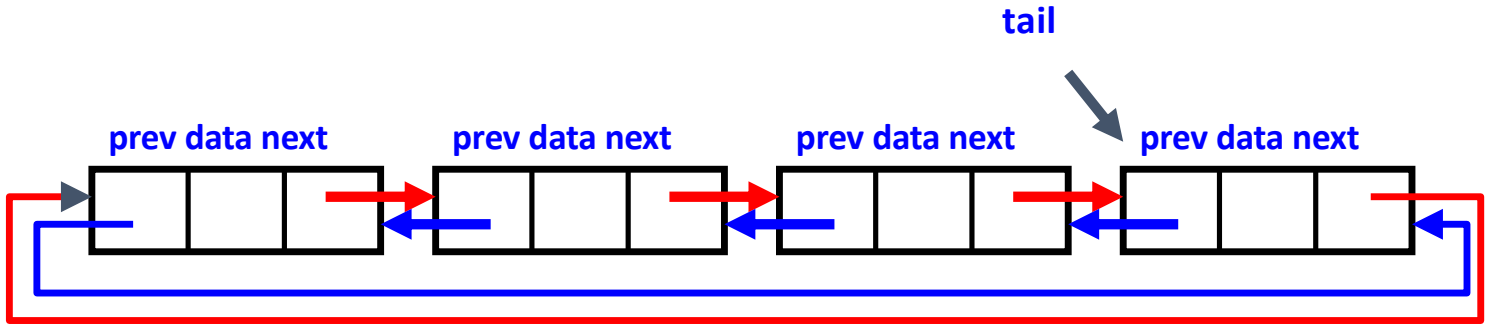


Why ptr to tail ? Why not ptr to head?

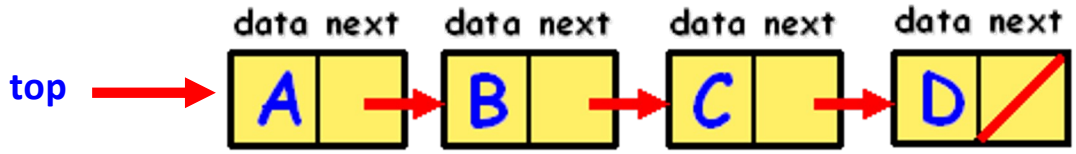
Doubly VS Singly Linked List



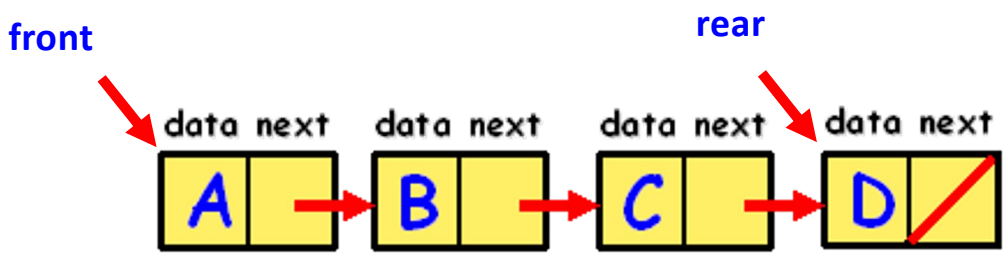
Doubly Circular List



Linked Stack

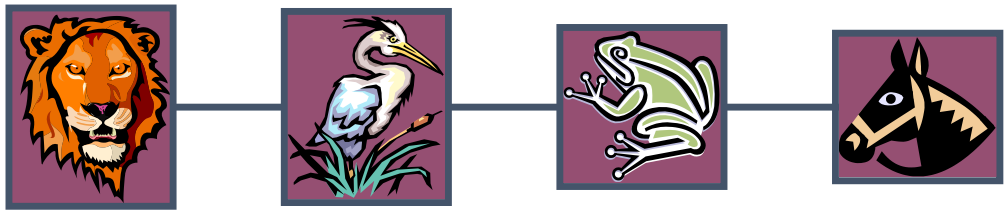


Linked Queue



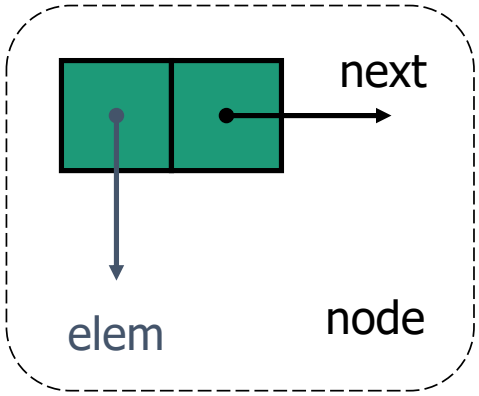
Can switch front & rear ?

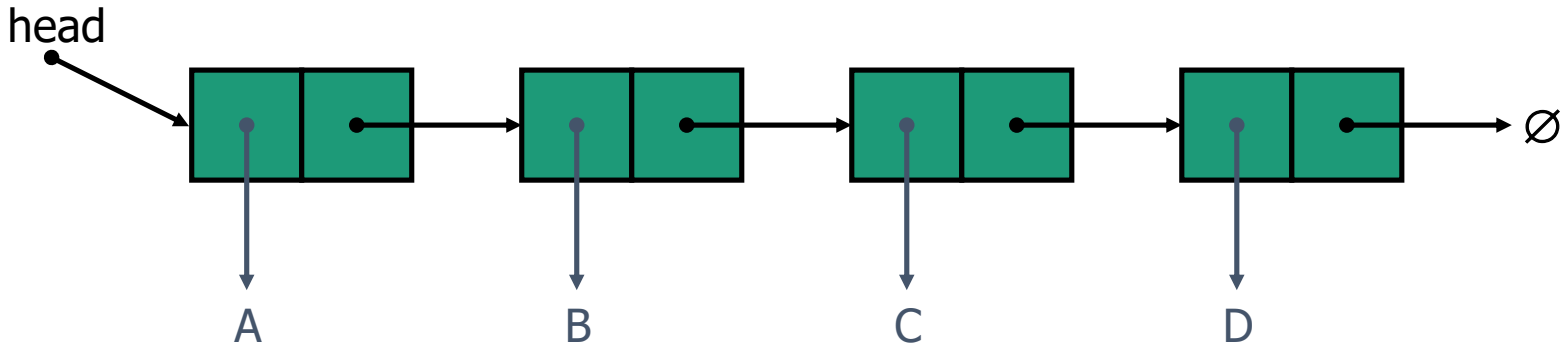
Linked Lists



Singly Linked List

- **A singly linked list** คือ โครงสร้างข้อมูลที่ประกอบด้วยลำดับของโหนด ที่เริ่มต้นด้วย ตัวชี้โหนดแรก (ตัวชี้หัว)
- **Each node stores** ประกอบด้วย
  - ข้อมูล
  - ตัวโยงไปยังโหนดถัดไป





The Node Class for List Nodes

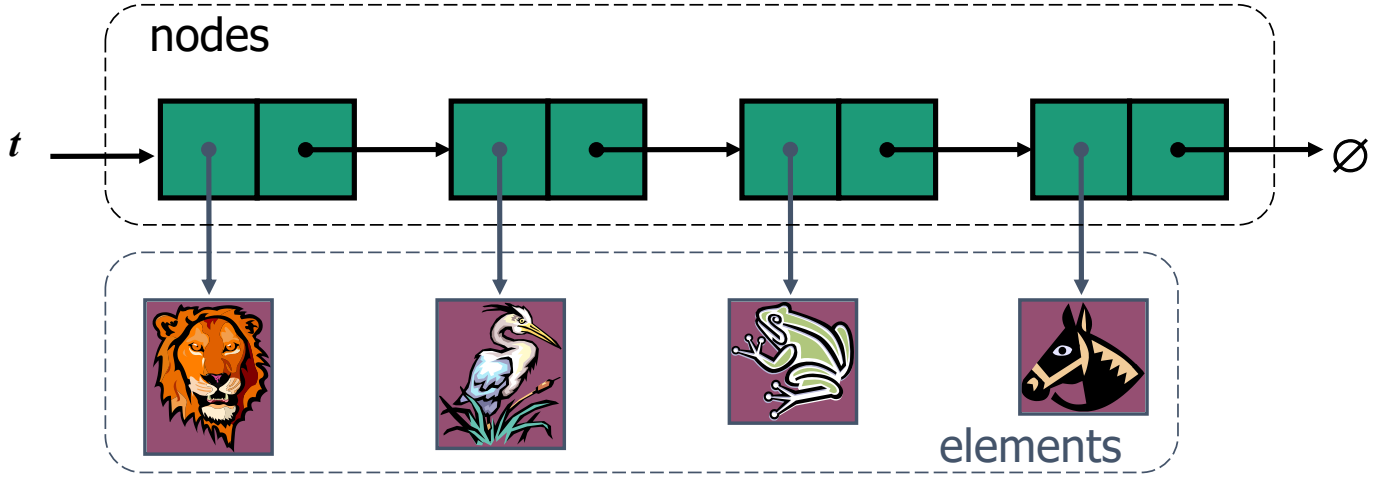
```
public class Node  {
   // Instance variables:
    private  Object element;
    private  Node next;
   /** Creates a node with null references to its element and
   next node. */
    public  Node()  {
       this(null,  null);
  }
   /** Creates a node with the given element and next node. */
    public  Node(Object e,  Node n)  {
       element  =  e;
       next  =  n;
  }
```



```
// Accessor methods:
    public  Object getElement()  {
       return  element;
  }
    public  Node getNext()  {
       return  next;
  }
    // Modifier methods:
    public void  setElement(Object newElem)  {
         element  =  newElem;
  }
    public void  setNext(Node newNext)  {
         next  =  newNext;
  }
}
```

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

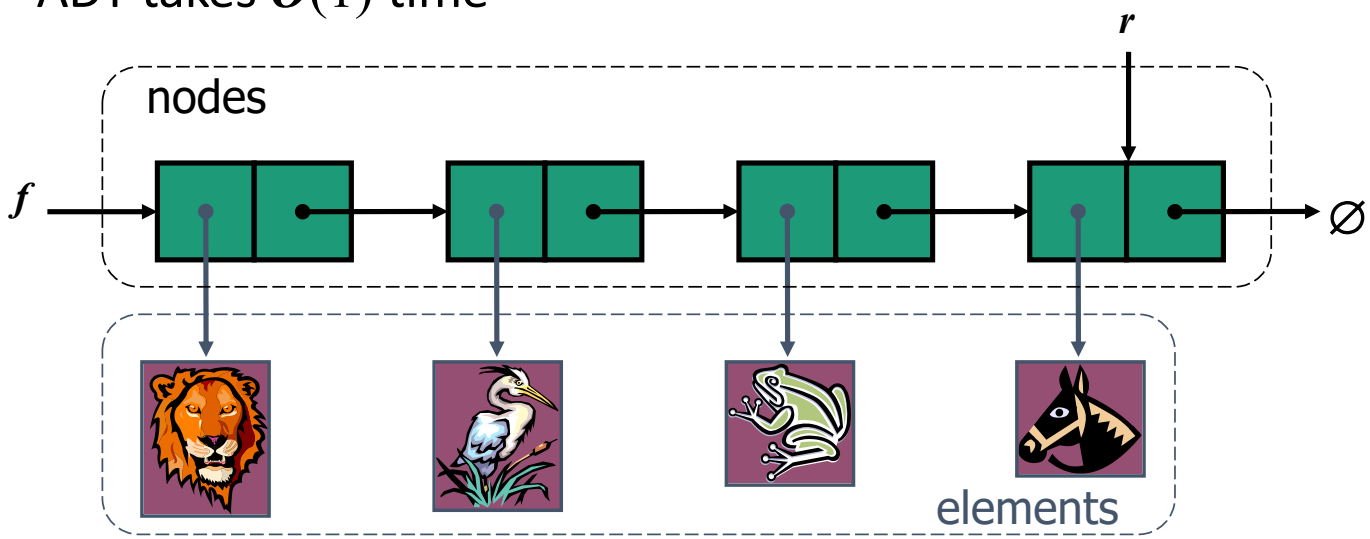
```
1  class LinkedStack:
2    """LIFO Stack implementation using a singly linked list for storage."""
3
4    #---------------------- nested _Node class ----------------------
5    class _Node:
6      """Lightweight, nonpublic class for storing a singly linked node."""
7      __slots__ = '_element', '_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     if self.is_empty():
38       raise Empty('Stack is empty')
39     return self._head._element                 # top of stack is at head of list
```

```
40   def pop(self):
41     """Remove and return the element from the top of the stack (i.e., LIFO).
42
43     Raise Empty exception if the stack is empty.
44     """
45     if self.is_empty():
46       raise Empty('Stack is empty')
47     answer = self._head._element
48     self._head = self._head._next              # bypass the former top node
49     self._size -= 1
50     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



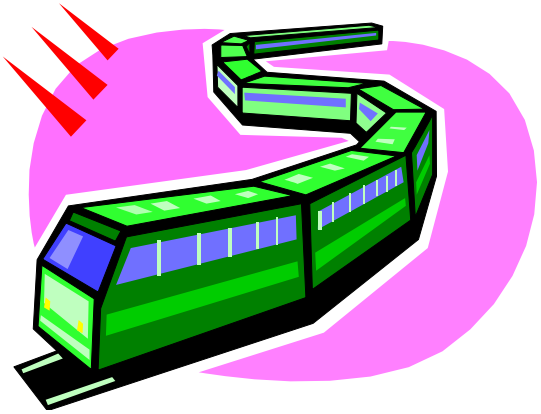
Linked-List Queue in Python

```
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

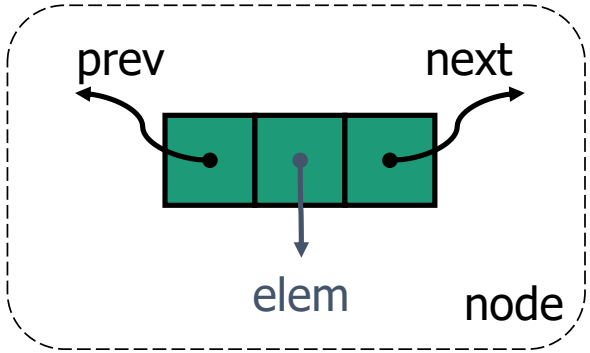
```
27    def dequeue(self):
28      """Remove and return the first element of the queue (i.e., FIFO).
29
30      Raise Empty exception if the queue is empty.
31      """
32      if self.is_empty():
33        raise Empty('Queue is empty')
34      answer = self._head._element
35      self._head = self._head._next
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
```

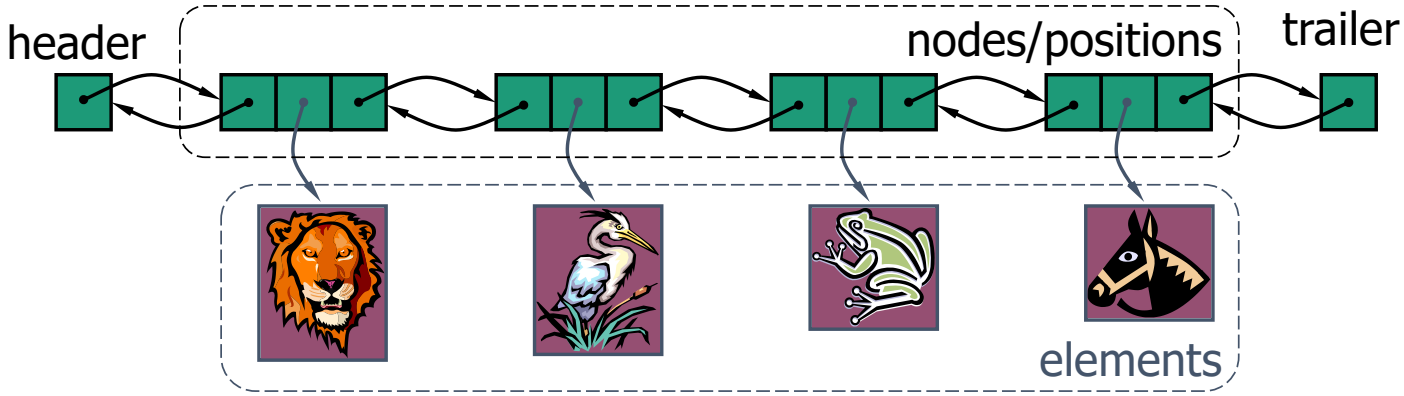
Doubly-Linked Lists



Doubly Linked List

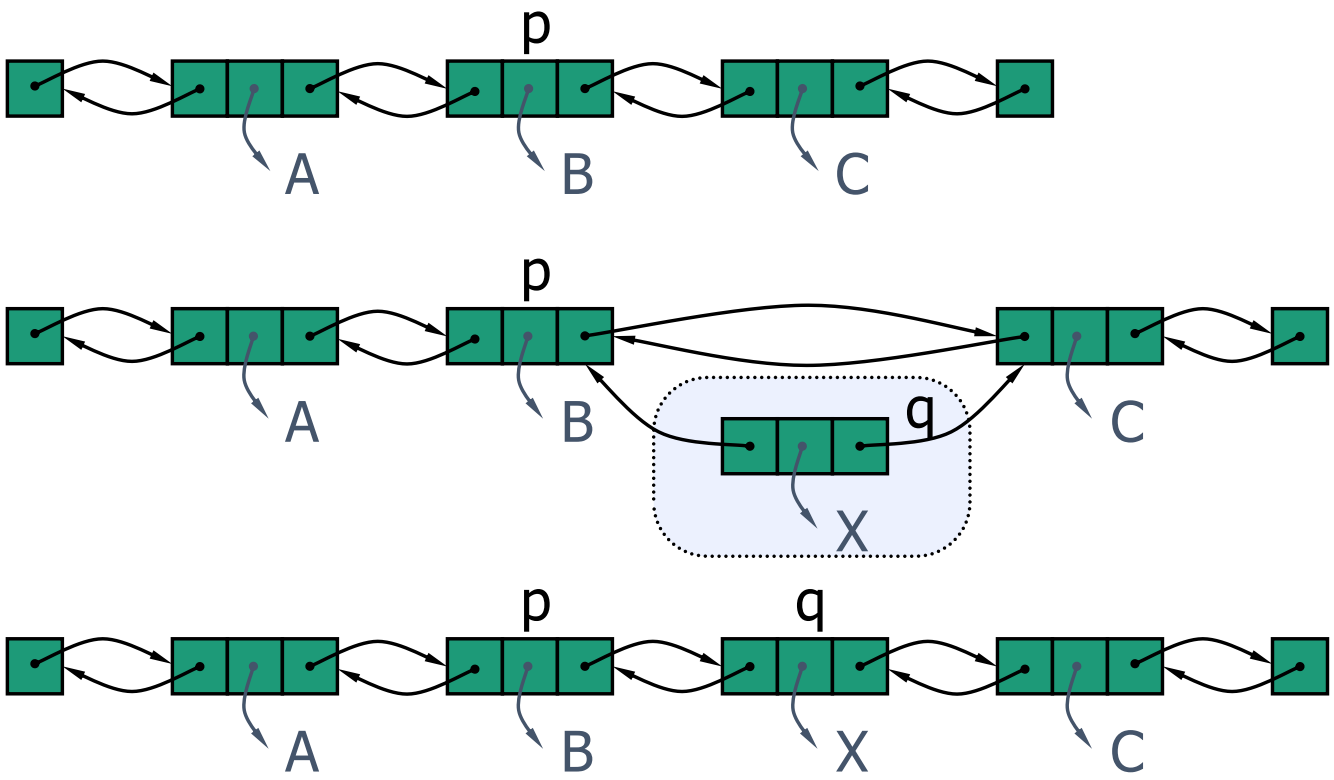
- A doubly linked list provides a natural implementation of the Node List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





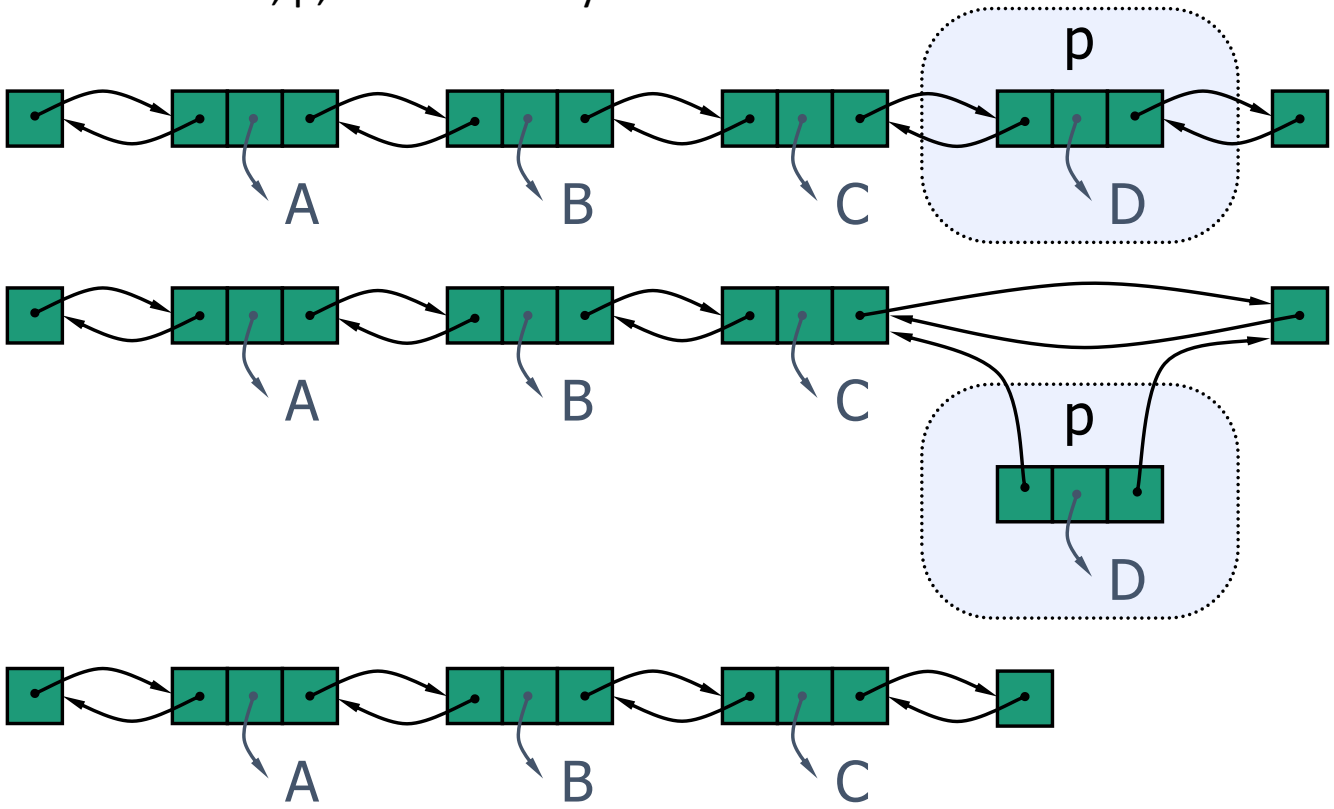
Insertion

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



Deletion

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



Doubly-Linked List in Python

```
1  class _DoublyLinkedBase:
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
```

```
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
```

Performance

- In a doubly linked list
  - The space used by a list with n elements is $O(n)$
  - The space used by each position of the list is $O(1)$
  - All the standard operations of a list run in $O(1)$ time

Positional List

- To provide for a general abstraction of a sequence of elements with the ability to identify the location of an element, we define a **positional list** ADT.
- A position acts as a marker or token within the broader positional list.
- A position p is unaffected by changes elsewhere in a list; the only way in which a position becomes invalid is if an explicit command is issued to delete it.
- A position instance is a simple object, supporting only the following method:
  - p.element( ): Return the element stored at position p.

Positional Accessor Operations

L.first( ):	Return the position of the first element of L, or None if L is empty.
L.last( ):	Return the position of the last element of L, or None if L is empty.
L.before(p):	Return the position of L immediately before position p, or None if p is the first position.
L.after(p):	Return the position of L immediately after position p, or None if p is the last position.
L.is_empty( ):	Return True if list L does not contain any elements.
len(L):	Return the number of elements in the list.
iter(L):	Return a forward iterator for the *elements* of the list. See Section 1.8 for discussion of iterators in Python.

Positional Update Operations

L.add_first(e):	Insert a new element e at the front of L, returning the position of the new element.
L.add_last(e):	Insert a new element e at the back of L, returning the position of the new element.
L.add_before(p, e):	Insert a new element e just before position p in L, returning the position of the new element.
L.add_after(p, e):	Insert a new element e just after position p in L, returning the position of the new element.
L.replace(p, e):	Replace the element at position p with element e, returning the element formerly at position p.
L.delete(p):	Remove and return the element at position p in L, invalidating the position.

Positional List in Python

```
1  class PositionalList(_DoublyLinkedBase):
2    """A sequential container of elements allowing positional access."""
3
4    #---------------------- nested Position class ----------------------
5    class Position:
6      """An abstraction representing the location of a single element."""
7
8      def __init__(self, container, node):
9        """Constructor should not be invoked by user."""
10       self._container = container
11       self._node = node
12
13     def element(self):
14       """Return the element stored at this Position."""
15       return self._node._element
16
17     def __eq__(self, other):
18       """Return True if other is a Position representing the same location."""
19       return type(other) is type(self) and other._node is self._node
```

Positional List in Python Part 2

```
20
21      def __ne__(self, other):
22        """Return True if other does not represent the same location."""
23        return not (self == other)          # opposite of __eq__
24
25    #---------------------------- utility method -----------------------------
26    def _validate(self, p):
27      """Return position's node, or raise appropriate error if invalid."""
28      if not isinstance(p, self.Position):
29        raise TypeError('p must be proper Position type')
30      if p._container is not self:
31        raise ValueError('p does not belong to this container')
32      if p._node._next is None:             # convention for deprecated nodes
33        raise ValueError('p is no longer valid')
34      return p._node
```

Positional List in Python, Part 3

```
35    #------------------------------ utility method -------------------------------
36    def _make_position(self, node):
37      """Return Position instance for given node (or None if sentinel)."""
38      if node is self._header or node is self._trailer:
39        return None                                  # boundary violation
40      else:
41        return self.Position(self, node)             # legitimate position
42
43    #------------------------------ accessors ------------------------------
44    def first(self):
45      """Return the first Position in the list (or None if list is empty)."""
46      return self._make_position(self._header._next)
47
48    def last(self):
49      """Return the last Position in the list (or None if list is empty)."""
50      return self._make_position(self._trailer._prev)
51
```

Positional List in Python, Part 4

```
52    def before(self, p):
53      """Return the Position just before Position p (or None if p is first)."""
54      node = self._validate(p)
55      return self._make_position(node._prev)
56
57    def after(self, p):
58      """Return the Position just after Position p (or None if p is last)."""
59      node = self._validate(p)
60      return self._make_position(node._next)
61
62    def __iter__(self):
63      """Generate a forward iteration of the elements of the list."""
64      cursor = self.first( )
65      while cursor is not None:
66        yield cursor.element( )
67        cursor = self.after(cursor)
```

Positional List in Python, Part 5

```
68    #---------------------------- mutators ----------------------------
69    # override inherited version to return Position, rather than Node
70    def _insert_between(self, e, predecessor, successor):
71      """Add element between existing nodes and return new Position."""
72      node = super()._insert_between(e, predecessor, successor)
73      return self._make_position(node)
74
75    def add_first(self, e):
76      """Insert element e at the front of the list and return new Position."""
77      return self._insert_between(e, self._header, self._header._next)
78
79    def add_last(self, e):
80      """Insert element e at the back of the list and return new Position."""
81      return self._insert_between(e, self._trailer._prev, self._trailer)
82
83    def add_before(self, p, e):
84      """Insert element e into list before Position p and return new Position."""
85      original = self._validate(p)
86      return self._insert_between(e, original._prev, original)
87
```

Positional List in Python, Part 6

```
88    def add_after(self, p, e):
89      """Insert element e into list after Position p and return new Position."""
90      original = self._validate(p)
91      return self._insert_between(e, original, original._next)
92
93    def delete(self, p):
94      """Remove and return the element at Position p."""
95      original = self._validate(p)
96      return self._delete_node(original)      # inherited method returns element
97
98    def replace(self, p, e):
99      """Replace the element at Position p with e.
100
101     Return the element formerly at Position p.
102     """
103     original = self._validate(p)
104     old_value = original._element          # temporarily store old element
105     original._element = e                  # replace with new element
106     return old_value                       # return the old element value
```

Applications

- Polynomial Expression
- Multilists
- Radix Sort

Polynomial expression

How about ... ?

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

$+$

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

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

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

$+$

$x^{76} - 8$

What data structure will you use?

Array?

Array?

Sparse -> Linked List

(has lots of 0 data)

1. class หนึ่งๆ มีใครลงบ้าง 2. นร. คนหนึ่งๆ ลง class ใดบ้าง