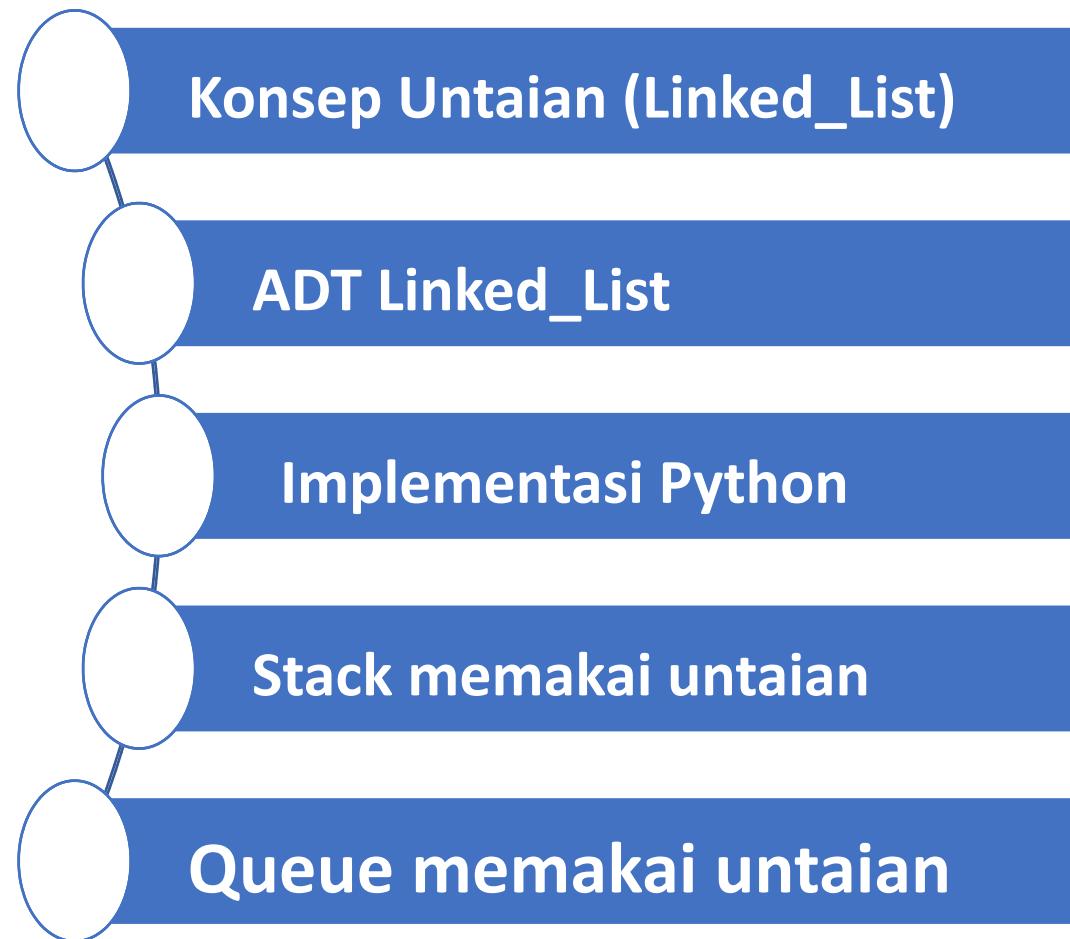


STRUKTUR DATA (PYTHON)

“Struktur Untaian Tunggal”

[@SUARGA | [Pertemuan 09]

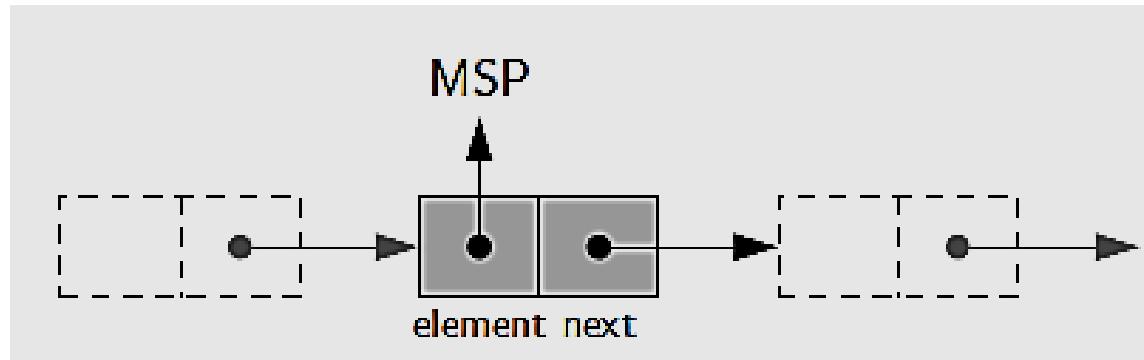
OutLine

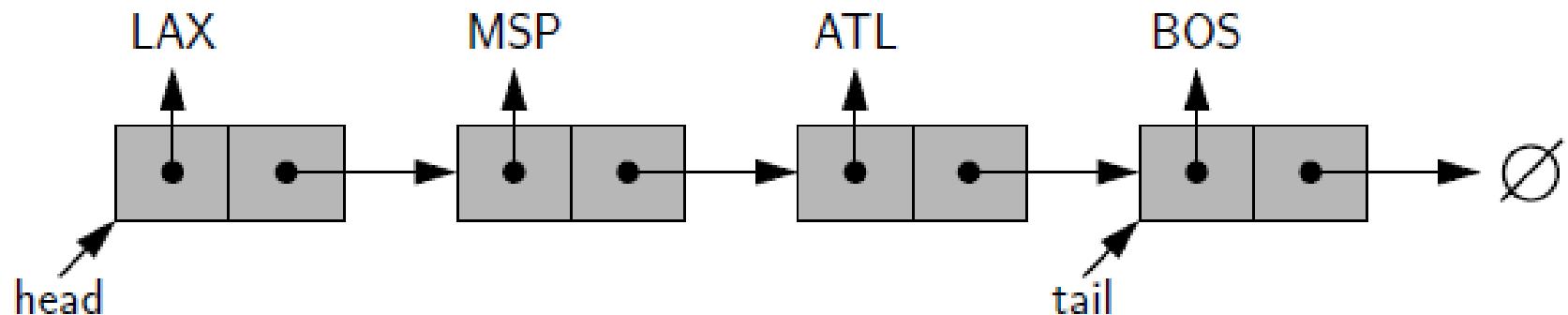




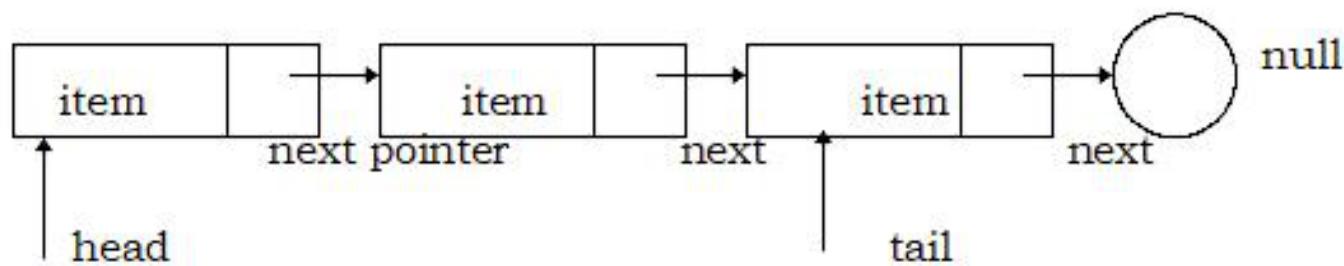
Konsep Untaian

- Struktur untaian adalah suatu struktur data, disebut “node” yang terdiri atas **wadah untuk data** (item/element) dan suatu **penunjuk** (pointer/next) ke elemen untaian berikutnya.
- Secara khusus pada bagian ini akan disajikan suatu struktur untaian yang hanya memiliki satu pointer dan biasa disebut sebagai *untaian tunggal (singly linked list)*.





Suatu pointer khusus di-sediakan untuk menunjuk ke awal untaian atau node pertama yaitu pointer ***head***, dan di akhir untaian suatu pointer lain biasa diadakan untuk menunjuk item kosong atau ***null***. Suatu pointer tambahan ***tail*** bisa ditambahkan untuk menunjuk node terakhir, sebelum null.



class ListNode

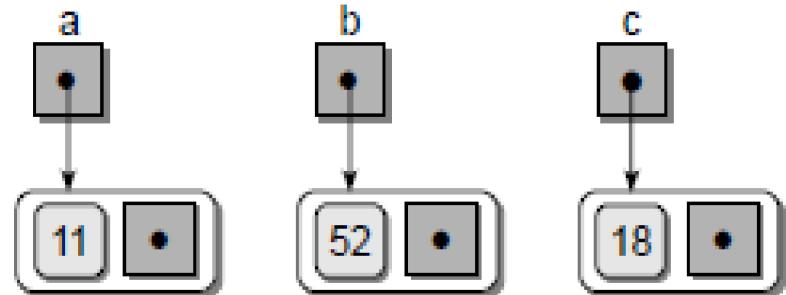
- Dalam Python, struktur dasar untaian tunggal ini dapat ditulis dalam bentuk “class ListNode”, terdiri atas item (wadah) dan next (penunjuk) sebagai berikut (Python tidak mengenal pointer) :

```
class ListNode :  
    def __init__(self, data) :  
        self.item = data  
        self.next = None
```

instruksi: `a = ListNode(11)`

`b = ListNode(52)`

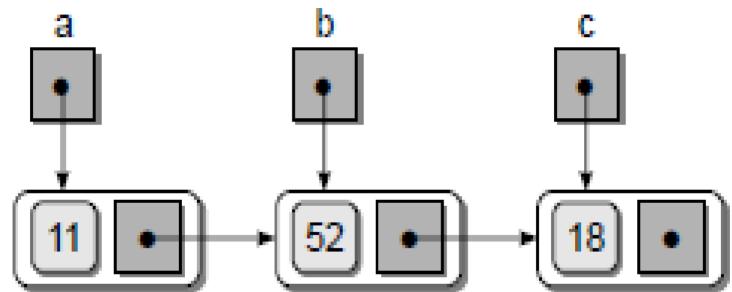
`c = ListNode(18)`



apabila ditambahkan instruksi:

`a.next = b`

`b.next = c`



ADT Singly Linked List

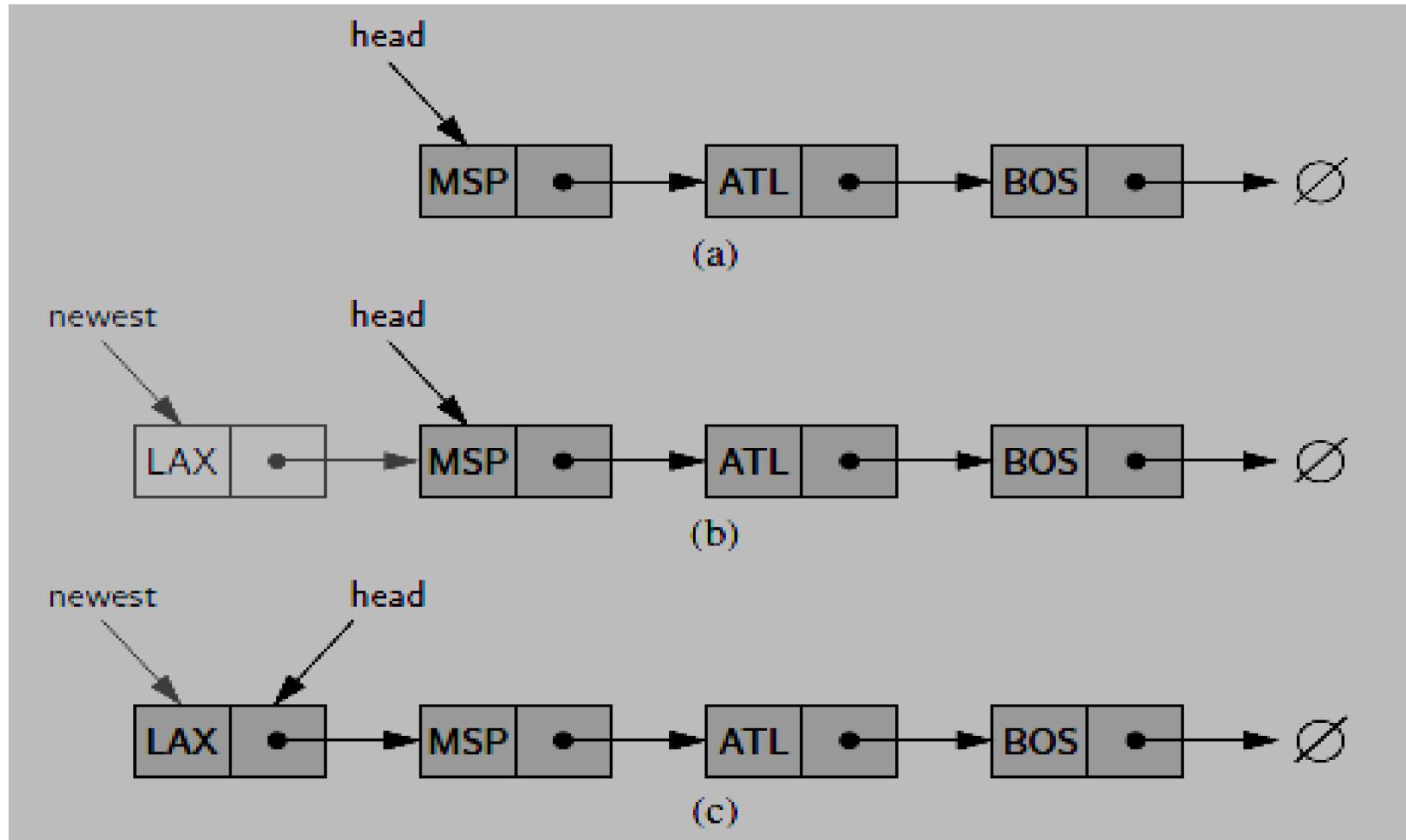
- Beberapa fungsi utama dalam ADT untaian antara lain adalah sebagai berikut:
- **init** : inisialisasi struktur untaian, pointer head menunjuk (\rightarrow) null.
- **add_first(e)** : memasukkan elemen baru e di-awal list
- **add_last(e)** : memasukkan elemen baru e di-akhir list
- **delete_first()** : menghapus elemen pertama dari untaian
- **delete_item(x)** : menghapus elemen tertentu dari untaian
- **search_list(e)** : mencari satu elemen dalam untaian
- **first()** : menampilkan elemen pertama
- **last()** : menampilkan elemen terakhir
- **is_empty()** : memeriksa apakah list kosong
- **insert_after()** : menyisipkan elemen baru setelah suatu node tertentu
- **traverse()** : menelusuri semua node dan menampilkan isi-nya

Fungsi inisialisasi: `__init__`

```
class _LinkedList:  
    #Base class untuk representasi linked list  
    class _Node:  
        #class non-public untuk node untaian  
        __slots__ = '_element' , '_next'      # streamline memory  
  
        def __init__(self, element, next):  
            # initialize node's fields  
            self._element = element          # user's element  
            self._next = next                # pointer ke node berikut  
  
    def __init__(self):  
        #menciptakan untaian kosong  
        self._head = None  
        self._tail = None  
        self._size = 0 # number of elements
```

Penyisipan data / insert

- Setelah untaian di-bentuk maka langkah berikutnya adalah langkah pemasukkan atau penyisipkan data/elemen. Sebenarnya ada beberapa teknik untuk memasukkan elemen ke dalam untaian, antara lain: **sisip depan (add_first)**, **sisip akhir (add_last)**, **sisip setelah node tertentu (insert_after)**.
- Prosedur sisip-depan (**add_first**) berarti menempatkan elemen baru selalu didepan, langkahnya adalah sebagai berikut:
 - Ciptakan node baru, newest(node);
 - buatlah pointer next node baru ini menujuk ke elemen yang sedang ditunjuk Head,
 newest.next <- head
 - isi node ini dengan elemen baru x,
 newest.item <- x;
 - pointer Head menujuk ke node baru ini,
 head <- newest;



illustrasi fungsi `add_first()`

Fungsi : add_first()

- Python coding:

```
add_first(L, e) :
```

```
# ciptakan node untuk e
newest = Node(e)
# next node baru menunjuk ke node yg ditunjuk head
newest.next = L.head
# head menunjuk node baru
L.head = newest
L.size = L.size + 1 # dan size ditambah 1
```

- Prosedur sisip-akhir (**add_last**) berarti menempatkan elemen baru selalu dibelakang, langkahnya bisa sebagai berikut:
 - Ciptakan node baru, newest(node)
 - tempatkan elemen baru pada node baru, **node.isi <- x;**
 - pointer node baru selalu menunjuk akhir (null),
newest.next ← null;
 - Telusuri untaian mulai dari Head hingga elemen terakhir (Last)


```
thisNode <- Head;
Last <- null;
while (thisNode.next <> null) do
    Last <- thisNode;
    thisNode <- thisNode.next;
endwhile;
```
 - pointer terakhir mnunjuk node baru, **Last.next ←node;**

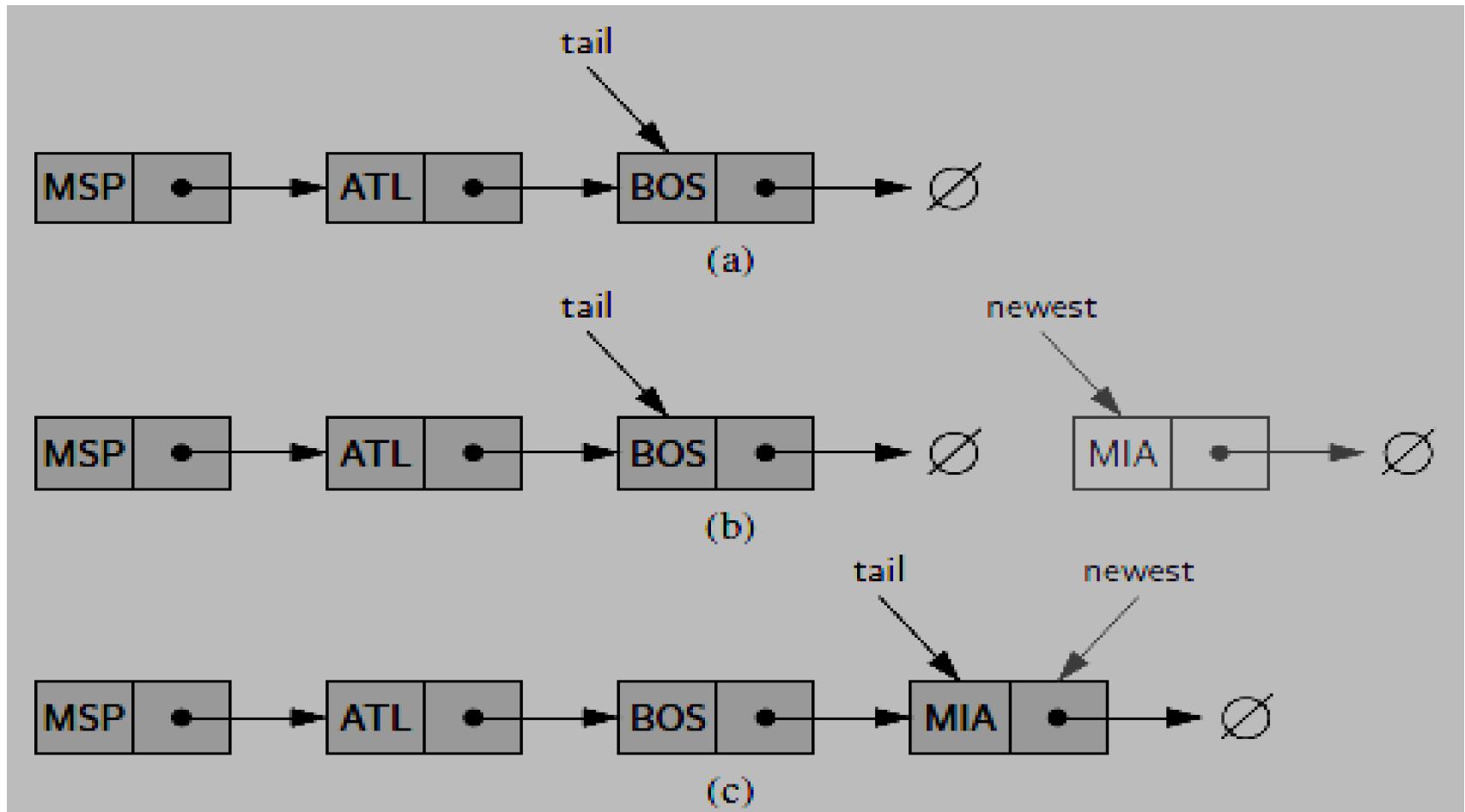
- Cara lain untuk sisip-akhir adalah dengan menciptakan suatu pointer akhir (tail), kemudian memakai pointer ini untuk selalu menuju elemen terakhir.
- Prosedur add_last bisa dilakukan sebagai berikut:
 - Ciptakan satu elemen (node) baru
 - Buatlah pointer next dari elemen baru ini menunjuk null
 - Bila pointer tail menunjuk null (empty), maka buatlah tail menuju elemen baru

Fungsi: add_last()

- **Python coding**

```
add_last(L, e):
```

```
    # ciptakan node baru untuk e
    newest = Node(e)
    # next node baru menunjuk null
    newest.next = Null
    # next dari tail menunjuk node baru
    L.tail.next = newest
    # pointer tail pindah ke node baru
    L.tail = newest
    L.size = L.size + 1 # size ditambah 1
```



Ilustrasi fungsi `add_last()`

- Menyisipkan node baru e setelah node tertentu (**insert_after**), bisa dilakukan dengan langkah sebagai berikut:
 - Cari posisi x dari node d dalam list (lihat search_list)
 - andaikan y adalah pointer dari x.next
 - ciptakan node baru newest(node)
 - buat pointer next node baru sama dengan y,
 $\text{newest.next} \leftarrow y$
 - buat pointer x.next menunjuk node baru, $x.\text{next} \leftarrow \text{newest}$

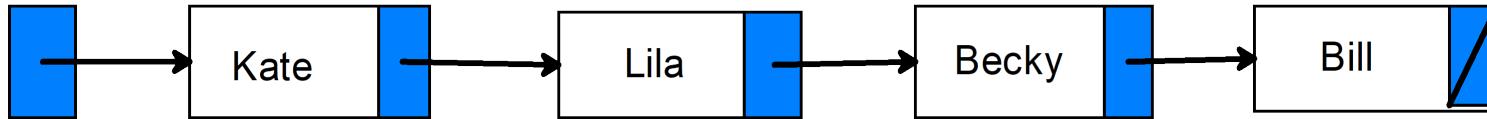
Fungsi: insert_after()

```
def insert_after(self, d, e):
    #masukkan element e setelah node d
    #dan return new node.
    x = self._head
    y = x._element
    found = False
    while (not found and x != None):
        if (y == d):
            found = True
            z = x._next
            newest = self._Node(e,z)
            x._next = newest
            self._size += 1
            break
```

```
else:  
    x = x._next  
    if (x != None):  
        y = x._element  
  
if (found):  
    return newest._element  
else:  
    print('Node : ',d, ' tdk ada dalam list')
```

Pencarian node (search)

- Prosedur pencarian elemen **search_List** dalam suatu untaian dapat dilakukan sebagai berikut:
 - telusuri untaian dengan memeriksa isi elemen apakah sama dengan elemen yang dicari
 - bila sama maka variabel found = true,
 - bila tidak found = false

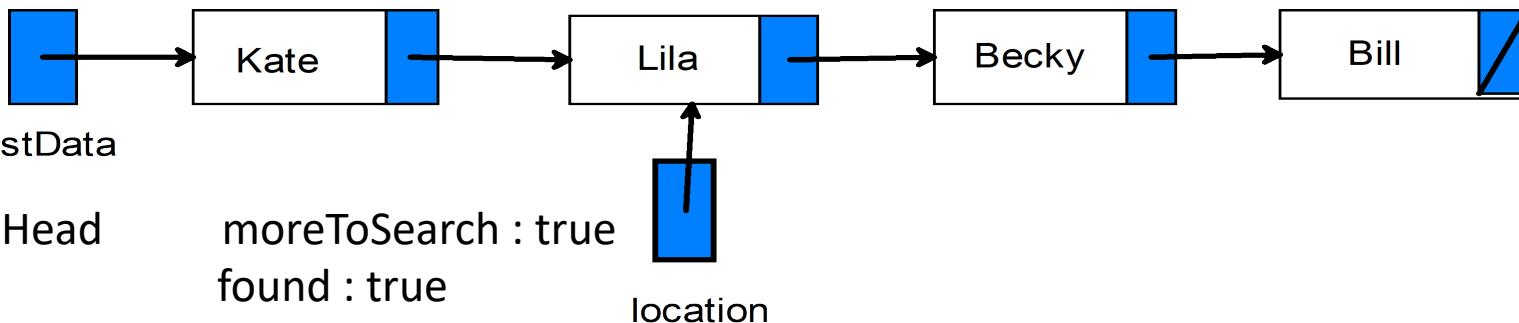


listData

Head

moreToSearch : false
found: false
location: false

Mencari “Charlie” dalam untaian menghasilkan found \leftarrow false;



listData

Head

moreToSearch : true
found : true
location

Mencari “Lila” dalam untaian menghasilkan found \leftarrow true.

Fungsi search_list()

```
def search_list(self,e):
    if self.is_empty():
        raise Empty('List is empty')
    x = self._head
    y= x._element
    i=1
    found = False
    while (not found and x != None):
        print(y)
        if (y == e):
            found = True
            break
```

```
else:  
    x = x._next  
    if (x != None):  
        y = x._element  
        i+=1  
    if (found):  
        print('ditemukan di posisi ke-',i)  
    else:  
        print(e, 'tidak ditemukan dalam list')  
return found
```

menghapus node (delete)

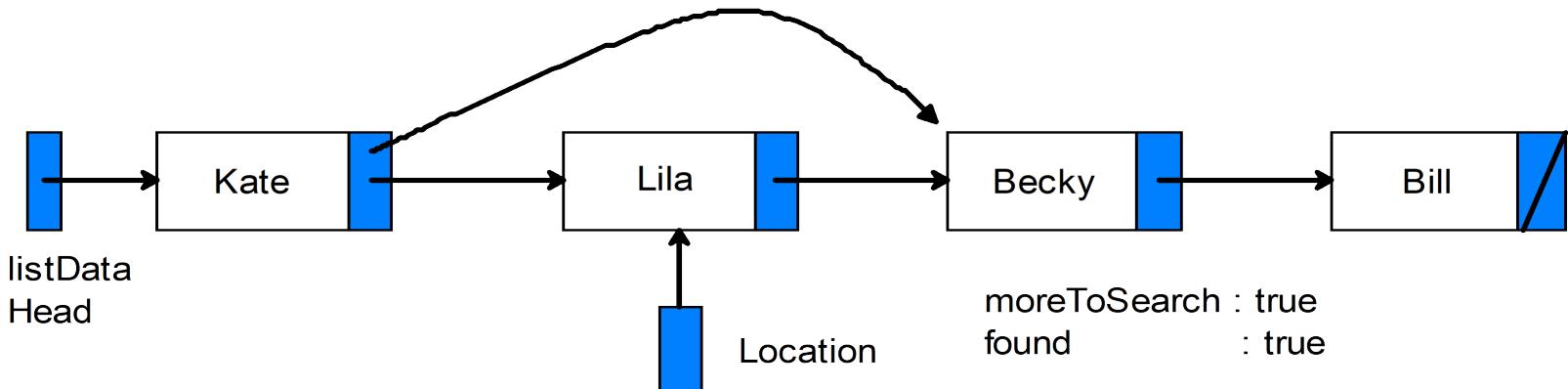
- Proses penghapusan satu elemen dari untaian (delete_Item) dapat dilakukan sebagai berikut:
 - cari elemen yang akan dihapus (location, found=true)
 - bila ketemu maka hapus (dispose)

Rincian Langkah

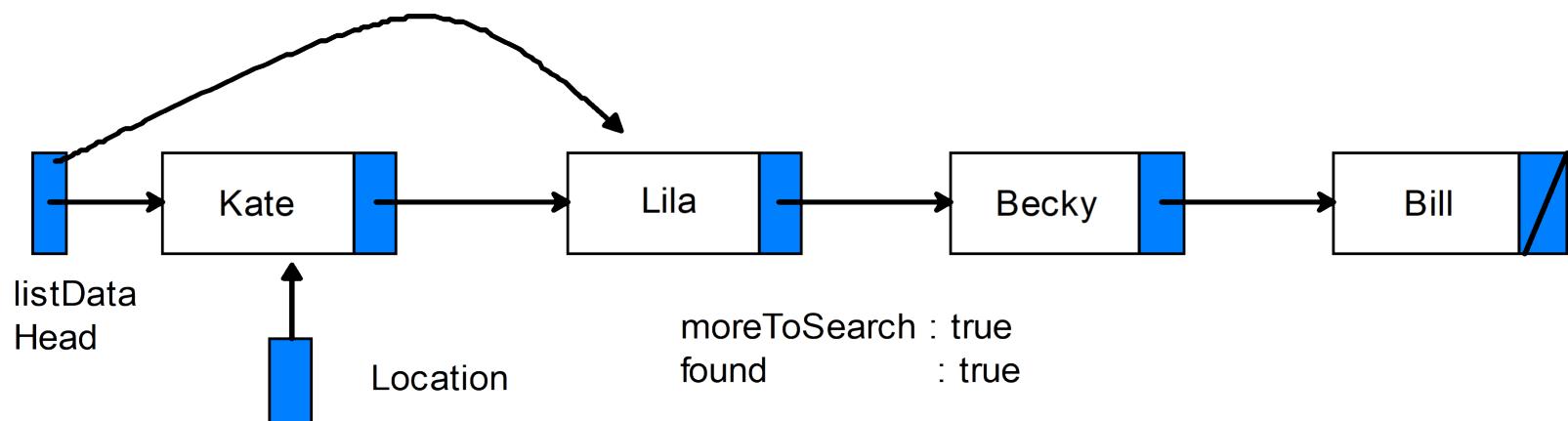
```
searchList(Head, thisptr, lastptr, x, done);
if (done)
then if (lastptr = Null)
      then lastptr <- Head.next;
          dispose(Head);
          Head <- lastptr;
      else lastptr.next <- thisptr.next;
          dispose(thisptr);
      endif
endif.
```

Proses pencarian elemen untuk dihapus

```
bila (x = Head.isi)
{ berarti elemen pertama yang akan dihapus }
maka:    lokasi <- Head;
          Head <- lokasi.next;
          dispose(lokasi);
bila tidak maka :
lokasi <- Head;
{ cari posisi elemen }
      selama ( x <> (lokasi.next).isi):
          lokasi <- lokasi.next;
          hapus <- lokasi.next;
          lokasi.next <- (lokasi.next).next;
          dispose(hapus);
```



Apabila yang dihapus (delete) adalah “Lila”, maka pointer ke Lila dipindahkan ke Becky



Apabila yang dihapus (delete) adalah “Kate”, maka Head pointer menunjuk Lila.

Fungsi delete_item()

```
def delete_item( self, item ):  
    predNode = None  
    curNode = self._head  
    while curNode is not None and  
          curNode._element != item :  
        predNode = curNode  
        curNode = curNode.next  
    # item harus ada agar bisa diambil.  
    assert curNode is not None,  
           "Item harusnya ada dalam list !!!"
```

```
# Lepaskan node dan berikan nilai itemnya.  
self._size -= 1  
if curNode is self._head :  
    self._head = curNode.next  
else :  
    predNode.next = curNode.next  
return curNode.item
```

- Salah satu cara menghapus elemen termudah adalah dengan menghapus elemen paling depan (**delete_first**), prosedurnya sebagai berikut:

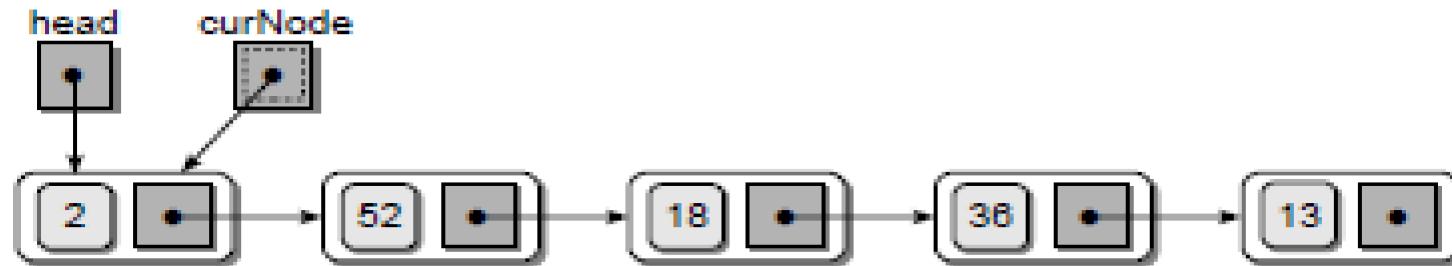
```
def delete_first(self):  
    if self.is_empty():  
        raise Empty('List is empty')  
    answer = self._head._element  
    self._head = self._head._next  
    self._size -= 1  
    return answer
```

Penelusuran (traverse)

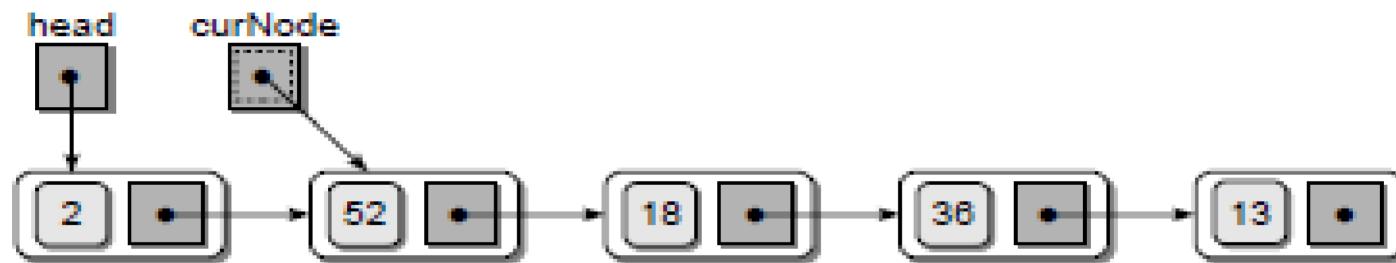
- Menelusuri semua elemen dari linked list dapat dimulai dari pointer head kemudian selanjutnya menelusuri pointer next ke node berikutnya hingga node terakhir dengan isi None ditemukan.

```
def traversal( head ):  
    curNode = head  
    while curNode is not None  
        print(curNode.item)  
        curNode = curNode.next
```

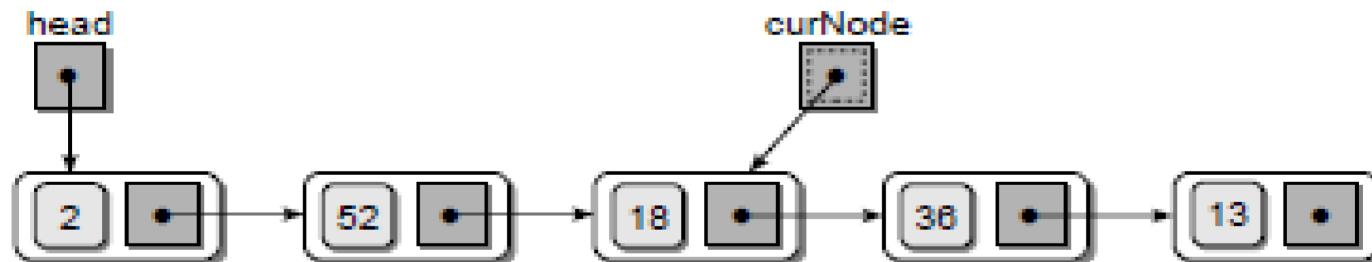
penelusuran mulai dari head



mengikuti pointer next ke node berikutnya



dan ke node selanjutnya



```
1 #LinkedList.py === @SUARGA
- class _LinkedList:
-     #A base class untuk representasi linked list."""
-
-     class _Node:
-         #Lightweight, class nonpublic class untuk node untaian.
-         __slots__ = '_element' , '_next' # streamline memory
-
-         def __init__(self, element, next): # constructor
-             self._element = element # user's element
-             self._next = next       # pointer ke node berikutnya
-
-     def __init__(self):
-         #Create an empty list."""
-         self._head = None
-         self._tail = None
-         self._size = 0 # jumlah awal elemen
-
-     def __len__(self):
```

```
    def __len__(self):
        """memberikan jumlah elemen dalam list."""
        return self._size

    def is_empty(self):
        """Apabila list kosong mengembalikan True"""
        return self._size == 0

27 def add_first(self,e):
    """Add element at the front
    newest = self._Node(e,self._head)
    self._head = newest
    if self._size == 0:
        self._tail = newest
    self._size += 1

    def add_last(self,e):
        """Add element at the back
```

```
    - □
    - □
        def add_last(self,e):
            #Add element at the back
            newest = self._Node(e,None)
            self._tail._next = newest
            self._tail = newest
            self._size +=1
    - □
40
    - □
42
        def delete_first(self):
            #delete first element
            if self.is_empty():
                raise Empty('List is empty')
            answer = self._head._element
            self._head = self._head._next
            self._size -= 1
            return answer
    - □
50
    - □
        def delete_last(self):
            print('Sorry, we can not remove the last node !')
    - □
    - □
        def delete_item( self, item ):
            """Delete item from the list
            If item is not found, do nothing
            """
            if self.is_empty():
                return None
            previous = None
            current = self._head
```

```
def delete_item( self, item ):
    #delete item from the list
    predNode = None
    curNode = self._head
    while curNode is not None and curNode._element != item :
        predNode = curNode
        curNode = curNode._next
60

# item harus ada agar bisa dihapus.
assert curNode is not None, "Item tidak ada dalam list !!!"

# Lepaskan node dan berikan nilai itemnya.
self._size -= 1
if curNode is self._head :
    self._head = curNode._next
else :
    predNode._next = curNode._next
70
return curNode._element

def first(self):
    if self.is_empty():


```

```
def first(self):
    if self.is_empty():
        raise Empty('List is empty')
    return self._head._element

80
def last(self):
    if self.is_empty():
        raise Empty('List is empty')
    return self._tail._element

def search_list(self,e):
    if self.is_empty():
        raise Empty('List is empty')
    x = self._head
    y= x._element
    i=1
    found = False
90
    while (not found and x != None):
        print(y)
        if (y == e):
            found = True
            break
        else:
```

```

        break
    else:
        x = x._next
        if (x != None):
            y = x._element
            i+=1
100
    if (found):
        print('ditemukan di posisi ke-',i)
    else:
        print(e, ' tidak ditemukan dalam list')

# traversing the list
def traverse( self ):
    curnode = self._head
    while curnode is not None:
        print(curnode._element)
        curnode = curnode._next

def insert_after(self, d, e):
    #Add element e after node d and return new node.""""
    x = self._head
    y = x._element
    found = False
    while (not found and x != None):

```

```
        found = False
        while (not found and x != None):
            if (y == d):
                found = True
                z = x._next
                newest = self._Node(e,z)
                x._next = newest
                self._size += 1
                break
            else:
                x = x._next
                if (x != None):
                    y = x._element
130
        if (found):
            return newest._element
        else:
            print('Node : ',d, ' tdk ada dalam list')
```

```
· #LListUsage.py == pemakaian Linked_List @Suarga
· from LinkedList import _LinkedList
· def main():
·     L = _LinkedList()
·     L.add_first("Ahmad")
·     L.add_first("Becky")
·     L.add_first("Charlie")
·     L.add_last("David")
·     print("Isi Linked List: ")
10    L.traverse()
·     print("insert Erlang setelah Becky")
·     L.insert_after("Becky", "Erlang")
·     print("Cari Erlang")
·     L.search_list("Erlang")
·     print("hapus Becky")
·     L.delete_item("Becky")
·     print("Isi Linked List setelah delete elemen: ")
·     L.traverse()
·     print("cari Becky")
20    L.search_list("Becky")
·     print("Coba hapus elemen terakhir")
·     L.delete_last()
·     print("Isi Linked List")
·     L.traverse()
·     print("apakah LList kosong? : ", L.is_empty())
·
·     main()
```

Python Interpreter

```
*** Remote Interpreter Reinitialized ***
Isi Linked List:
Charlie
Becky
Ahmad
David
insert Erlang setelah Becky
Cari Erlang
Charlie
Becky
Erlang
ditemukan di posisi ke- 3
hapus Becky
Isi Linked List setelah delete elemen:
Charlie
Erlang
Ahmad
David
```

```
cari Becky
Charlie
Erlang
Ahmad
David
Becky tidak ditemukan dalam list
Coba hapus elemen terakhir
Sorry, we can not remove the last node !
Isi Linked List
Charlie
Erlang
Ahmad
David
apakah LList kosong? : False
>>> |
```

Stack memakai Untaian

- Struktur tumpukan (Stack) dapat di-implementasi menggunakan LinkedList sebagai basisnya.
- Berikut ini implementasi dari “LinkedStack.py”

```
1 #LinkedStack.py == @Suarga
2 class LinkedStack:
3     """LIFO Stack implementation using a singly linked list for storage."""
4
5     #----- nested Node class -----
6     class _Node:
7         """Lightweight, nonpublic class for storing a singly linked node."""
8         __slots__ = '_element', '_next'      # streamline memory usage
9
10    def __init__(self, element, next): # initialize node's fields
11        self._element = element          # reference to user's element
12        self._next = next                # reference to next node
13
14    #----- stack methods -----
15    def __init__(self):
16        """Create an empty stack."""
17        self._head = None                # reference to the head node
18        self._size = 0                   # number of stack elements
19
20    def __len__(self):
21        """Return the number of elements in the stack."""
22        return self._size
23
24    def is_empty(self):
```

```
def is_empty(self):
    """Return True if the stack is empty."""
    return self._size == 0

30 def push(self, e):
    """Add element e to the top of the stack."""
    self._head = self._Node(e, self._head) # create and link a new node
    self._size += 1

def top(self):
    """Return (but do not remove) the element at the top of the stack.

    #Raise Empty exception if the stack is empty.

    if self.is_empty():
        raise Empty('Stack is empty')
    return self._head._element

40 def pop(self):
```

```
def pop(self):
    #Remove and return the element from the top of the stack (i.e., LIFO).
    #Raise Empty exception if the stack is empty.

    if self.is_empty():
        raise Empty('Stack is empty')
    answer = self._head._element
    self._head = self._head._next          # bypass the former top node
    self._size -= 1
    return answer
```

Contoh Penggunaan

```
>>> s=LinkedStack()
>>> s.push(5)
>>> s.push(3)
>>> print(len(s))
2
>>> print(s.top())
3
>>> s.pop()
3
>>> s.is_empty()
False
>>> s.pop()
5
>>> print(len(s))
0
>>> s.is_empty()
True
>>>
```

Implementasi Queue memakai Untaian

- LinkedList dapat dapat diaplikasikan untuk membuat antrian (queue) sebagai pada tumpukan
- Berikut ini listing: LinkedQueue.py

```
#LinkedQueue.py == @Suarga
class LinkedQueue:
    #FIFO queue implementation using
    #a singly linked list for storage."""

    class _Node:
        #Lightweight, nonpublic class for storing
        #a singly linked node."""
        __slots__ = '_element', '_next'      # streamline memory usage
10
        def __init__(self, element, next): # initialize node's fields
            self._element = element          # reference to user's element
            self._next = next                # reference to next node

    # ----- method -----
20
    def __init__(self):
        #Create an empty queue."""
        self._head = None
        self._tail = None
        self._size = 0 # number of queue elements

    def __len__(self):
        #Return the number of elements in the queue."""
        return self._size
25
```

```
def is_empty(self):
    """Return True if the queue is empty."""
    return self._size == 0
30

def first(self):
    """Return, but do not remove the element at the front.
    If self.is_empty():
        raise Exception('Queue is empty')
    return self._head._element # front aligned with head of list
35

def dequeue(self):
    """Remove and return the first element of the queue (i.e., FIFO).
    Raise Empty exception if the queue is empty.
40
    if self.is_empty():
        raise Exception('Queue is empty')
    answer = self._head._element
    self._head = self._head._next
    self._size -= 1
    if self.is_empty(): # special case as queue is empty
        self._tail = None # removed head had been the tail
    return answer
45

def enqueue(self, e):
50
```

```
50 def enqueue(self, e):
    """
    Add an element to the back of queue."""
    newest = self._Node(e, None) # node will be new tail node
    if self.is_empty():
        self._head = newest # special case: previously empty
    else:
        self._tail._next = newest
    self._tail = newest # update reference to tail node
    self._size += 1

60 def traverse( self ):
    curnode = self._head
    while curnode is not None:
        print(curnode._element)
        curnode = curnode._next
```

Contoh Penggunaan

```
>>> q=LinkedQueue()
>>> q.enqueue(10)
>>> q.enqueue(9)
>>> q.enqueue(8)
>>> print(len(q))
3
>>> print(q.first())
10
>>> q.is_empty()
False
>>> q.dequeue()
10
>>> print(len(q))
2
>>> q.dequeue()
9
>>> q.dequeue()
8
>>> q.is_empty()
True
>>>
```