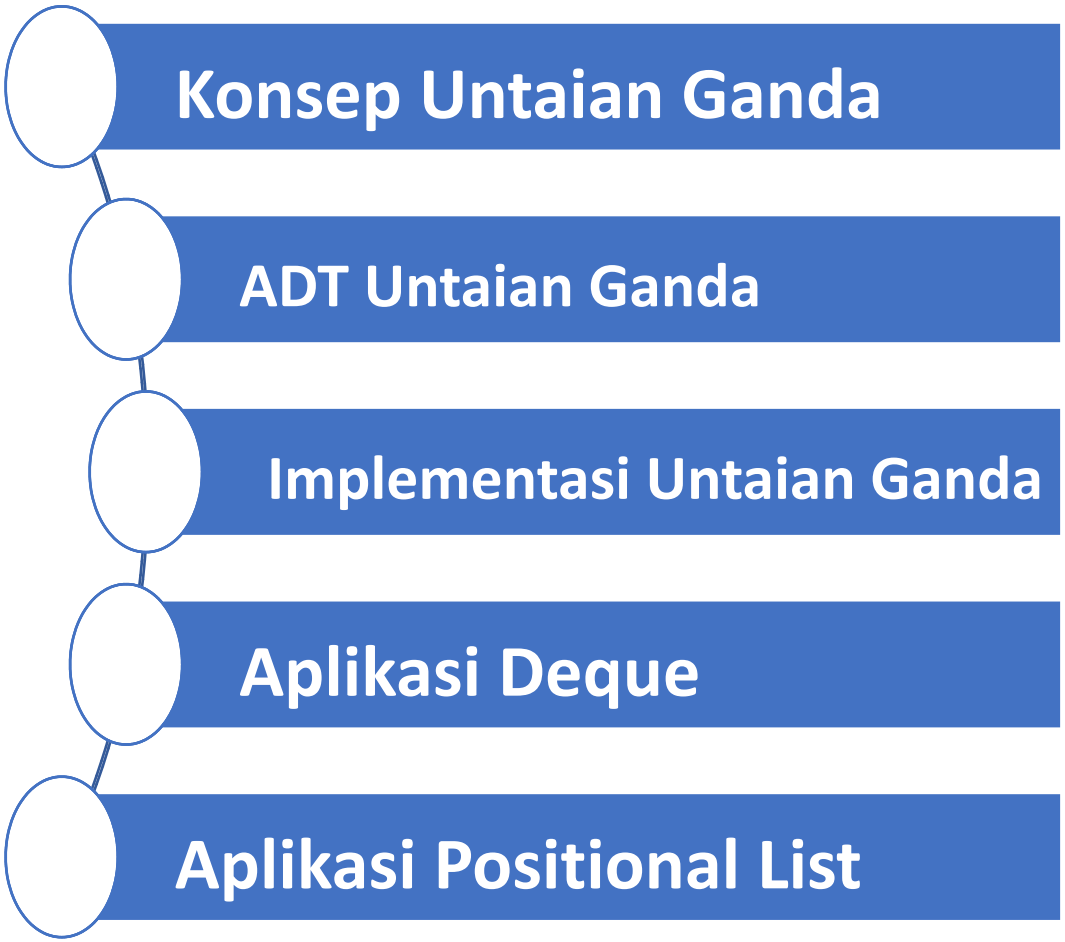
#KampusKuningGading

STRUKTUR DATA (PYTHON)

"Untaian Ganda (Doubly Linked List)"

[@SUARGA| [Pertemuan 11]

OutLine

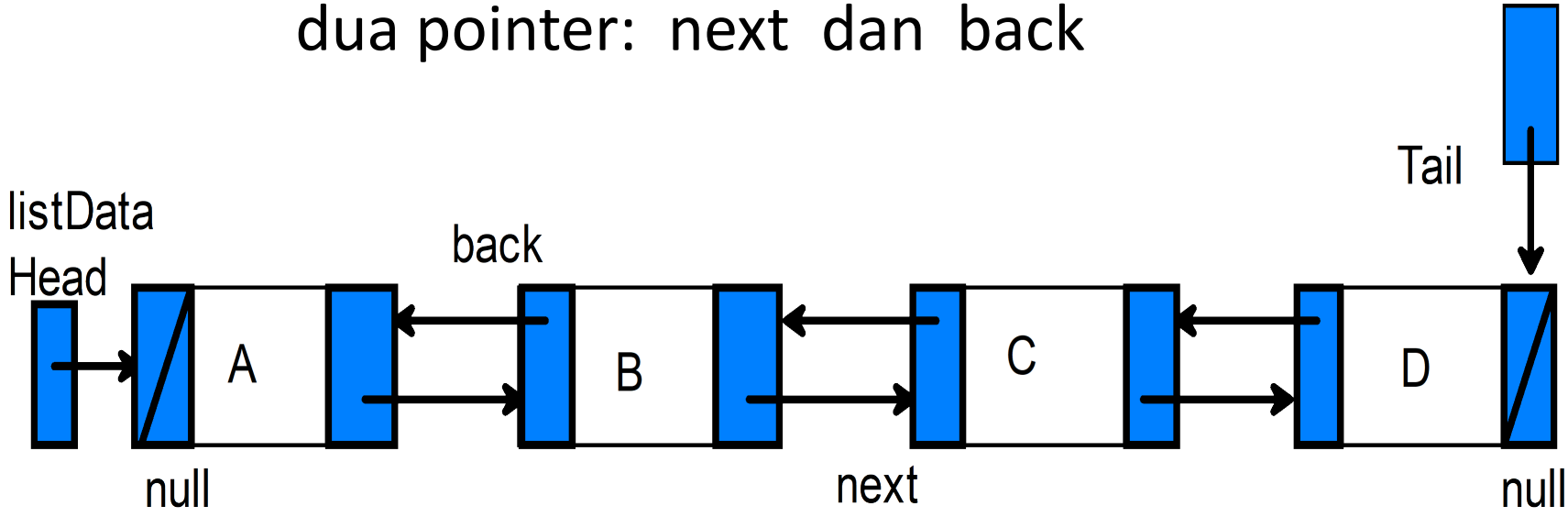

Konsep Untaian Ganda
ADT Untaian Ganda
Implementasi Untaian Ganda
Aplikasi Deque
Aplikasi Positional List

Konsep Untaian Ganda

- Struktur untaian ganda (doubly linked list) adalah suatu untaian yang memiliki dua pointer, satu pointer menunjuk ke belakang yaitu *next* dan satu pointer menunjuk ke depan yaitu *back*.
- Struktur ini memungkinkan akses pada untaian bersifat maju (forward) atau bersifat mundur (backward) dari posisi mana saja didalam untaian. *Head* adalah pointer yang menunjuk pada elemen pertama, dan *Tail* adalah pointer yang menujuk pada elemen terakhir.

Gambaran Untaian Ganda (Doubly Linked List) ==> Dlist

dua pointer: next dan back



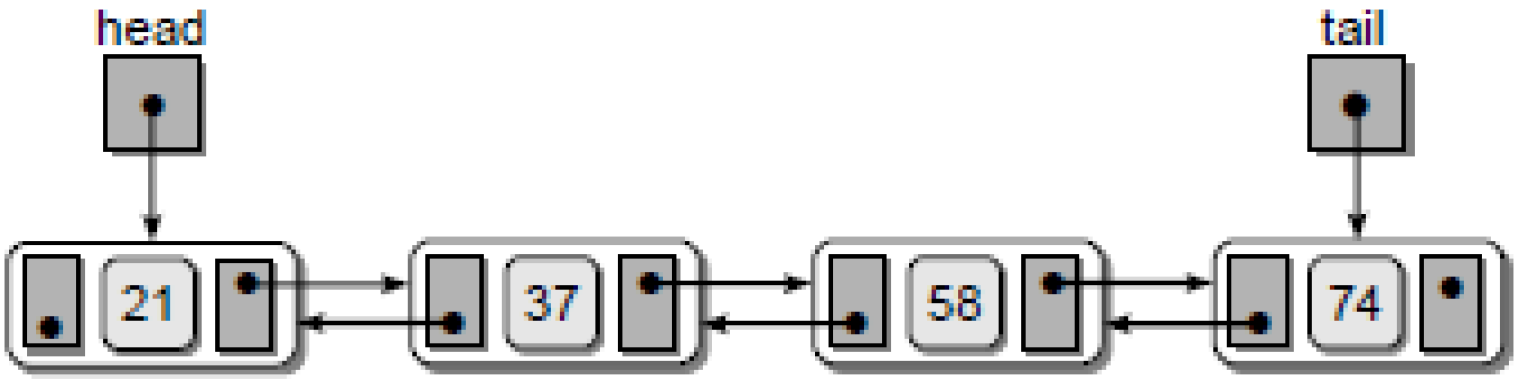
Model node Dlist

model record data:

```
pointer : ^Dlist;
Type Dlist : record
<
      isi   :  item;
      next : pointer;
      back : pointer;
>
```

model Python class

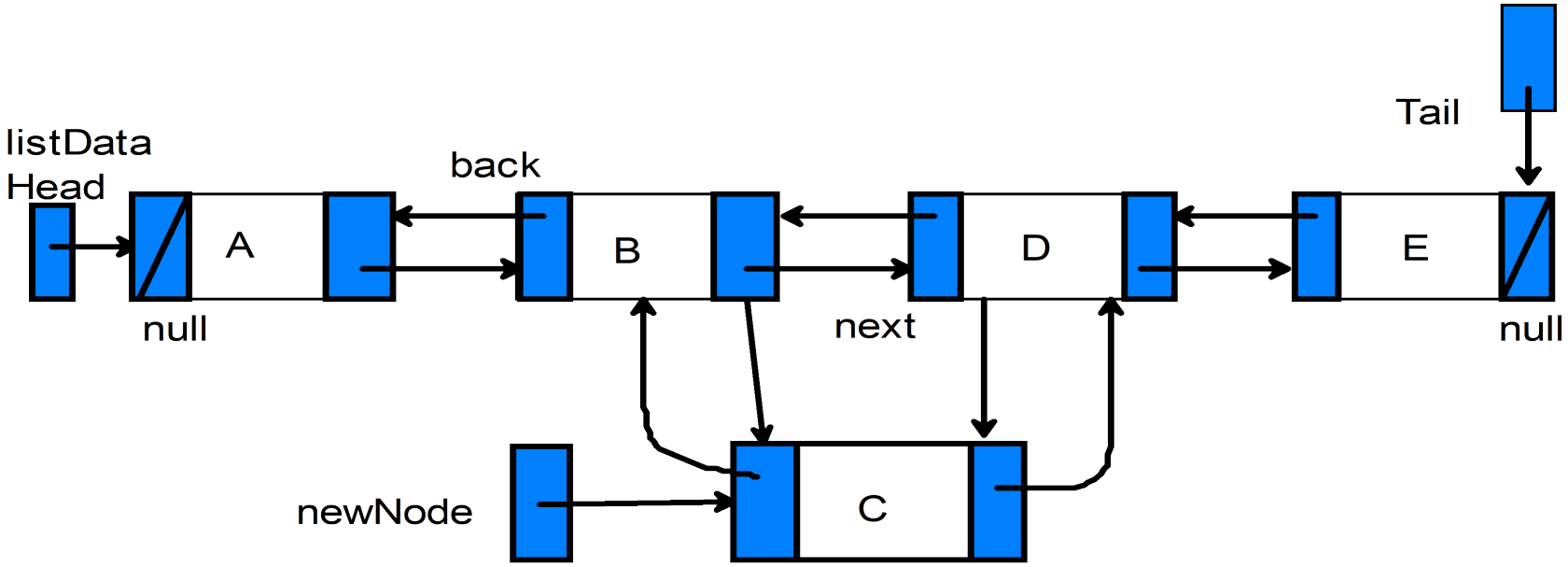
```
class Node :
   def __init__(self, data):
       self.isi = data
       self.next = None
       self.back = None
```

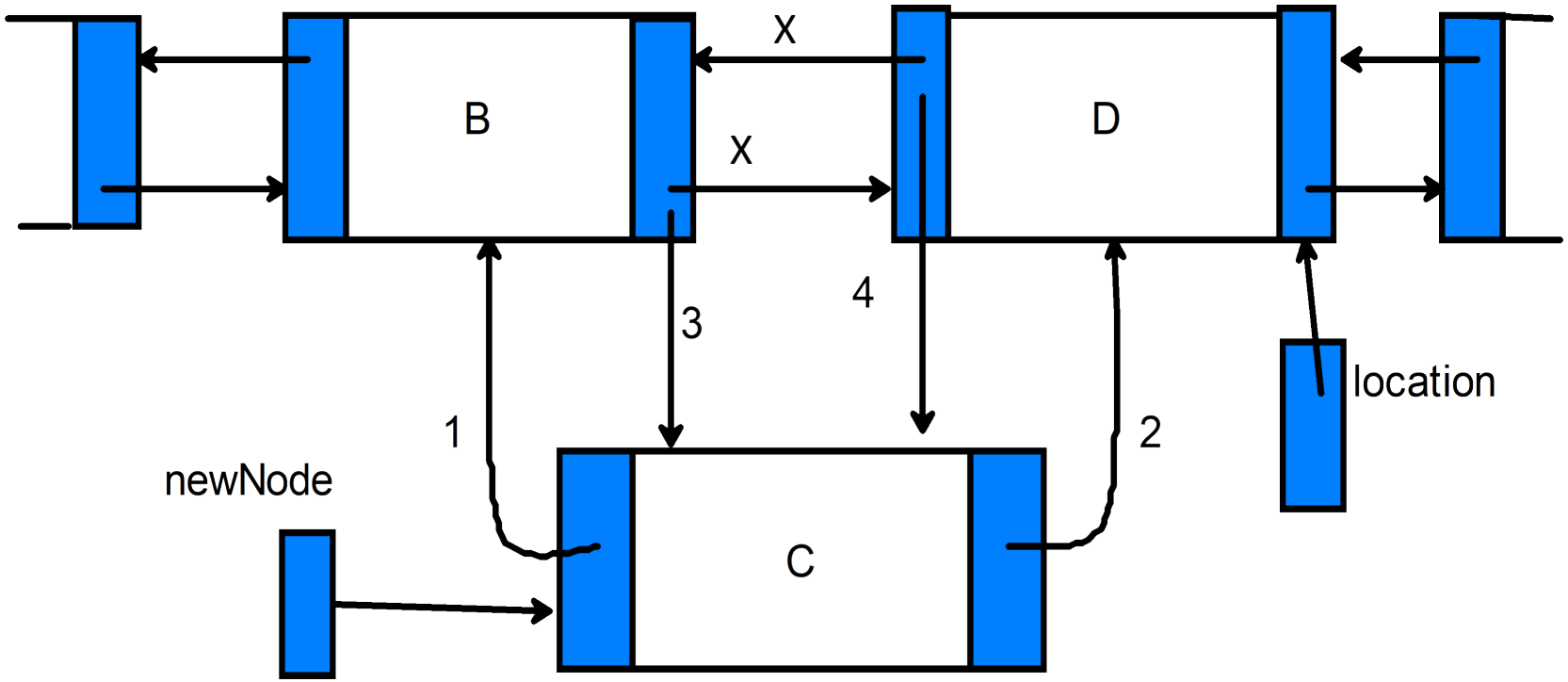


Model Dlist dengan 4 elemen

Proses penyisipan: insert

Proses menyisipkan suatu elemen baru pada antara Head dan Tail bisa dilihat pada gambar berikut ini. Terlihat pada gambar bahwa ada 4 pointer yang harus diatur.





detil penyisipan elemen baru ke dalam untaian ganda

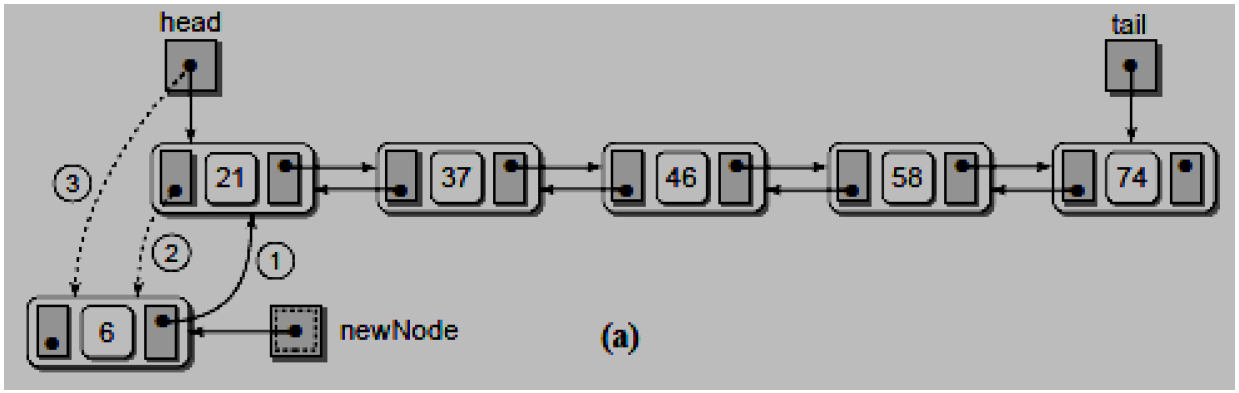
- Proses penyisipan ini secara umum adalah sebagai berikut:

1. Ciptakan node baru, New_node=Node(x), node.next <- null; node.back <- null;
2. suatu pointer location untuk mencari posisi,
   location <- Head;
3. Selama x masih <= isi pointer, maju terus,
   while (x <= (location.next).isi) do
   location <- location.next;
4. pointer back dari node baru (1) di-update,
   node.back <- location.back;
5. pointer back dari thisPoint (4) menunjuk node,
   location.back <- node;
6. pointer next posisi sebelumnya (3) menunjuk node,
   (location.back).next <- node;
7. pointer next dari node baru (2) menunjuk posisi saat ini,
   node.next<-location;

Kasus: node baru di depan

- Namun bisa saja terjadi bahwa node baru memiliki isi paling kecil sehingga harus ditempatkan di-depan, sebelum head:

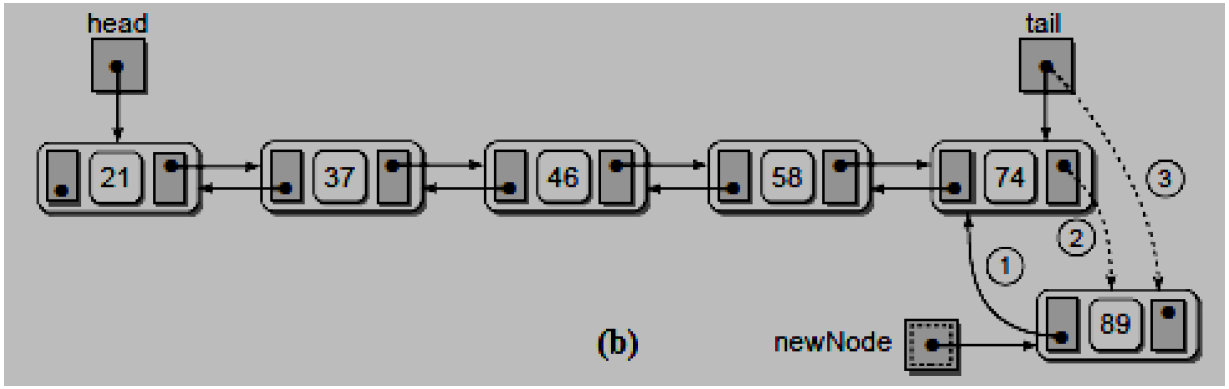
```
if value < head.isi:
        newnode.next = head    #1
        head.back = newnode    #2
        head = newnode         #3
```



Kasus: node baru di belakang

- Atau bisa saja isi node-baru paling besar sehingga harus ditempatkan pada posisi paling belakang setelah tail.

```
if value > tail.isi:
        newnode.back = tail      #1
        tail.next = newnode      #2
        tail = newnode           #3
```



Coding insert node

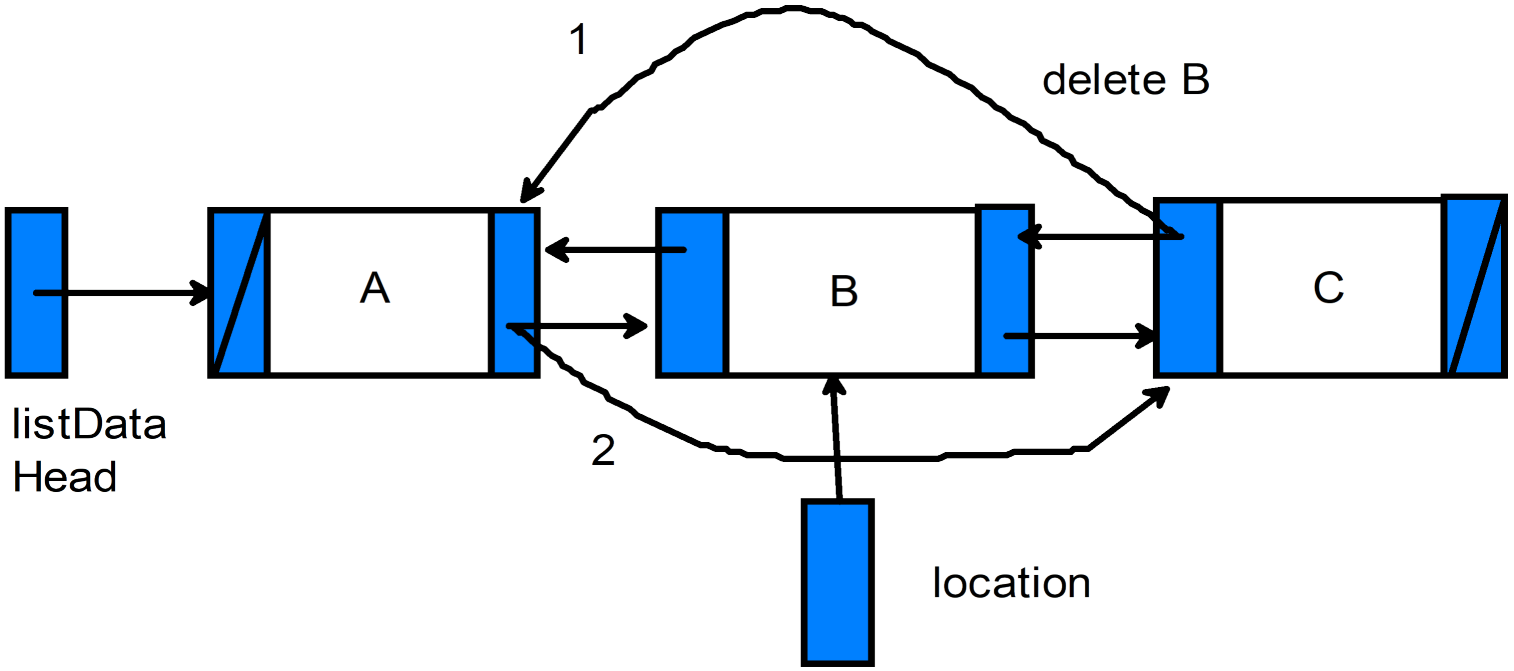
```
newnode = DListNode( value )
if head is None :          # empty list
  head = newnode
  tail = head
elif value < head.data :   # insert before head
  newnode.next = head
  head.prev = newnode
  head = newnode
elif value > tail.data :   # insert after tail
  newnode.prev = tail
  tail.next = newnode
  tail = newnode
else :                     # insert in the middle
  node = head
  while node is not None and node.data < value :
    node = node.next

  newnode.next = node
  newnode.prev = node.prev
  node.prev.next = newnode
  node.prev = newnode
```

proses delete node

Proses menghapus sebuah elemen dari untaian ganda adalah sebagai berikut.

Suatu elemen yang terdapat didalam sebuah untaian ganda berurut dapat dihapus dengan cara mencari posisi (location) dari elemen tersebut (misalnya B), kemudian setelah ditemukan maka lakukan pemindahan pointer, (1) (location.next).back = location.back (2) (location.back).next = location.next, kemudian hapus elemen pada posisi tersebur (dipose(location)). Langkah mengapus satu elemen dari sebuah untaian ganda dapat dijelaskan berikut ini.



node B mau di-buang, cari location dari B, kemudian lakukan pemindahan pointer:
(location.next).back = location.back (1)
(location.back).next = location.next (2)

ADT my_DList.py

- __init__() : memulai DList
- add_first(): tambah data di depan
- add_last():tambah data di belakang
- add_after(): tambah data setelah node
- add_before():tambah data setelah node
- remove(): hapus node
- print_list(): tampil isi DList

```python
#my_DList.py
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.back = None

class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None

    def add_first(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.tail = new_node
        else:
            new_node.next = self.head
            self.head.back = new_node
            self.head = new_node
```

```
def add_last(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        self.tail = new_node
    else:
        self.tail.next = new_node
        new_node.back = self.tail
        self.tail = new_node

def add_after(self, node, data):
    new_node = Node(data)
    current = self.head
    while current:
        if current.data == node:
            new_node.next = current.next
            new_node.back = current
            current.next = new_node
        current = current.next
    if new_node.next is not None:
        new_node.next.back = new_node
```

```
def add_before(self, node, data):
    new_node = Node(data)
    current = self.head
    while current:
        if current.data == node:
            new_node.back = current.back
            new_node.next = current
            current.back = new_node
        current = current.next
    if new_node.back is not None:
        new_node.back.next = new_node

def remove(self, node):
    current = self.head
    if self.head.data == node:
        self.head = node.next
    if self.tail.data == node:
        self.tail = node.back
    else:
        while current:
            if current.data == node:
                current.next.back = current.back
                current.back.next = current.next
            current = current.next
```

```
def print_list(self):
    current = self.head
    while current is not None:
        print(current.data, end=" <=> ")
        current = current.next
    print("None")
```

```python
#my_DList_test.py

from my_DList import *

def main():
    DL = DoublyLinkedList()
    print("Add first: 1 2 3")
    DL.add_first(1)
    DL.add_first(2)
    DL.add_first(3)
    print("Add last: 4 5")
    DL.add_last(4)
    DL.add_last(5)
    print("Isi DList:")
    DL.print_list()
    print("Add 6 before 1")
    DL.add_before(1,6)
    print("Add 7 after 4")
    DL.add_after(4,7)
    print("Isi Dlist")
    DL.print_list()
    print("delete 4")
    DL.remove(4)
    print("Isi DList:")
    DL.print_list()

main()
```

```
========== RESTART: D:/USER/Python/my_DLi
Add first: 1 2 3
Add last: 4 5
Isi DList:
3 <=> 2 <=> 1 <=> 4 <=> 5 <=> None
Add 6 before 1
Add 7 after 4
Isi Dlist
3 <=> 2 <=> 6 <=> 1 <=> 4 <=> 7 <=> 5 <=> None
delete 4
Isi DList:
3 <=> 2 <=> 6 <=> 1 <=> 7 <=> 5 <=> None
```

DList sebagai basis Deque

- Sebuah implementasi DList yang digunakan sebagai basis dari struktur “double ended queue” (Deque) diberikan pada slide selanjutnya.

ADT dari base DList

- __init__() : memulai DList
- __len__() : banyaknya data dalam DList
- is_empty(): apakah DList kosong
- _insert_between() : menyisipkan node baru
- _delete_node() : menghapus node

base_Class

- diperlukan sebuah class untuk node Dlist, atau merupakan basis dari Untaian Ganda (base class)

```
class _DoublyLinkedBase:
    #A base class for a doubly linked list representation.”””
    class _Node:
        #Lightweight, nonpublic class for a doubly linked node.
        __slots__ = '_element' , '_back' , '_next' # streamline memory

        def __init__(self, element, back, next): # initialize node’s fields
            self._element = element # user’s element
            self._back = back # previous node reference
            self._next = next
```

inisialisasi Dlist

```
    def __init__(self):
        #Create an empty list.”””
        self._header = self._Node(None,None,None)
        self._trailer = self._Node(None,None,None)
        self._header._next = self._trailer # trailer is after header
        self._trailer._back = self._header # header is before trailer
        self._size = 0 # number of elements
```

fungsi __init__() menciptakan sebuah untaian ganda yang kosong.

- terdapat dua pointer utama: _header, menunjuk head (kepala) Dlist, kemudian _trailer, menunjuk tail (ekor) Dlist
- selanjutnya disediakan pointer _next dari _header, dan pointer _back dari trailer
- cacah _size adalah nol

fungsi cacah elemen: __len__()

- Fungsi ini mengembalikan cacah elemen DList yaitu _size.

```
def __len__(self):
    #Return the number of elements in the
    #list.”””
    return self._size
```

fungsi memeriksa Dlist: kosong?

- fungsi ini memeriksa apakah cacah _size == 0, bila ya (True) maka berarti Dlist kosong, bila tidak (false) berarti Dlist tidak kosong.

```
def is_empty(self):
        #Return True if list is empty.”””
        return self._size == 0
```

```
    def _insert_between(self, e, predecessor, successor):
        #Add element e between two nodes and return new node.""""
30      newest = self._Node(e, predecessor, successor)
        # linked to neighbors
        predecessor._next = newest
        successor._back = newest
        self._size += 1
        return newest
```

elemen e akan disisipkan antara predecessor dan successor
newest adalah node dari data e, oleh sebab itu pointer
predecessor._next menunjuk **newest** dan
successor._back menunjuk **newest**
akibatnya newest ada diantara predecessor dan successor.

```
1. location <- Head;
2. while (x <> location.isi) do
          location <- location.next;
3. (location. back).next <- location.next;
4. (location.next).back <- location.back;
5. dispose(location);
```

```python
    def _delete_node(self, node):
        #Delete nonsentinel node, and return its element.”””
        predecessor = node._back
        successor = node._next
        predecessor._next = successor
        successor._back = predecessor
        self._size -= 1
        element = node._element # record deleted element
        node._back = node._next = node._element = None
        # deprecate node
        return element     # return deleted element
```

Listing lengkap

```
#_DoublyLinkedBase.py == @Suarga,
#implementasi class dari Untaian Ganda
class _DoublyLinkedBase:
    #A base class for a doubly linked list representation."""
    class _Node:
        #Lightweight, nonpublic class for a doubly linked node.
        __slots__ = '_element' , '_back' , '_next' # streamline memory

        def __init__(self, element, back, next): # initialize node's fields
            self._element = element # user's element
            self._back = back # previous node reference
            self._next = next

    def __init__(self):
        #Create an empty list."""
        self._header = self._Node(None,None,None)
        self._trailer = self._Node(None,None,None)
        self._header._next = self._trailer # trailer is after header
        self._trailer._back = self._header # header is before trailer
        self._size = 0 # number of elements
```

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

def is_empty(self):
    #Return True if list is empty."""
    return self._size == 0

def _insert_between(self, e, predecessor, successor):
    #Add element e between two nodes and return new node."""
    newest = self._Node(e, predecessor, successor)
    # linked to neighbors
    predecessor._next = newest
    successor._back = newest
    self._size += 1
    return newest

def _delete_node(self, node):
    #Delete nonsentinel node, and return its element."""
    predecessor = node._back
    successor = node._next
    predecessor._next = successor
    successor._back = predecessor
    self._size -= 1
    element = node._element # record deleted element
    node._back = node._next = node._element = None
    # deprecate node
    return element     # return deleted element
```

Aplikasi:Deque (Double-ended queue)

- Deque atau antrian dengan dua ujung adalah antrian dimana proses insert (enqueue) maupun delete (dequeue) bisa dilakukan baik di-depan maupun di-belakang antrian. Dengan demikian pada antrian ini diperlukan prosedur ADT sebagai berikut:

- **add_first(e)** : menambah elemen e dari depan
- **add_last(e)** : menambah elemen e dari belakang
- **delete_first()** : membuang elemen dari depan
- **delete_last()** : membuang elemen dari belakang
- **first()** : melihat elemen pertama
- **last()** : melihat elemen terakhir
- **is_empty()** : memeriksa apakah antrian kosong
- **len()** : menghitung isi antrian

Implementasi Deque memakai _DoublyLinkedBase

```
#LinkedDeque.py == Aplikasi Deque  $Suarga
from _DoublyLinkedBase import _DoublyLinkedBase
#LinekdDeque mewarisi (inheritance) _DoublyLinkedBase
class LinkedDeque(_DoublyLinkedBase): # note the use of inheritance
#Double-ended queue implementation based on a doubly linked list.”””

    def first(self):
        #Return, do not remove the element at the front
        if self.is_empty():
            raise Empty('Deque is empty')
        return self._header._next._element # item just after header

    def last(self):
        #Return,do not remove the element at the back
        if self.is_empty():
            raise Empty('Deque is empty')
        return self._trailer._back._element #item just before trailer
```

```
    def insert_first(self, e):
        #Add an element to the front of the deque.”””
        self._insert_between(e, self._header, self._header._next)
        # after header

    def insert_last(self, e):
        #Add an element to the back of the deque.”””
        self._insert_between(e, self._trailer._back, self._trailer)
        # before trailer

    def delete_first(self):
        #Remove and return the element from the front
        #Raise Empty exception if the deque is empty.
        if self.is_empty():
            raise Empty('Deque is empty')
        return self._delete_node(self._header._next)
        # use inherited method
```

```
def delete_last(self):
    #Remove and return the element from the back.
    #Raise Empty exception if the deque is empty.
    if self.is_empty():
        raise Empty('Deque is empty')
    return self._delete_node(self._trailer._back)
    # use inherited method
```

```python
#LinkedDeque_test.py == @Suarga
#mencoba aplikasi LinkedDequeu
from LinkedDeque import *

def main():
    print("Menciptakan Deque")
    Q = LinkedDeque()
    print("Memasukkan 5 data:")
    print("3 data didepan: 2,4,6")
    Q.insert_first(2)
    Q.insert_first(4)
    Q.insert_first(6)
    print("2 data dibelakang: 5,7")
    Q.insert_last(5)
    Q.insert_last(7)
    print("Panjang antrian:")
    print(len(Q))
    print("Data pertama:")
    print(Q.first())
    print("Data terakhir:")
    print(Q.last())
    print("Melayani yang didepan:")
    print(Q.delete_first())
    print("Melayani yang dibelakang:")
    print(Q.delete_last())
    print("Data terakhir antrian = ")
    print(Q.last())
    print("Apakah Antrian sudah kosong?")
    print(Q.is_empty())

main()
```

```
>>>
========== RESTART: D:/USER/Python/LinkedDe
Menciptakan Deque
Memasukkan 5 data:
3 data didepan: 2,4,6
2 data dibelakang: 5,7
Panjang antrian:
5
Data pertama:
6
Data terakhir:
7
Melayani yang didepan:
6
Melayani yang dibelakang:
7
Data terakhir antrian =
5
Apakah Antrian sudah kosong?
False
>>>
```

Aplikasi: Positional List

- Positional List adalah untaian yang dapat meng-handel perubahan elemen untaian secara dinamik, misalnya dalam satu antrian tiba-tiba ada orang yang keluar dari antrian, maka tentu urutan antrian akan berubah, atau ada seseorang dalam antrian yang merelakan teman-nya masuk antrian di-depan posisi-nya.
- Pada sistem berbasis index seperti larik, maka index elemen bisa diubah untuk menangani perubahan dinamik dalam larik.

ADT Positional List

first() : memberikan posisi dari elemen pertama, None bila kosong
last() : memberikan posisi dari elemen terakhir, none bila kosong
before(p): memberikan posisi elemen sebelum posisi p, none bila p posisi pertama dalam list
after(p) : memberikan posisi elemen setelah posisi p, None bila p adalah posisi terakhir dalam list
is_empty() : true bila list kosong
len(L) : memberikan jumlah elemen dalam L
iter(L) : mengembalikan iterator elemen dalam list L
add_first(e) : menyisipkan elemen baru e dibagian depan L
add_last(e) : menyisipkan elemen baru e dibagian belakang L
add_before(p, e) : menyisipkan elemen baru e sebelum posisi p
add_after(p, e) : menyisipkan elemen baru e setelah posisi p
replace(p, e) : mengganti elemen pada posisi p dengan e
delete(p) : menghapus elemen pada posisi p

Menelusuri Positional List

- Variabel posisi p sangat penting dalam penelusuran isi untaian, misalnya potongan kode berikut ini menampilkan semua elemen dari untaian.

```
kursor = data.first()        # posisi pertama
while kursor is not None:    # selama ada
      print(kursor.element()) # tampilkan
      kursor = data.after(kursor)  # pindah
     #ke posisi selanjutnya
```

Implementasi Positional List

```
#PositionalList.py  == implementasi @Suarga
from _DoublyLinkedBase import _DoublyLinkedBase
#inheritance dari _doublyLinkedBase
class PositionalList(_DoublyLinkedBase):
    #A sequential container of elements allowing positional access.

#--------------------------- nested Position class ----------------------
    class Position:
    #An abstraction representing the location of a single element."""

        def __init__(self, container, node):
        #Constructor should not be invoked by user."""
            self._container = container
            self._node = node

        def element(self):
        #Return the element stored at this Position."""
            return self._node._element

        def __eq__ (self, other):
        #Return True if other is a Position representing the same location."
            return type(other) is type(self) and other._node is self._node

        def __ne__(self, other):
        #Return True if other does not represent the same location."""
            return not (self == other) # opposite of eq
```

```
#------------------------------ utility method ------------------------------
    def _validate(self, p):
        #Return position s node, or raise appropriate error if invalid."""
        if not isinstance(p, self.Position):
            raise TypeError( 'p must be proper Position type' )
        if p._container is not self:
            raise ValueError( 'p does not belong to this container' )
        if p._node._next is None: # convention for deprecated nodes
            raise ValueError( 'p is no longer valid' )
        return p._node
#------------------------------ utility method ------------------------------
    def _make_position(self, node):
        #Return Position instance for given node (or None if sentinel)."""
        if node is self._header or node is self._trailer:
            return None # boundary violation
        else:
            return self.Position(self, node) # legitimate position

#------------------------------ accessors ------------------------------
    def first(self):
        #Return the first Position in the list (or None if list is empty)."""
        return self._make_position(self._header._next)

    def last(self):
        #Return the last Position in the list (or None if list is empty)."""
        return self._make_position(self._trailer._back)

    def before(self, p):
        #Return the Position just before Position p (or None if p is first)."
        node = self._validate(p)
        return self._make_position(node._back)
```

```
    def after(self, p):
        #Return the Position just after Position p (or None if p is last).""""
        node = self._validate(p)
        return self._make_position(node._next)

    def __iter__ (self):
        #Generate a forward iteration of the elements of the list.""""
        cursor = self.first()
        while cursor is not None:
            yield cursor.element( )
            print(cursor.element())
            cursor = self.after(cursor)

#------------------------------ mutators ------------------------------
# override inherited version to return Position, rather than Node
    def _insert_between(self, e, predecessor, successor):
        #Add element between existing nodes and return new Position.""""
        node = super()._insert_between(e, predecessor, successor)
        return self._make_position(node)

    def add_first(self, e):
        #Insert element e at the front of the list and return new Position.""""
        return self._insert_between(e, self._header, self._header._next)

    def add_last(self, e):
        #Insert element e at the back of the list and return new Position.""""
        return self._insert_between(e, self._trailer._back, self._trailer)

    def add_before(self, p, e):
        #Insert element e into list before Position p and return new Position.'
        original = self._validate(p)
        return self._insert_between(e, original._back, original)
```

```
def add_after(self, p, e):
    #Insert element e into list after Position p and return new Position.""
    original = self._validate(p)
    return self._insert_between(e, original, original._next)

def delete(self, p):
    #Remove and return the element at Position p."""
    original = self._validate(p)
    return self._delete_node(original) # inherited method returns element

def replace(self, p, e):
    #Replace the element at Position p with e.
    #Return the element formerly at Position p.

    original = self._validate(p)
    old_value = original._element # temporarily store old element
    original._element = e # replace with new element
    return old_value # return the old element value
```

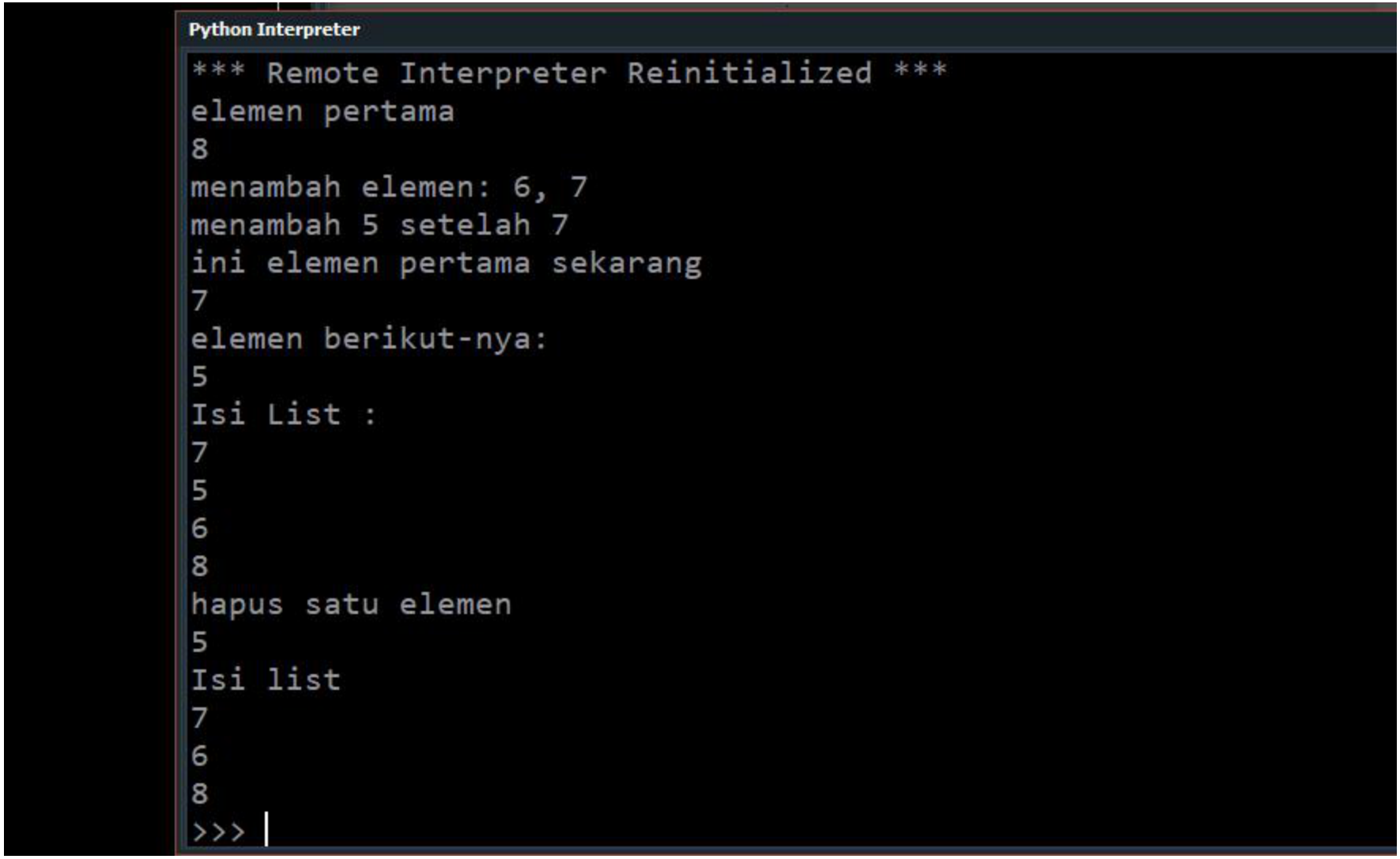
test positional.list

```python
#testPL.py
#test positional list
from PositionalList import PositionalList
def traverse():
    kursor = L.first()
    while kursor is not None:  # selama ada
        print(kursor.element()) # tampilkan
        kursor = L.after(kursor)  # pindah ke posisi selanjutnya

L=PositionalList()
p=L.add_last(8)
k=L.first()
print("elemen pertama")
print(k.element())
```

```
print("menambah elemen: 6, 7")
p=L.add_first(6)
p=L.add_first(7)
print("menambah 5 setelah 7")
p=L.add_after(p,5)
print("ini elemen pertama sekarang")
k=L.first()
print(k.element())
print("elemen berikut-nya:")
k=L.after(k)
print(k.element())
print("Isi List : ")
traverse()
print("hapus satu elemen")
print(L.delete(k))
print("Isi list")
traverse()
```

hasil test


Python Interpreter
*** Remote Interpreter Reinitialized ***
elemen pertama
8
menambah elemen: 6, 7
menambah 5 setelah 7
ini elemen pertama sekarang
7
elemen berikut-nya:
5
Isi List :
7
5
6
8
hapus satu elemen
5
Isi list
7
6
8
>>>