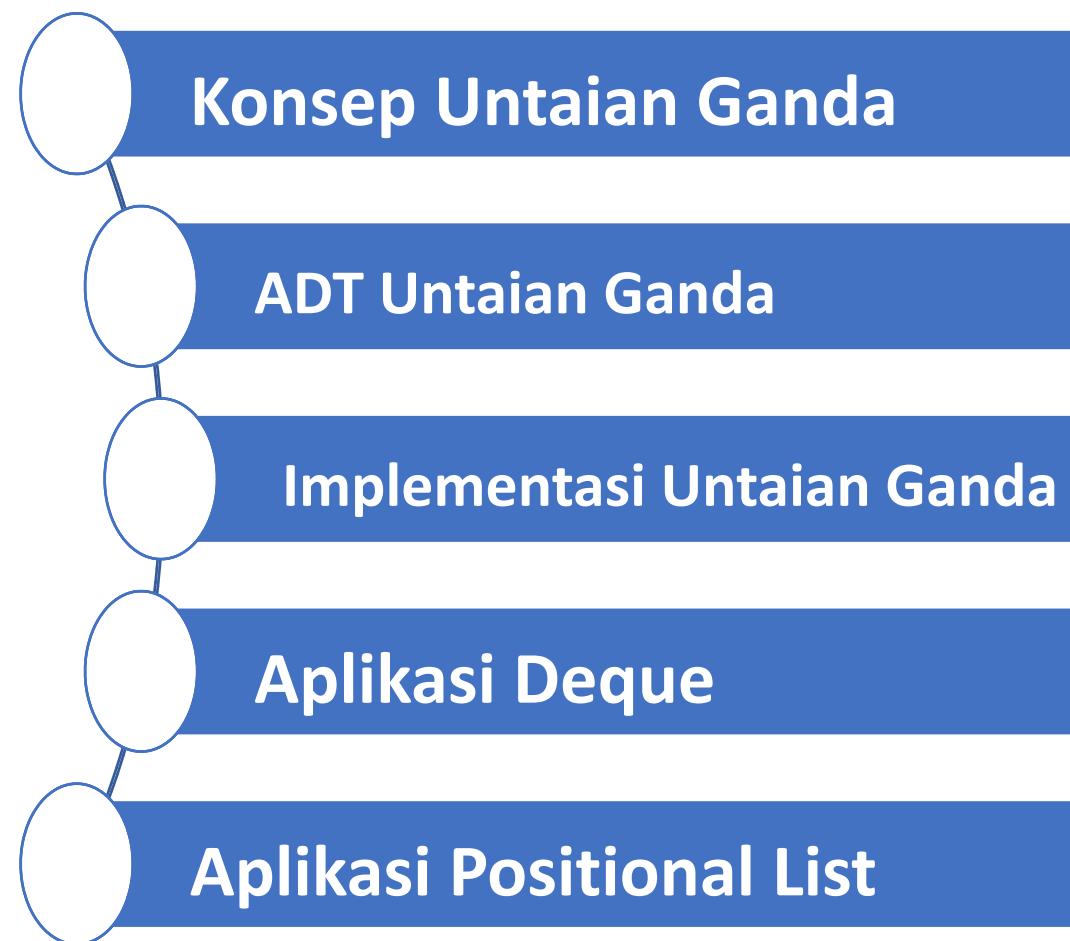


STRUKTUR DATA (PYTHON)

“Untaian Ganda (Doubly Linked List)”

[@SUARGA] [Pertemuan 11]

OutLine

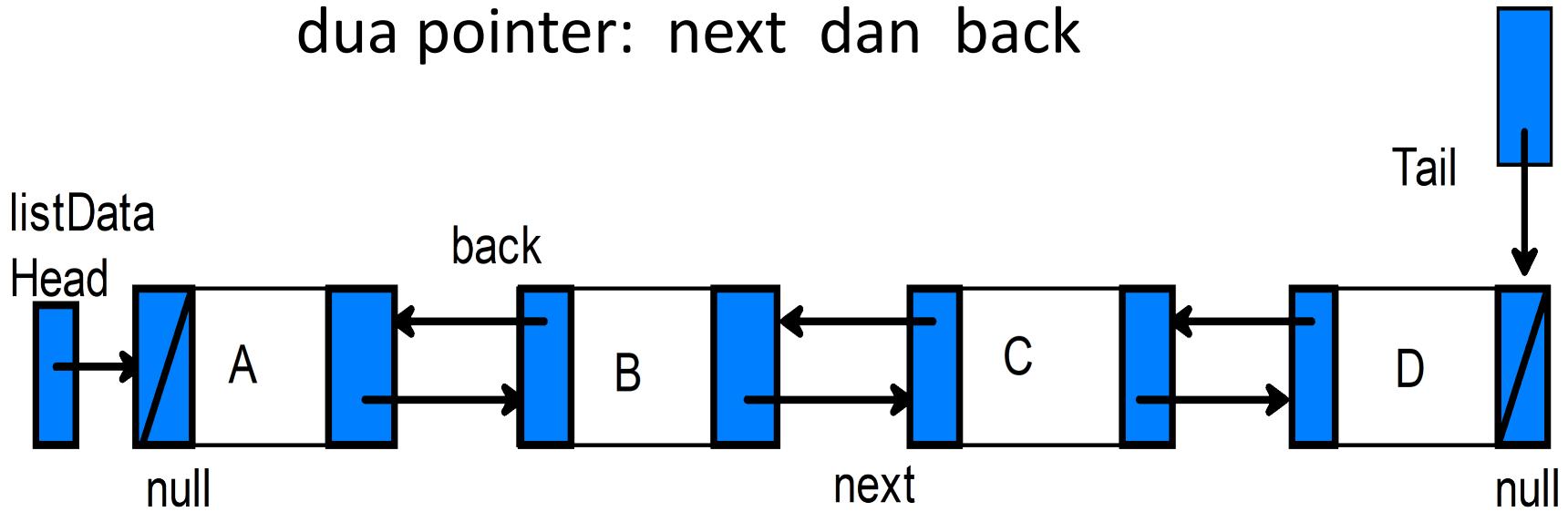




Konsep Untaian Ganda

- 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



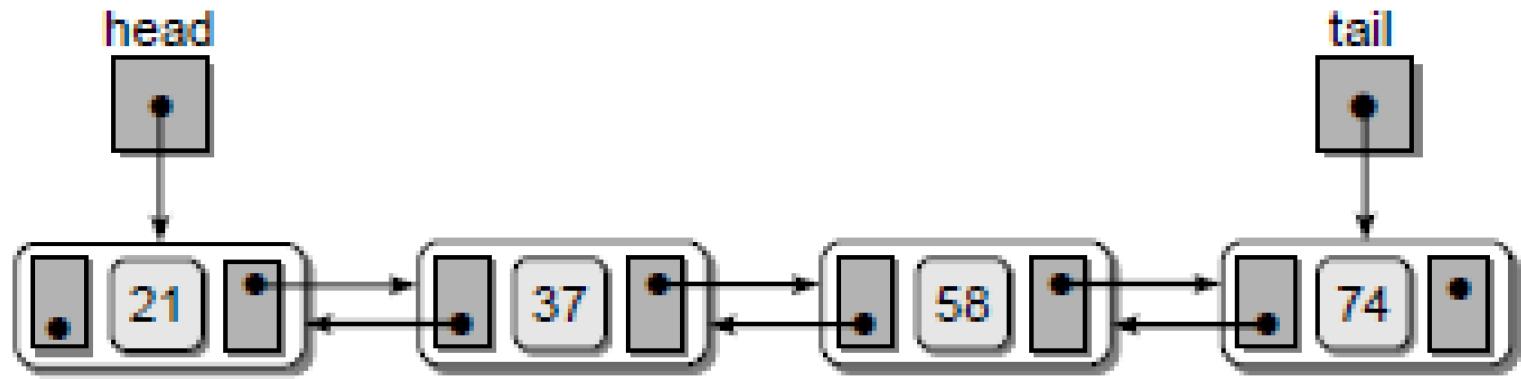
Model node Dlist

model record data:

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

model Python class

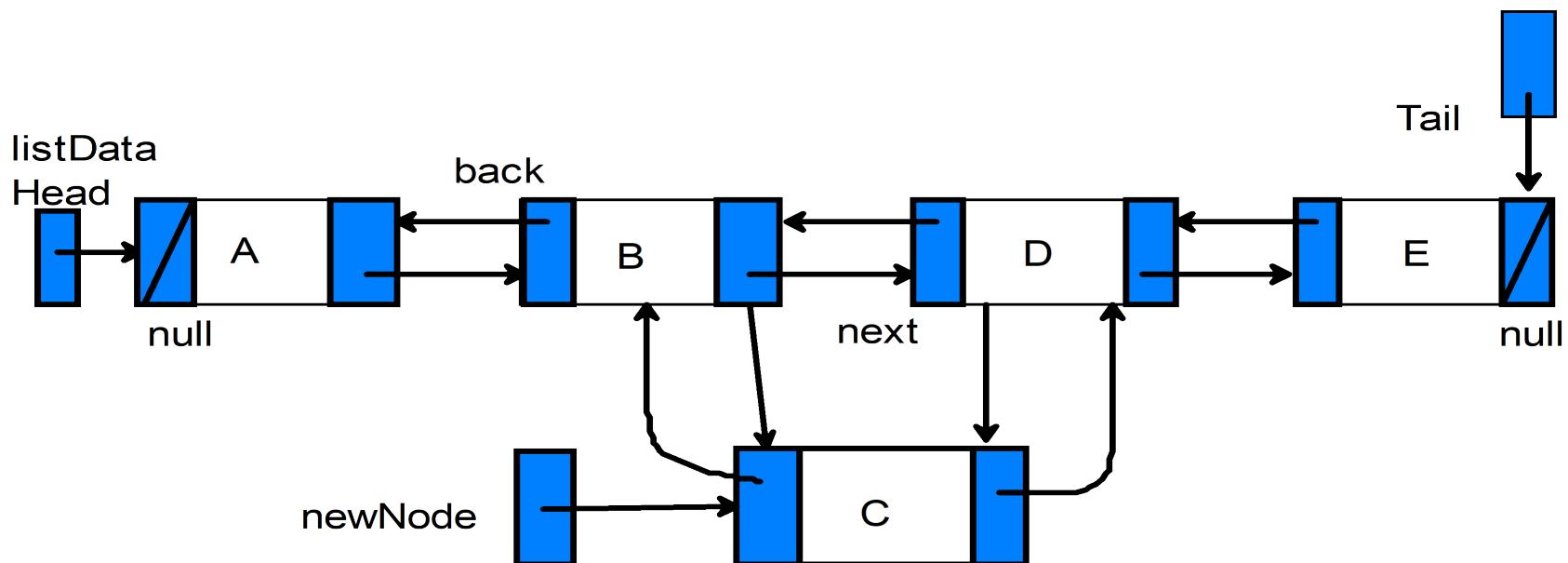
```
class Node :  
    def __init__(self, data):  
        self.isi = data  
        self.next = None  
        self.back = None
```

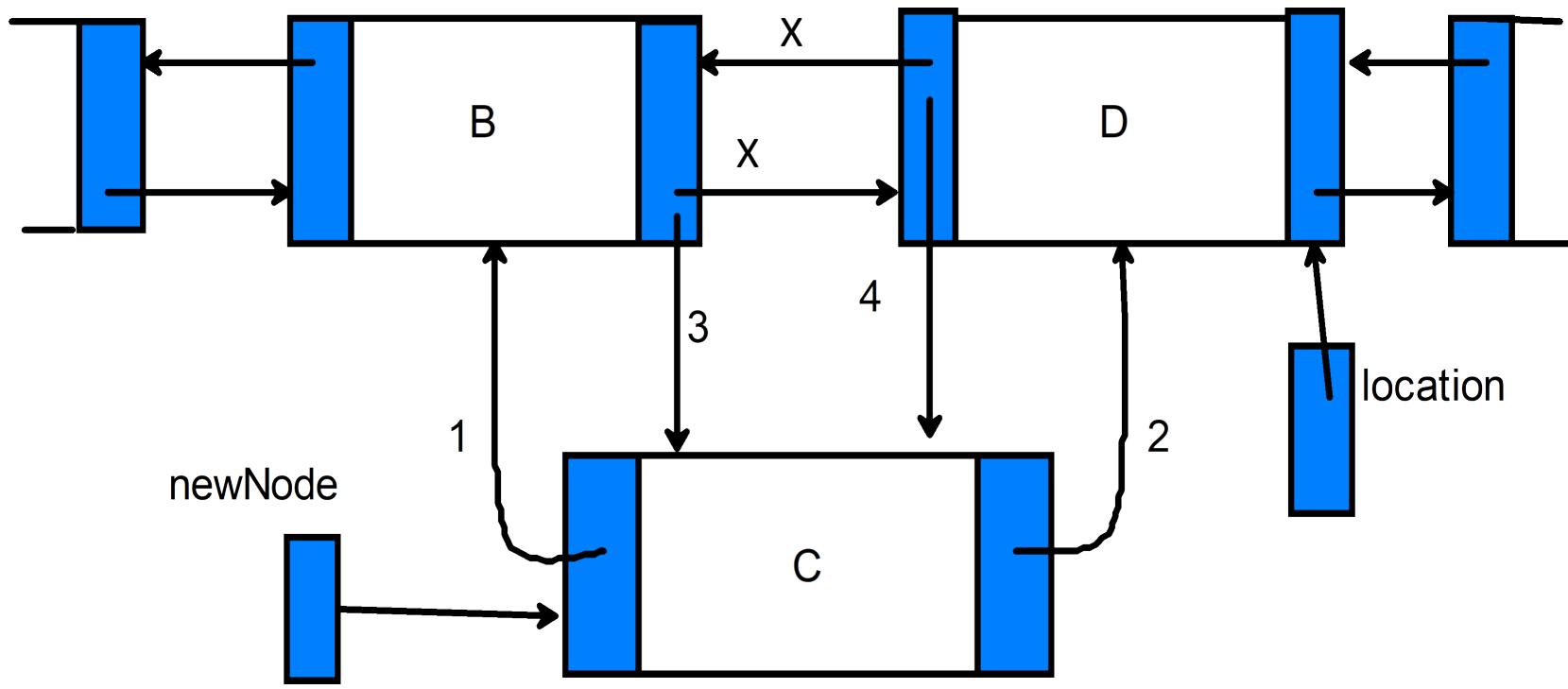


Model Dlist dengan 4 elemen

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.





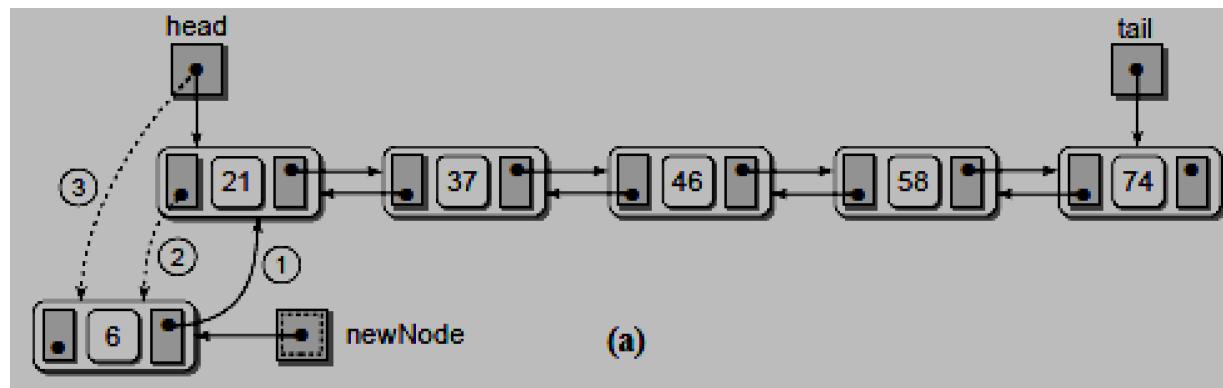
detil penyisipan elemen baru ke dalam untaian ganda

- 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 \leq 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:

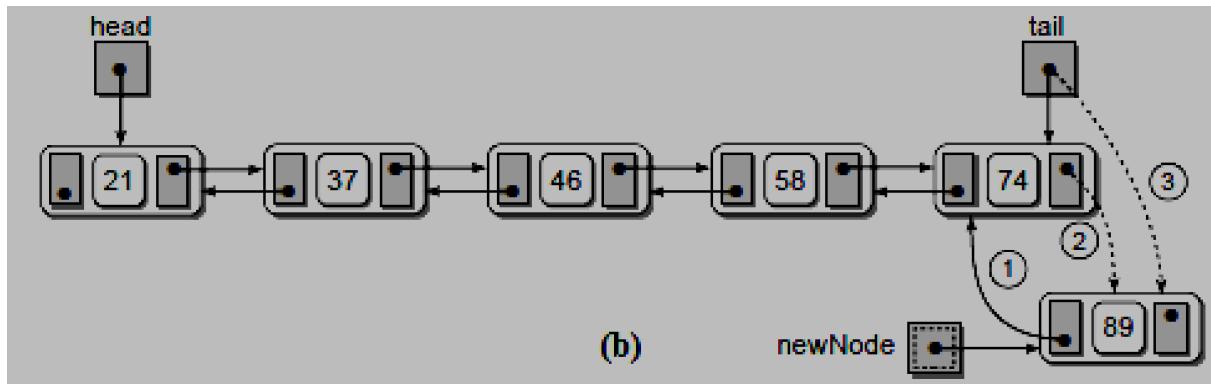
```
if value < head.isi:  
    newnode.next = head      #1  
    head.back = newnode      #2  
    head = newnode           #3
```



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

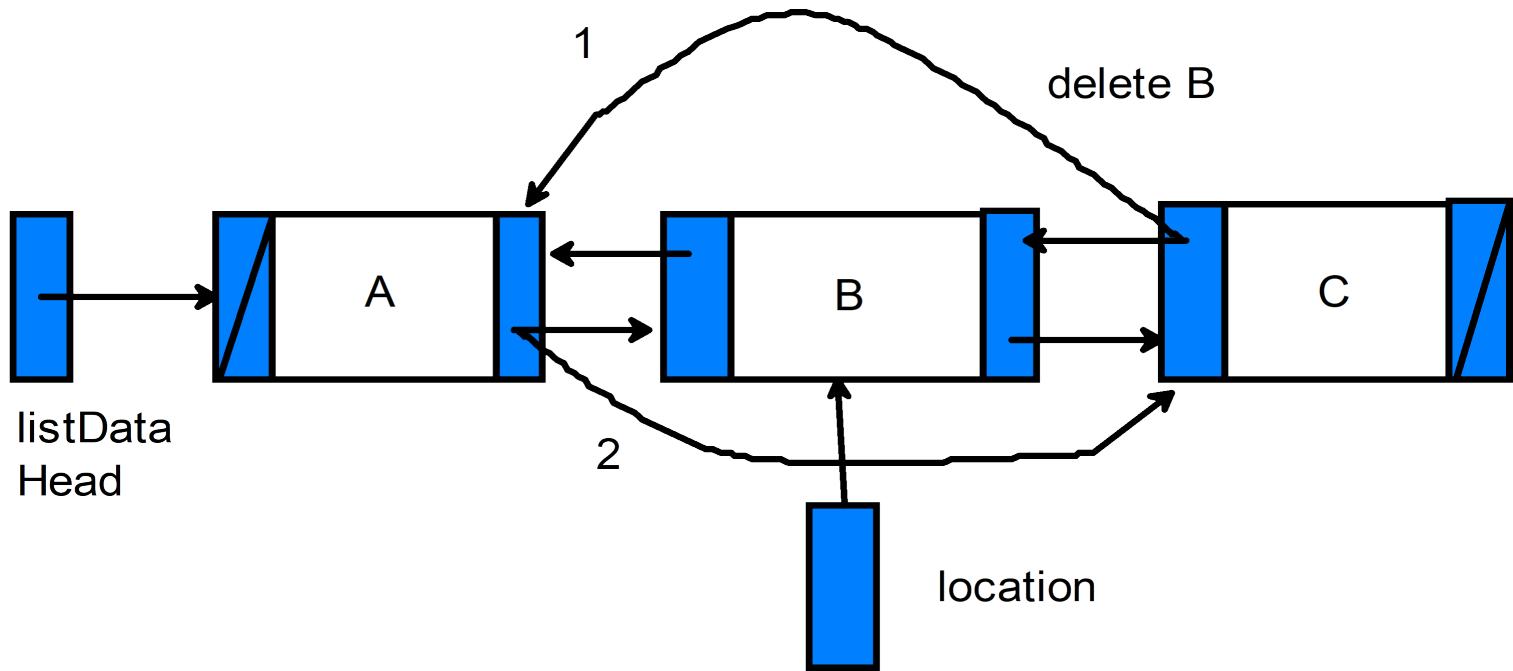
```
newnode = DListNode( value )
if head is None :           # empty list
    head = newnode
    tail = head
elif value < head.data :   # insert before head
    newnode.next = head
    head.prev = newnode
    head = newnode
elif value > tail.data :   # insert after tail
    newnode.prev = tail
    tail.next = 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 tersebut (dipose(location)). Langkah menghapus 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

```
#my_DLList.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")
```

```
#my_DLList_test.py
```

```
from my_DLList 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_DLis.py =====
Add first: 1 2 3
Add last: 4 5
Isi DLList:
3 <=> 2 <=> 1 <=> 4 <=> 5 <=> None
Add 6 before 1
Add 7 after 4
Isi DLlist
3 <=> 2 <=> 6 <=> 1 <=> 4 <=> 7 <=> 5 <=> None
delete 4
Isi DLList:
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 _DoublyLinkedListBase:  
    #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`
- cakupan `_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 caca `_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.""""
    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);

```
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

```
#_DoublyLinkedListBase.py == @Suarga,
#implementasi class dari Untaian Ganda
class _DoublyLinkedListBase:
    #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 _DoublyLinkedListBase

```
1 #LinkedDeque.py == Aplikasi Deque $Suarga
2 from _DoublyLinkedListBase import _DoublyLinkedListBase
3 #LinekdDeque mewarisi (inheritance) _DoublyLinkedListBase
4 class LinkedDeque(_DoublyLinkedListBase): # note the use of inheritance
5     #Double-ended queue implementation based on a doubly linked list.""""
6
7         def first(self):
8             #Return, do not remove the element at the front
9             if self.is_empty():
10                 raise Empty('Deque is empty')
11             return self._header._next._element # item just after header
12
13         def last(self):
14             #Return,do not remove the element at the back
15             if self.is_empty():
16                 raise Empty('Deque is empty')
17             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
```

```

#LinkedDeque_test.py == @Suarga
#mencoba aplikasi LinkedDeque
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 _DoublyLinkedListBase import _DoublyLinkedListBase
#inheritance dari _doublyLinkedListBase
class PositionalList(_DoublyLinkedListBase):
    #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
```

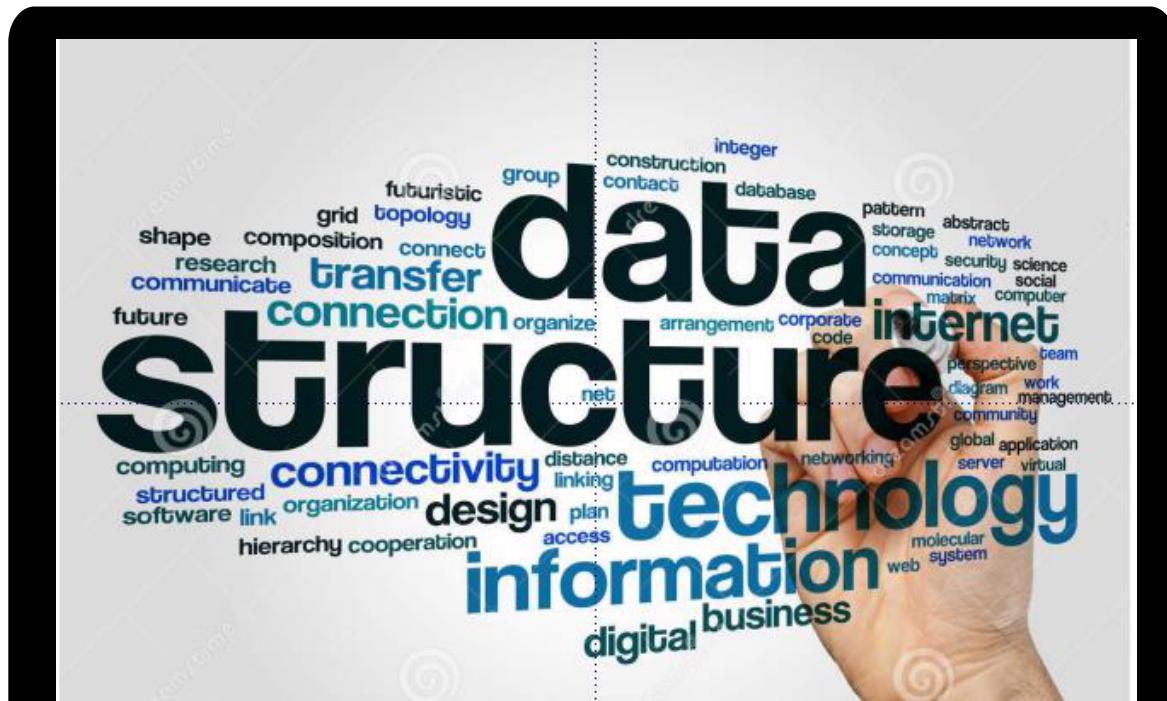
test positional.list

```
· #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
·
10 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)
20 print("ini elemen pertama sekarang")
- k=L.first()
- print(k.element())
- print("elemen berikutnya:")
24 k=L.after(k)
- print(k.element())
- print("Isi List : ")
- traverse()
- print("hapus satu elemen")
- print(L.delete(k))
30 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 berikutnya:
5
Isi List :
7
5
6
8
hapus satu elemen
5
Isi list
7
6
8
>>> |
```

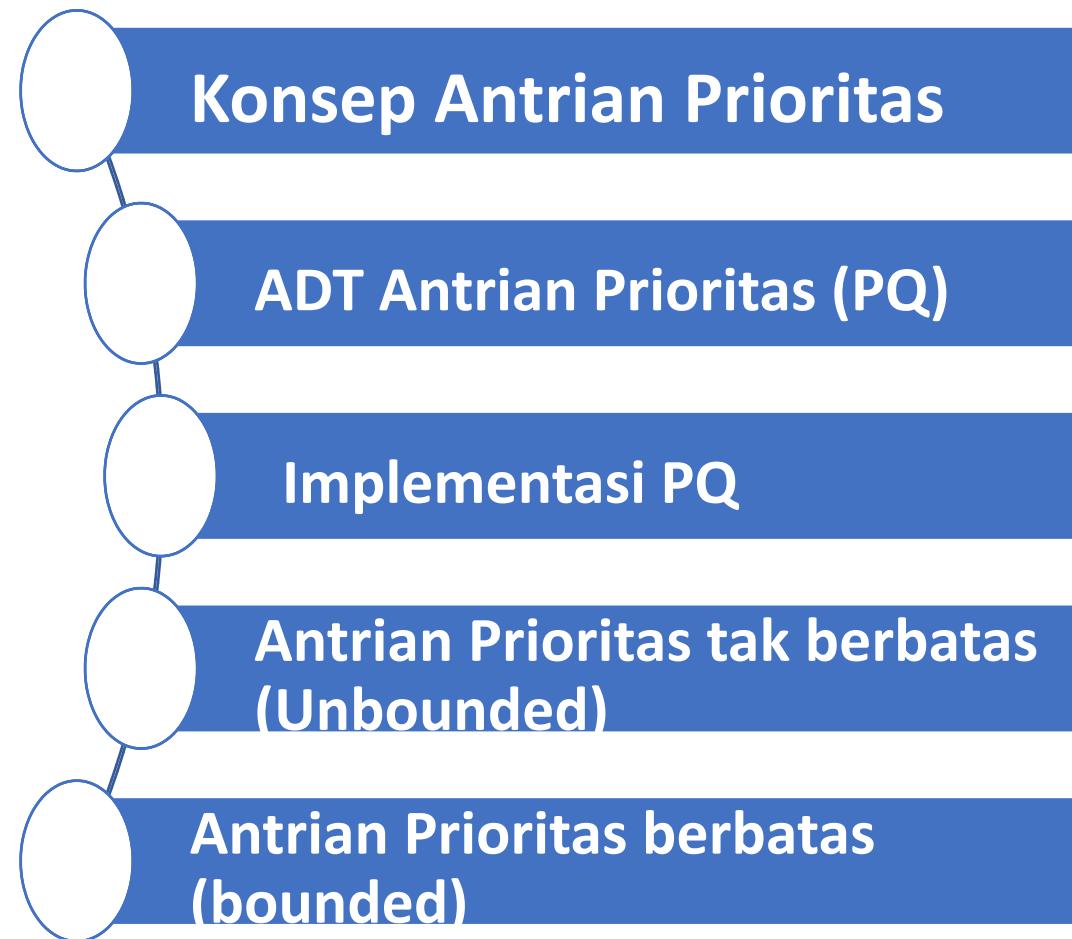


STRUKTUR DATA (PYTHON)

“Antrian Prioritas”

[@SUARGA] | [Pertemuan 12]

OutLine





Konsep Antrian Prioritas

- Antrian (queue) biasanya memiliki kebijakan FIFO, yang pertama datang akan dilayani terlebih dahulu, namun
- Antrian Prioritas (Priority Queue) adalah antrian yang dilengkapi dengan skala prioritas, jadi selain menurut urutan, prioritas juga mendapat perhatian utama.
- contoh: antrian naik pesawat terbang, biasa orang-orang penting mendapat prioritas utama mengalahkan orang yang sudah antri
- Selanjutnya data akan diurutkan bukan menurut kedatangan-nya (bukan FIFO) tetapi menurut urutan prioritas atau kunci k, dengan urutan ascending.

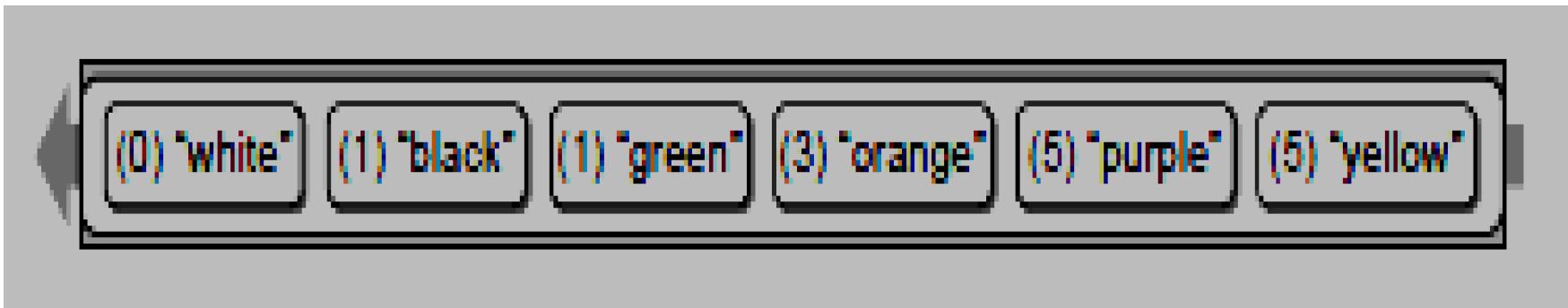
Priority Queue ADT

Beberapa fungsi ADT dari antrian prioritas :

- **add(k, v)** : menambahkan item v dengan prioritas k
- **min()** : menampilkan data (k,v) dimana nilai k terkecil
- **remove_min()** : memberi pelayanan pada data dengan k minimum
- **is_empty()** : memeriksa apakah antrian kosong
- **len(P)** : menghitung banyaknya item dalam antrian prioritas P

Catatan: antrian diberi angka 0 sd n, dimana 0 adalah prioritas tertinggi.

- **Contoh operasi dari fungsi-fungsi ADT :**
- P=PQ () #inisialisasi Priority Queue (PQ)
- P.add(5, 'purple') #data 'purple' dengan prioritas 5
- P.add(1, 'black') #data 'black' dengan prioritas 1
- P.add(3, 'orange') # dan seterusnya ...
- P.add(0, 'white')
- P.add(1, 'green')
- P.add(5, 'yellow')
- P.min() #mencari k min, hasilnya (0, 'white')
- len(P) #banyaknya item, hasilnya 6



Data akan disimpan dengan urutan prioritas, misalnya ‘white’ walaupun dimasukkan belakangan tetapi di-simpan didepan karena prioritas-nya 0, adalah prioritas tertinggi. Apabila prioritas sama maka data disimpan sesuai urutannya dimasukkan, misalnya (1, ‘black’) lebih dulu dari (1,’green’), karena (1, black) lebih dahulu dimasukkan.

- P.remove_min()
 - #mengeluarkan (0, 'white') dari antrian
 - len(P) # panjang = 5
-
- #mengambil data secara ber-urut
 - while not P.is_empty() :
 - item = P.remove_min()
 - print(item)
 -
 - Hasilnya:
 - black
 - green
 - orange
 - purple
 - yellow

Model PriorityQueue (PQ)

- Item / anggota antrian prioritas (PQ) memiliki dua elemen penting:
 - **key** : kunci atau prioritas
 - **value** : nilai data
- Maka basis class (PQBase class) dari antrian prioritas harus memuat kedua elemen tersebut

class PQBase

```
class PQBase:  
    # kelas abstrak untuk priority queue  
  
    class _Item:  
        __slots__ = '_key' , '_value'  
  
        def __init__(self, k, v):  
            self._key = k  
            self._value = v  
  
        def __lt__(self, other):  
            #membandingkan prioritas k  
            return self._key < other._key  
  
    def is_empty(self):    # apakah kosong ?  
        return len(self) == 0
```

Implementasi PQ tak-berurutan

- Pada implementasi tak-berurutan (UnsortedPQ) semua node baru akan dimasukkan seperti pada proses enqueue struktur antrian biasa (FIFO), dimana node yang baru ditambahkan dari belakang antrian, dengan demikian **proses penambahan node menjadi cepat**, karena tidak perlu menelusuri antrian untuk mencari tempat yang sesuai.
- Namun **proses untuk mencari suatu kunci tertentu atau proses untuk mengambil satu node untuk dilayani (remove_min) akan memerlukan waktu lebih lama**, karena node tidak ber-urut menurut kunci prioritas minimum maka antrian harus di-telusuri untuk mendapatkan posisi prioritas minimum.

UnsortedPQ Implementation

```
#UnsortedPQ.py == @Suarga
from PQBase import PQBase
from PositionalList import PositionalList

class UnsortedPQ(PQBase):

    def _find_min(self):
        # mencari node dengan kode prioritas minimum
        if self.is_empty():
            raise Exception('Antrian kosong')
        small = self._data.first()
        walk = self._data.after(small)
        while walk is not None:
            if walk.element() < small.element():
                small = walk
            walk = self._data.after(walk)
        return small

    def __init__(self):
        # inisialisasi dengan positional-list
        self._data = PositionalList()
```

```
def is_empty(self):
    return (len(self._data) == 0)

def __len__(self):
    # memberikan panjang list
    return len(self._data)

30 def add(self, key, value):
    # menambah data
    self._data.add_last(self._Item(key,value))

def min(self):
    # mencari data minimum
    p = self._find_min()
    item = p.element()
    return (item._key, item._value)

40 def remove_min(self):
    # membuang data minimum
    p = self._find_min()
    item = self._data.delete(p)
    return (item._key, item._value)
```

```
def traverse(self):
    walk = self._data.first()
    while walk is not None:
        item = walk.element()
        print(item._key, item._value)
        walk = self._data.after(walk)

50

def testUSPQ():
    USPQ = UnsortedPQ()
    USPQ.add(3, 'red')
    USPQ.add(5, 'purple')
    USPQ.add(0, 'white')
    USPQ.add(4, 'green')
    USPQ.add(2, 'blue')
    USPQ.add(1, 'yellow')
    n = USPQ.__len__()
    print('Jumlah elemen = ', n)
    print('Isi PQ : ')
    USPQ.traverse()
    (mink, minv) = USPQ.min()
    print('elemen minimum key = ', mink, ' isi = ', minv)
    (mink, minv) = USPQ.remove_min()
    print('elemen yg diambil key = ', mink, ' isi = ', minv)
    print('Isi PQ setelah remove : ')
    USPQ.traverse()

70

if __name__ == '__main__':
    testUSPQ()
```

Hasil Test

Python Interpreter

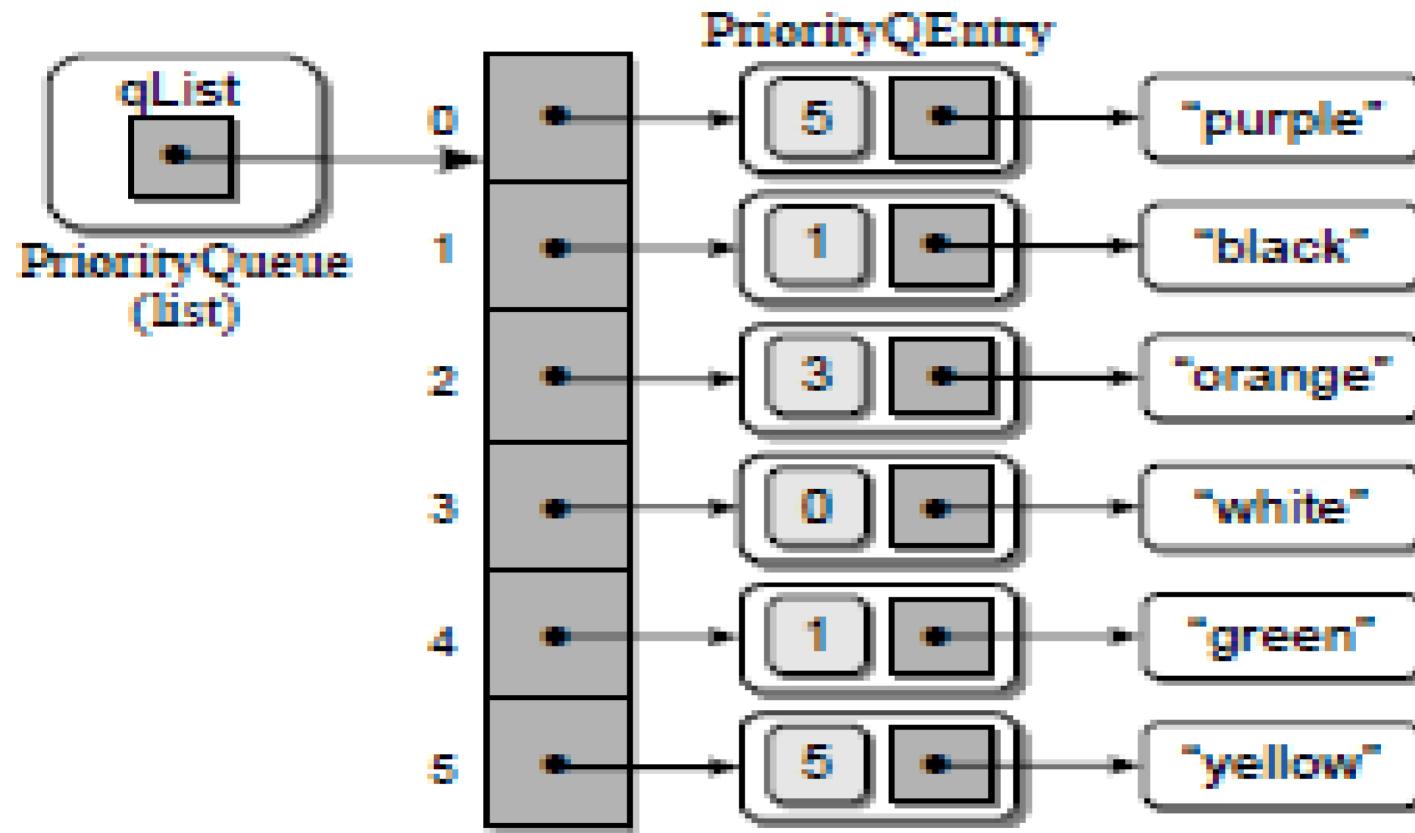
```
>>>
*** Remote Interpreter Reinitialized ***
Jumlah elemen =  6
Isi PQ :
3 red
5 purple
0 white
4 green
2 blue
1 yellow
elemen minimum key =  0  isi =  white
elemen yg diambil key =  0  isi =  white
Isi PQ setelah remove :
3 red
5 purple
4 green
2 blue
1 yellow
>>> |
```

UnboundedPQ

- Antrian Prioritas tak berbatas (unbounded priority queue), dapat diciptakan dengan memanfaatkan Python struktur data internal Python yaitu List().
- Menggunakan struktur List() membuat implementasi ini tidak dibatasi berapa banyak anggota / item yang boleh disimpan
- Namun PQ ini bersifat unsortedPQ

Contoh: PQ Python List Implementation

Andaikan suatu Priority queue akan dibentuk dengan enam item didalamnya seperti pada gambar berikut ini.



Maka perintah untuk membentuk antrian prioritas ini bisa dilakukan sebagai berikut:

- from PriorityQueue import *
- #testPriorityQueue
- Q=PriorityQueue()
- Q.add('purple',5)
- Q.add('black',1)
- Q.add('orange',3)
- Q.add('white',0)
- Q.add('green',5)
- Q.add('yellow',1)
- print('size = ',Q.__len__())
- print('Items in order: ')
- while not Q.isEmpty():
- item = Q.remove_item()
- print(item)

Hasil percobaan diatas menghasilkan berikut ini.

- size = 6
 - Items in order:
 - white
 - black
 - yellow
 - orange
 - purple
 - green
- >>>

```
1 # Implementasi ADT antrian prioritas tdk-berbatas (unbounded)
# memakai Python list, dimana item baru ditambahkan di belakang.
#
2 class PriorityQueue :
3     # mencipta unbounded priority queue kosong.
4         def __init__( self ):
5             self._qList = list()
6
7     # memeriksa apakah PQ kosong?
8         def isEmpty( self ):
9             return len( self ) == 0
10
11     # memberikan jumlah item dalam PQ
12         def __len__( self ):
13             return len( self._qList )
14
15     # menambah entri baru ke dalam PQ
16         def add( self, item, priority ):
17             # Create a new instance of the storage class
18             # and append it to the list.
19                 entry = _PriorityQEntry( item, priority )
20                 self._qList.append( entry )
21                 #self._qList.insert(priority, entry)
```

```
# mencari item dengan prioritas tertinggi (key minimum)
def min( self ) :
    assert not self.isEmpty(),"Cannot dequeue from an empty queue."
    # cari entri dengan prioritas tertinggi.
    ix=0
    highest = self._qList[0].priority
    size = len(self._qList)
    for i in range( size ) :
        # periksa apakah yang ke-i tertinggi
        # priority (smaller integer).
        if self._qList[i].priority < highest :
            highest = self._qList[i].priority
            ix=i
    entry = self._qList[ix]
    # tampilkan yang tertinggi prioritas-nya.
    return (entry.priority, entry.item)

# Removes and returns the first item in the queue.
def remove_item( self ) :
    assert not self.isEmpty(),"Cannot dequeue from an empty queue."
    # cari entri dengan prioritas tertinggi.
```

```
# cari entri dengan prioritas tertinggi.  
ix=0  
highest = self._qList[0].priority  
size = len(self._qList)  
for i in range( size ) :  
    # periksa apakah yang ke-i tertinggi  
    # priority (smaller integer).  
    if self._qList[i].priority < highest :  
        highest = self._qList[i].priority  
        ix=i  
  
    # ambil/keluarkan yang tertinggi prioritas-nya.  
entry = self._qList.pop( ix )  
return (entry.priority, entry.item)  
  
def traverse(self):  
    #menampilkan isi PQ  
    n = len(self._qList)  
    for i in range(n) :  
        print (self._qList[i].priority, self._qList[i].item)
```

```
· # Private storage class for associating queue
· # items with their priority.
70 · class _PriorityQEntry( object ):
·     def __init__( self, item, priority ):
·         self.item = item
·         self.priority = priority
74
·     def testPQ():
·         Q = PriorityQueue()
·         Q.add('purple',5)
·         Q.add('black',1)
·         Q.add('orange',3)
·         Q.add('white',0)
·         Q.add('green',1)
·         Q.add('yellow',5)
·         n = Q.__len__()
·         print('Jumlah elemen = ',n)
·         print('Isi PQ : ')
·         Q.traverse()
·         (mink,minv) = Q.min()
·         print('elemen minimum key = ',mink,' isi = ',minv)
```

```
    (mink,minv) = Q.min()
    print('elemen minimum key = ',mink,' isi = ',minv)
89   (mink,minv) = Q.remove_item()
90   print('elemen yg diambil key = ',mink,' isi = ',minv)
    print('Isi PQ setelah remove : ')
    Q.traverse()
    print('elemen sisa urut menurut key : ')
    print('Items in order: ')
    while not Q.isEmpty():
        (key,item) = Q.remove_item()
        print(key,item)

100 if __name__ == '__main__':
     testPQ()
```

Hasil Test

Python Interpreter

```
*** Remote Interpreter Reinitialized ***
Jumlah elemen =  6
Isi PQ :
5 purple
1 black
3 orange
0 white
1 green
5 yellow
elemen minimum key =  0  isi =  white
elemen yg diambil key =  0  isi =  white
Isi PQ setelah remove :
5 purple
1 black
3 orange
1 green
5 yellow
elemen sisa urut menurut key :
Items in order:
1 black
1 green
3 orange
5 purple
5 yellow
>>> |
```

PQ berbatas (Bounded PQ)

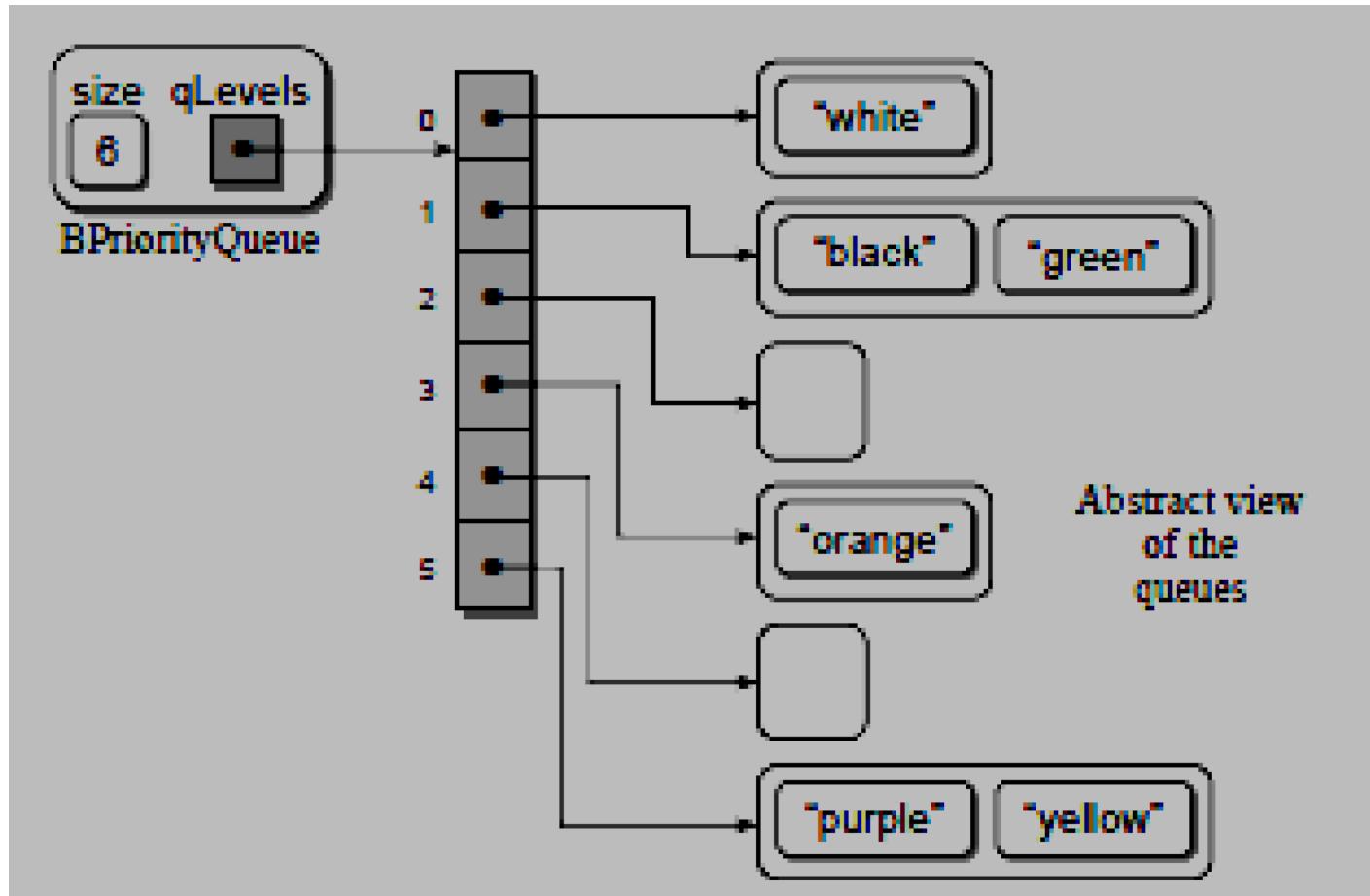
- **Bounded-Priority Queue** adalah antrian prioritas yang terbatas, dimana prioritas dibatasi maksimal sama dengan suatu integer p , atau $[0 \dots p]$.
- Antrian ini yang diberi nama **BpriorityQueue** di-implementasikan memakai **LinkedList**, dengan ADT yang sama dengan antrian prioritas biasa.
- Pada implementasi-nya, constructor menciptakan dua data field, yaitu larik (array) yang diberi nama **_qLevels**, untuk menyimpan prioritas dan berfungsi menjadi head ke antrian linked-list, yaitu **LinkedQueue()**. **LinkedQueue** akan di-isi dengan item/data sesuai urutan pemasukan-nya ke dalam antrian.

Contoh

- BPQ berikut andaikan dibatasi hanya 6 (0..5) prioritas

```
BP=PriorityQueue(6)
BP.add(3, 'orange')
BP.add(1, 'black')
BP.add(0, 'white')
BP.add(5, 'purple')
BP.add(1, 'green')
BP.add(5, 'yellow')
```

Hasil-nya



Implementasi BPQ

```
# Implementation of the bounded Priority Queue ADT using an array of
# queues in which the queues are implemented using a linked list.
# BPriorityQueue.py == @Suarga
from Array import *
#from llistqueue import *
from LinkedQueue import *

class BPriorityQueue :
    # Creates an empty bounded priority queue.
    def __init__( self, numLevels ):
        self._qSize = 0
        self._qLevels = Array( numLevels )
        for i in range( numLevels ) :
            self._qLevels[i] = LinkedQueue()

    # Returns True if the queue is empty.
    def isEmpty( self ) :
        return len( self ) == 0
```

```
20     # Returns the number of items in the queue.
21     def __len__( self ):
22         return self._qSize
23
24     # Adds the given item to the queue.
25     def add( self, priority, item ):
26         assert priority >= 0 and priority < len(self._qLevels), \
27             "Invalid priority level."
28         self._qLevels[priority].enqueue( item )
29         self._qSize += 1
30
31     # Removes and returns the next item in the queue.
32     def remove_min( self ) :
33         # Make sure the queue is not empty.
34         assert not self.isEmpty(), "Cannot dequeue from an empty queue."
35         # Find the first non-empty queue.
36         i = 0
37         p = len(self._qLevels)
38         while i < p and self._qLevels[i].is_empty() :
39             i += 1
40
41         # We know the queue is not empty, so dequeue from the ith queue.
42         self._qSize -= 1
43         return self._qLevels[i].dequeue()
```

test BPriorityQueue

```
- BP=BPriorityQueue(6)
- BP.add(3,'orange')
- BP.add(1,'black')
- BP.add(0,'white')
50 BP.add(5,'purple')
- BP.add(1,'green')
- BP.add(5,'yellow')

- print('size = ', BP.__len__())
- n = len(BP._qLevels)
- print('\n Isi Priority Queue: ')
- for i in range(n):
-     BP._qLevels[i].traverse()

60 print('\n Isi dihapus dengan urutan : ')
- while not BP.is_Empty():
-     item = BP.remove_min()
-     print(item)

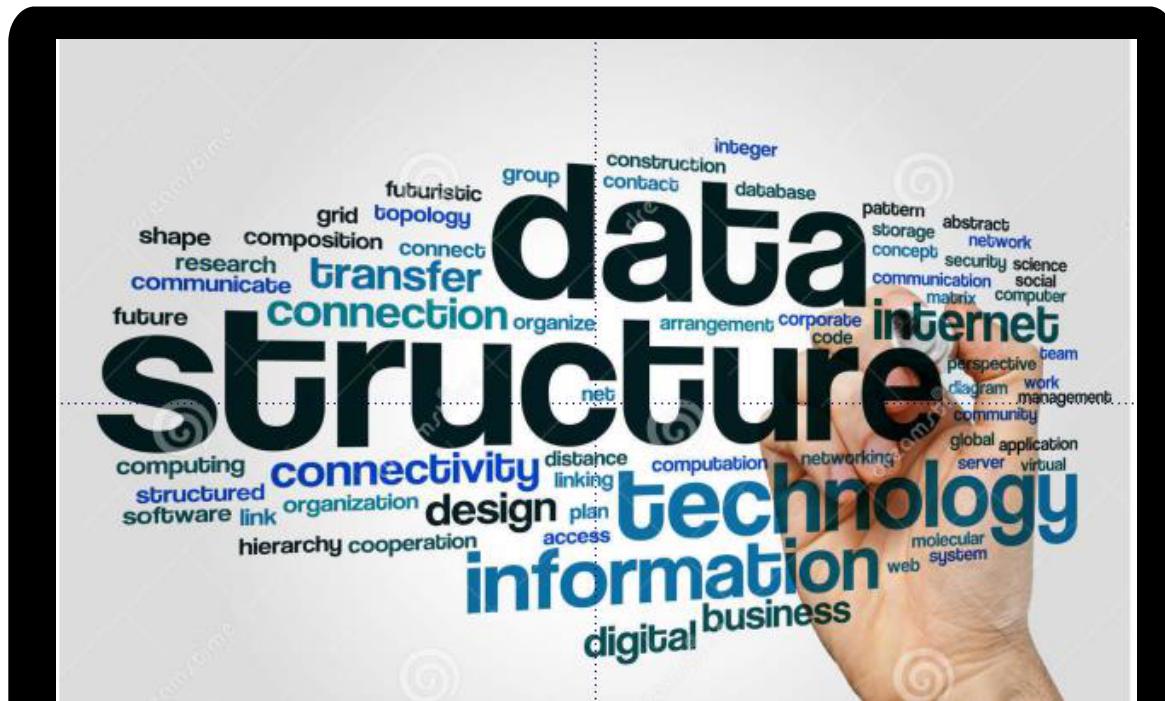
- print('size setelah isi dibuang = ', BP.__len__())
```

Hasil test:

```
>>>
*** Remote Interpreter Reinitialized ***
size = 6

    Isi Priority Queue:
white
black
green
orange
purple
yellow

    Isi dihapus dengan urutan :
white
black
green
orange
purple
yellow
size setelah isi dibuang = 0
>>>
```

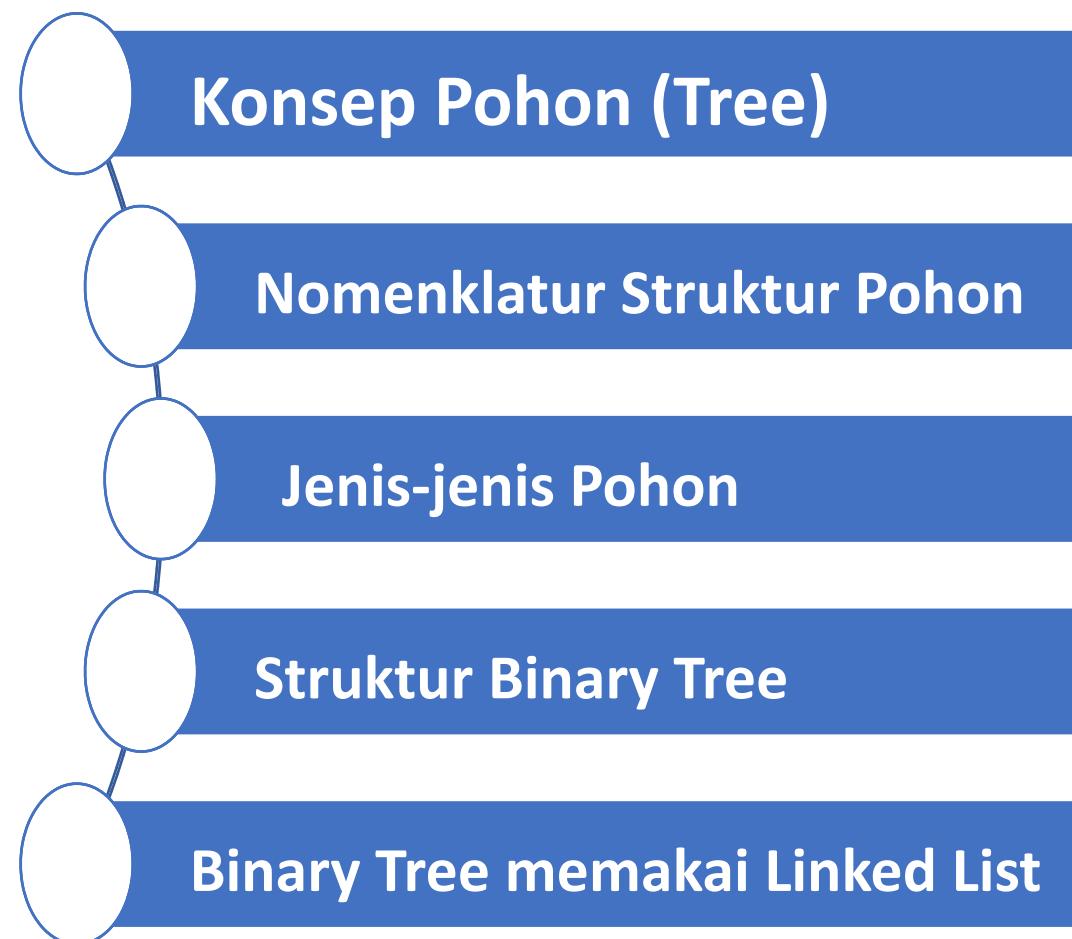


STRUKTUR DATA (PYTHON)

“Struktur Pohon Biner”

[@SUARGA] | [Pertemuan 13]

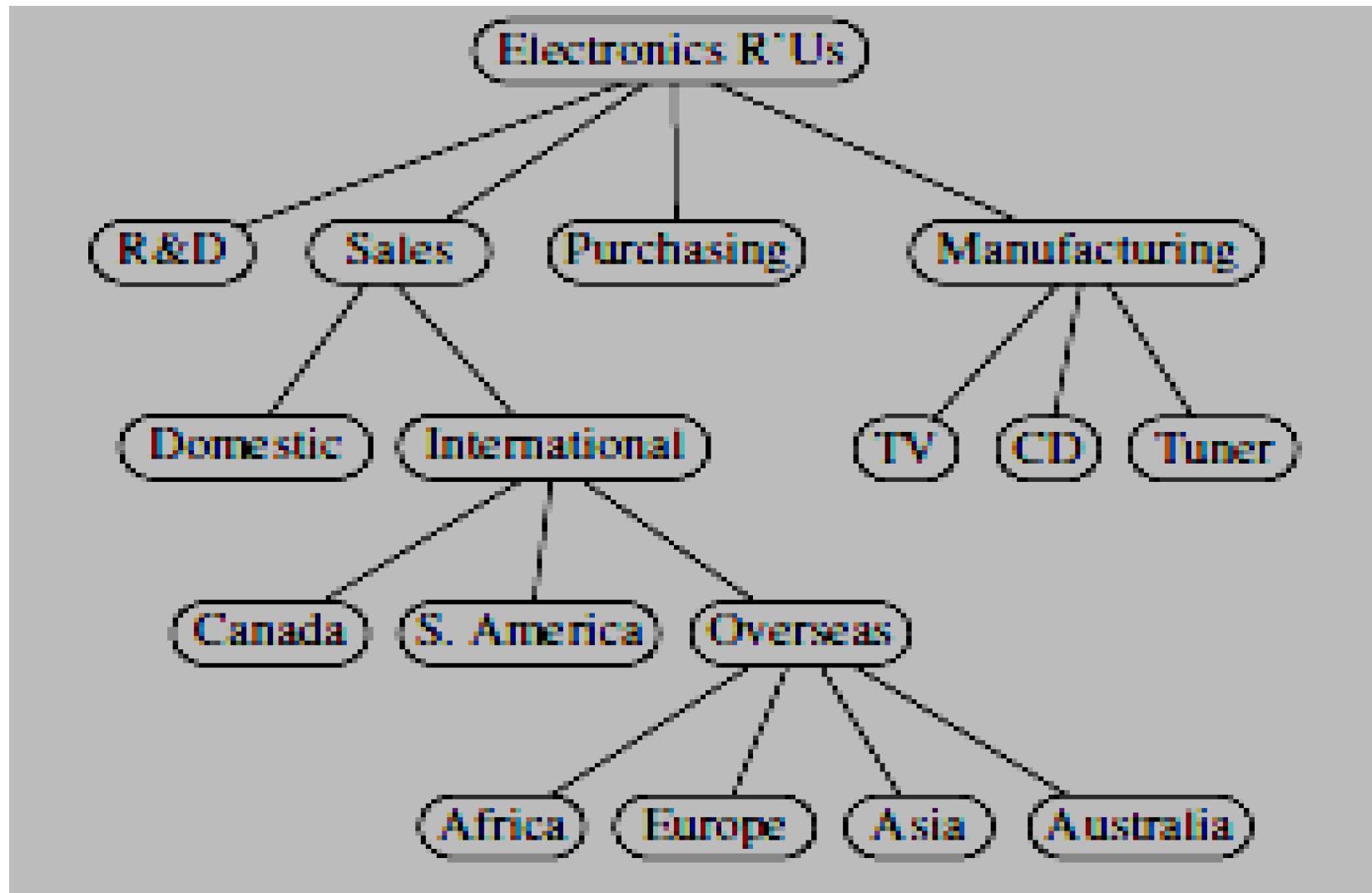
OutLine





Konsep Struktur Pohon (Tree)

- Struktur pohon (tree) adalah abstraksi tipe data yang menyimpan elemen-nya secara hirarki, dari akar (root) hingga ke daun (leaves), namun secara grafis struktur pohon justru digambar sebagai pohon terbalik, dimana akar berada diatas dan daun berada dibawah.
- Struktur pohon digambarkan seperti hirarki suatu organisasi, dimana ada Direktur sebagai akar kemudian dibawah-nya ada beberapa Kepala-Bagian dan seterusnya kebawah hingga ke hirarki paling bawah, para Staff.



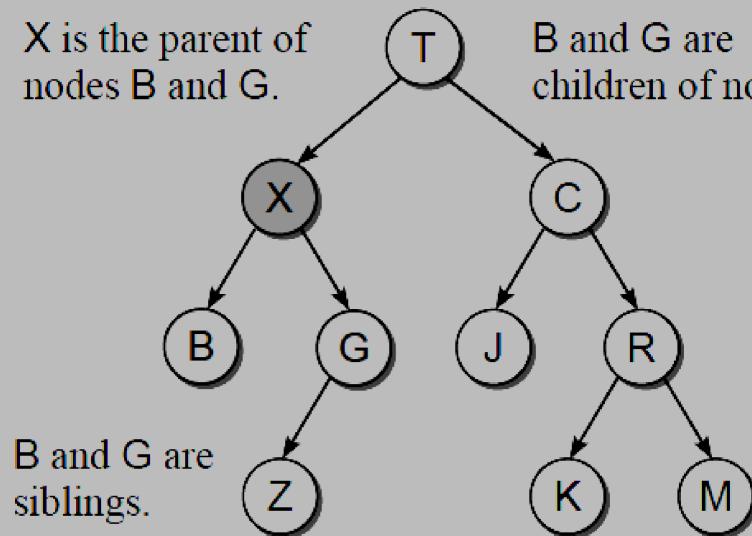
Nomeklatur Pohon

- **Root** : akar, adalah elemen (node) paling atas, merupakan pintu atau pointer untuk meng-akses elemen lain dalam pohon. Akar adalah satu-satunya node yang tidak memiliki ‘atasan’ atau parent. Ketika suatu pohon mulai dibentuk maka node pertama yang harus diciptakan adalah akar (root) ini.
- **Path** : jalur yang menghubungkan dari node asal ke suatu node tujuan melalui beberapa penggal penghubung yang disebut **edge** (dahan).

- **Parent** : atasan, induk, merupakan node yang berada diatas suatu node tertentu. Suatu node dibatasi hanya memiliki satu node atasan.
- **Children** : anak atau bawahan, merupakan node yang berada dibawah satu node. Satu node bisa memiliki lebih dari satu bawahan.
- **Subtree** : suatu pohon bisa dibagi menjadi beberapa bagian-pohon (subtree), sehingga suatu node bisa merupakan root dari subtree-nya.
- **Node** : elemen dari pohon, setiap node yang memiliki minimal satu anak disebut **interior-node** dan node yang tidak memiliki anak disebut **leaf node** (daun).

X is the parent of nodes B and G.

B and G are children of node X.

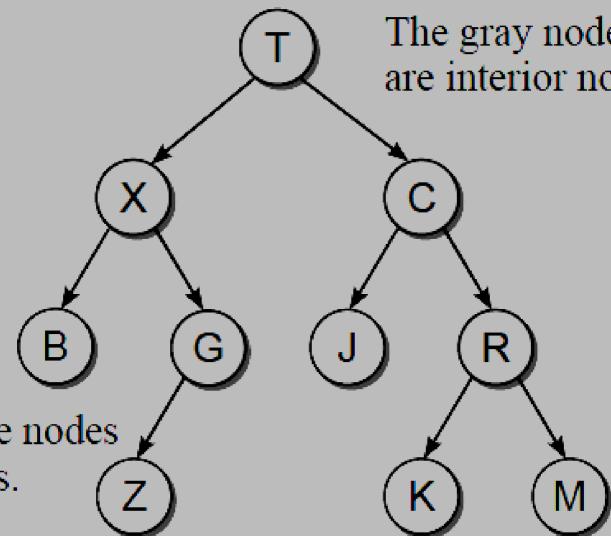


B and G are siblings.

(a)

The gray nodes are interior nodes.

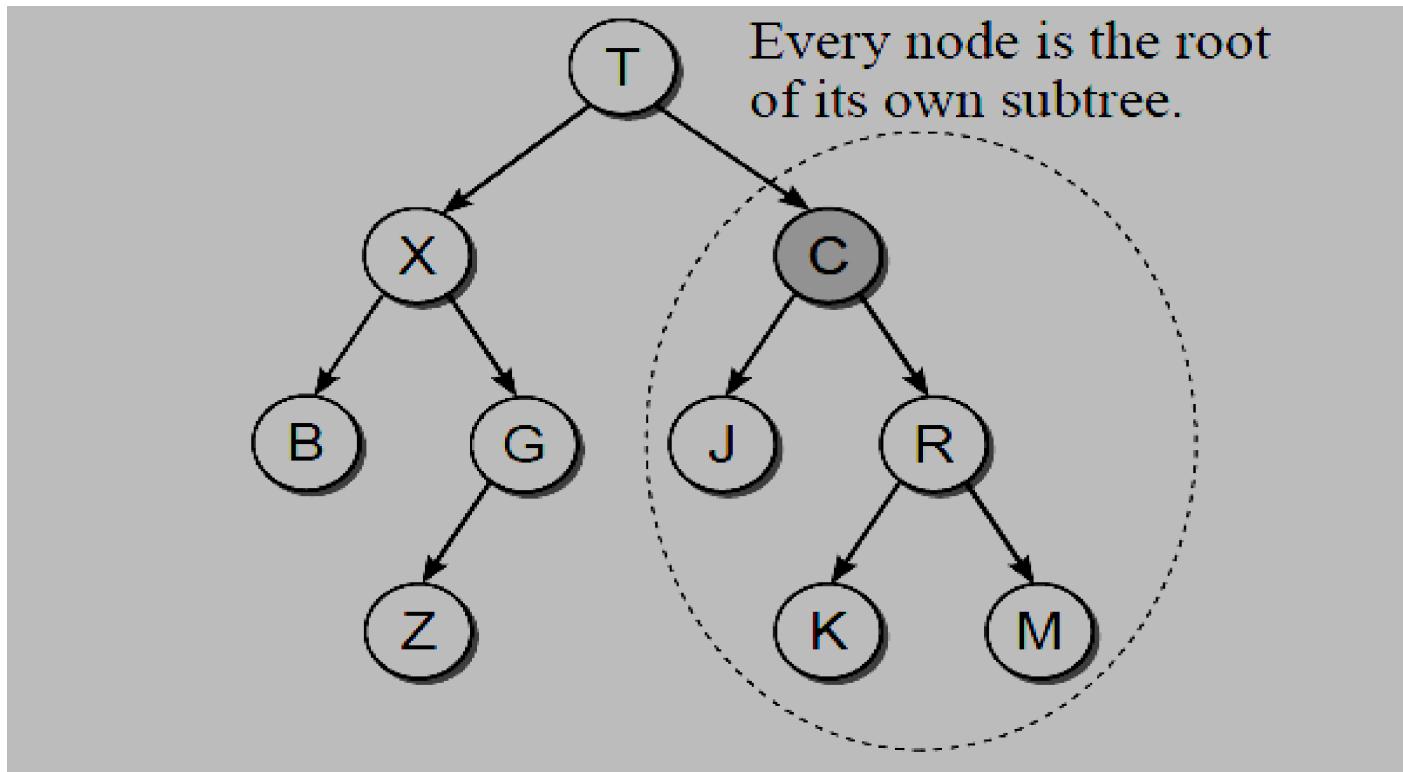
The white nodes are leaves.



(b)

T =root, X=parent, B dan G = children

Node abu-abu = interior-node, node putih = leaf node



Subtree, Node C adalah root dari subtree-nya

Jenis-Jenis Tree

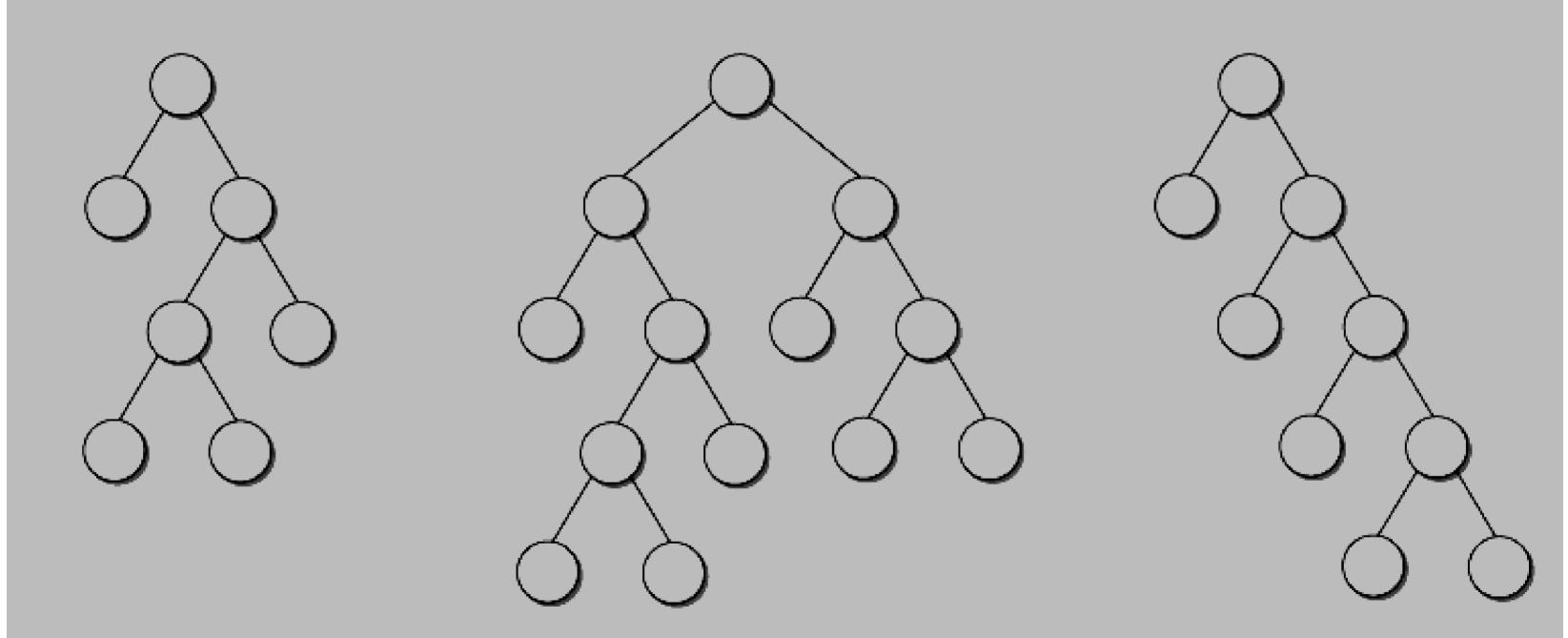
- Setiap pohon yang memiliki node dengan maksimum anak sebanyak m disebut **m -nary tree**.
- Satu jenis pohon yang paling banyak dibicarakan dalam struktur data, adalah pohon yang setiap node-nya hanya memiliki maksimum dua (bi) anak, disebut **binary-tree**.
- **Binary Tree** : pohon biner dengan maksimum dua anak
- **AVL** : suatu pohon biner yang seimbang yang dipakai untuk pencarian data (balanced binary search tree) yang memiliki sifat khas, yaitu tinggi (height) dari dua cabang dari suatu node maksimal hanya berbeda satu.

- **BST** : Binary-Search-Tree, suatu pohon biner yang seimbang dengan karakteristik tertentu. Setiap node hanya boleh memiliki maksimum dua anak cabang.
- **Red-Black Tree** : suatu pohon biner seimbang, dimana node-nya diberi warna merah atau hitam, mengikuti suatu aturan.
- **2-3- Tree** : suatu pohon seimbang yang memiliki aturan yang lebih ketat, setiap node boleh berisi dua kunci, dan maksimum tiga anak.

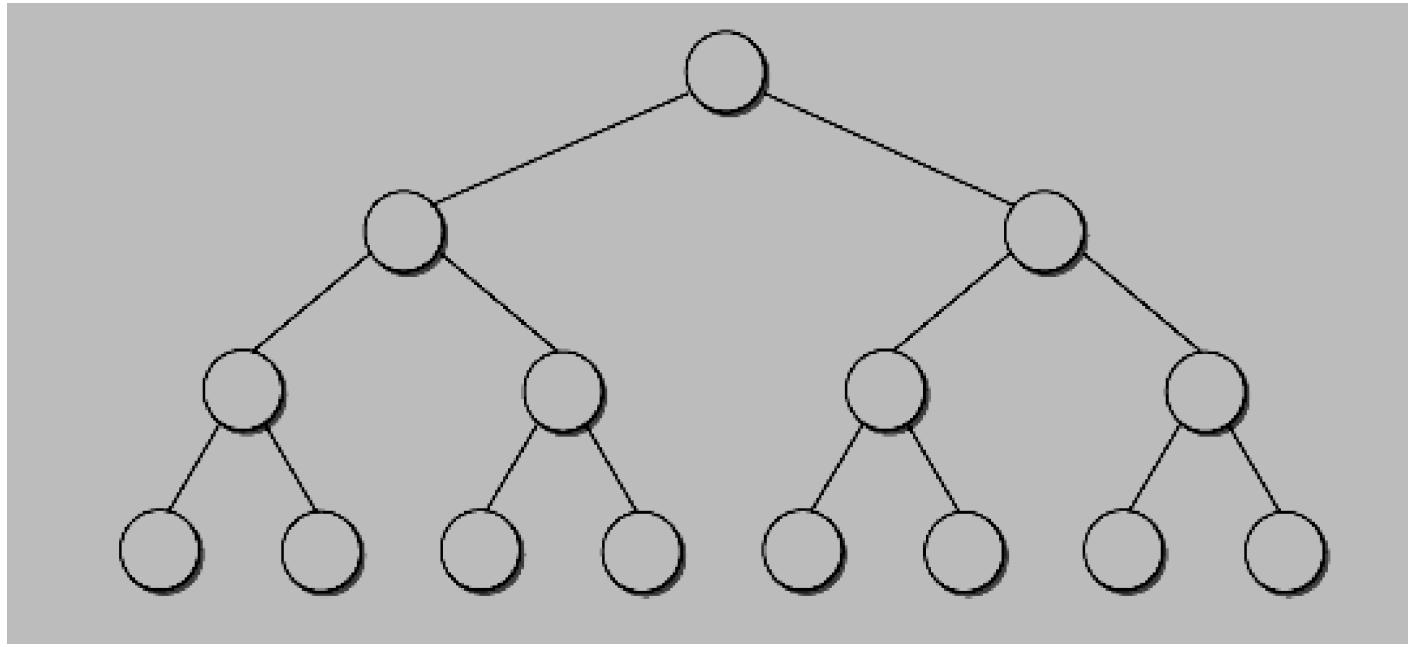
Binary-Tree

- Binary-tree atau pohon-biner adalah pohon dimana setiap vertex / node memiliki maksimum dua anak, satu di sisi kiri dan satu di sisi kanan.
- Dalam Python elemen dasar binary tree didefinisikan sebagai:

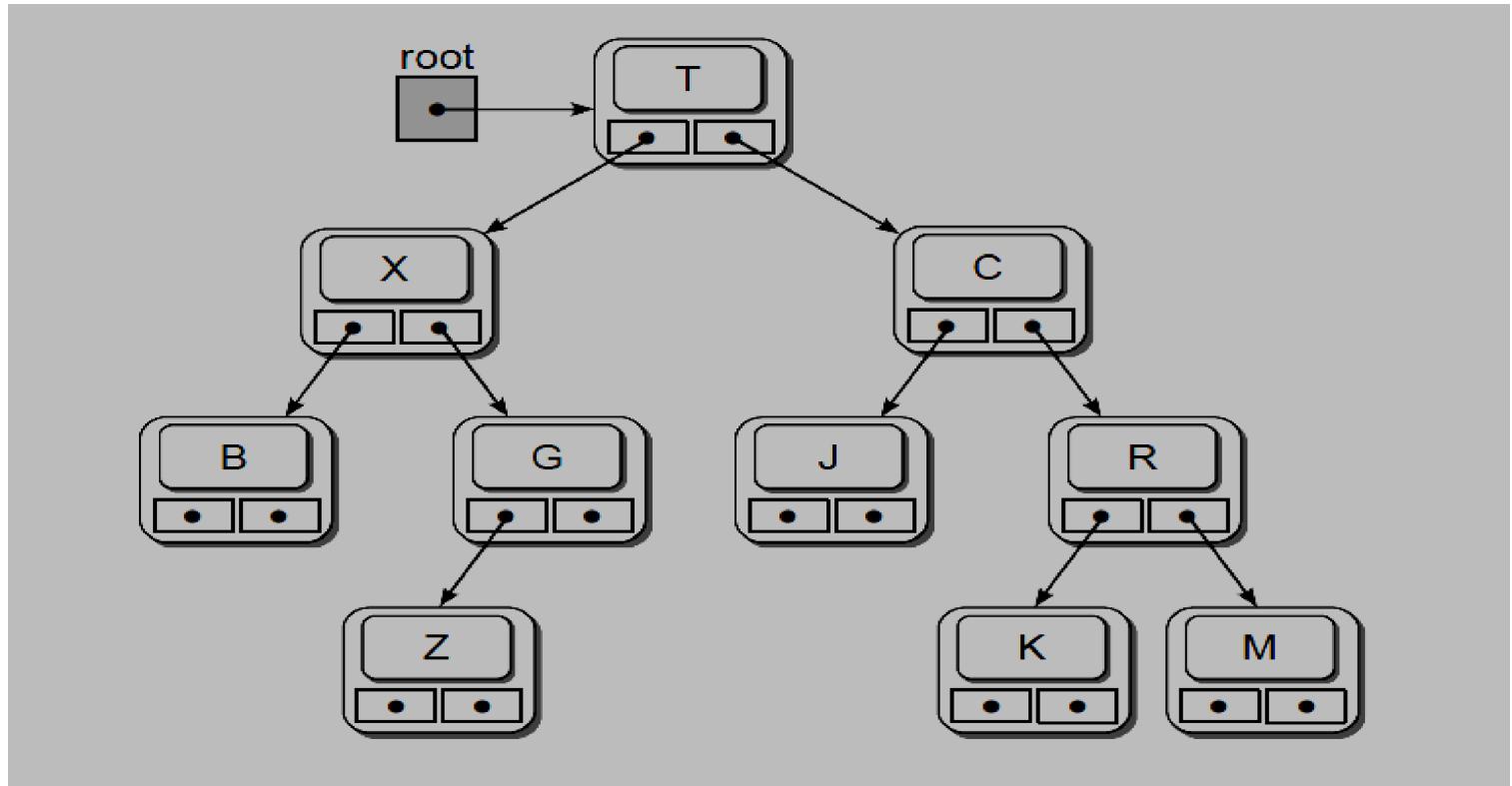
```
class _BinTreeNode :  
    def __init__(self, data):  
        self.data = data  
        self.left = None  
        self.right = None
```



: **Full binary tree**



Perfect binary tree



Implementasi fisik pohon biner

Implementasi Sederhana

Salah satu alternatif implementasi binary tree adalah dengan memakai Python list sebagai struktur dasar.

1. Definisikan binary-tree dengan root r dan sisi-kiri dan sisi-kanan merupakan list yang masih kosong:

```
def BinaryTree(r):
    return [r, [], []]
```

2. Definisikan fungsi untuk mengisi sisi-kiri:

```
def insertLeft(root,newBranch):
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1,[newBranch,t,[]])
    else:
        root.insert(1,[newBranch, [], []])
    return root
```

3. Definisikan fungsi untuk mengisi sisi-kanan:

```
def insertRight(root,newBranch):  
    t = root.pop(2)  
    if len(t) > 1:  
        root.insert(2,[newBranch,[],t])  
    else:  
        root.insert(2,[newBranch,[],[]])  
    return root
```

4. Fungsi untuk membaca nilai root:

```
def getRootVal(root):  
    return root[0]
```

5. Fungsi untuk mengganti nilai root:

```
def setRootVal(root,newVal):  
    root[0] = newVal
```

6. Fungsi untuk mengambil turunan sisi-kiri:

```
def getLeftChild(root):  
    return root[1]
```

7. Fungsi untuk mengambil turunan sisi-kanan:

```
def getRightChild(root):  
    return root[2]
```

Implementasi berbasis list

```
#pbiner.py - pohon biner berbasis list
def BinaryTree(r):
    return [r, [], []]

def insertLeft(root,newBranch):          #sisip di kiri
    t = root.pop(1)
    if len(t) > 1:
        root.insert(1,[newBranch,t,[]])
    else:
        root.insert(1,[newBranch, [], []])
    return root

def insertRight(root,newBranch):         #sisip di kanan
    t = root.pop(2)
    if len(t) > 1:
        root.insert(2,[newBranch,[],t])
    else:
        root.insert(2,[newBranch,[],[]])
    return root

def getRootVal(root):                  #ambil nilai root
    return root[0]

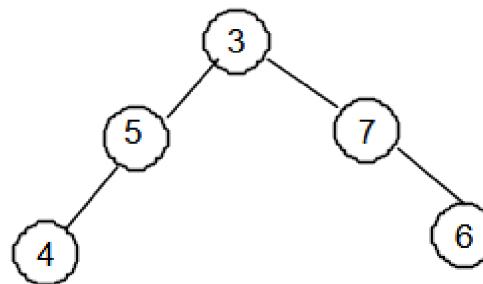
def setRootVal(root,newVal):           #pasang nilai root
    root[0] = newVal

def getLeftChild(root):                #ambil anak di kiri
    return root[1]

def getRightChild(root):               #ambil anak di kanan
    return root[2]
```

Contoh

```
>>> r=BinaryTree(3)                      # root = 3
>>> insertLeft(r,4)                     # sisip 4 di kiri
[3, [4, [], []], []]
>>> insertLeft(r,5)                     # sisip 5 di kiri
[3, [5, [4, [], []], []], []]
>>> insertRight(r,6)                    # sisip 6 di kanan
[3, [5, [4, [], []], []], [6, [], []]]
>>> insertRight(r,7)                    # sisip 7 di kanan
[3, [5, [4, [], []], []], [7, [6, [], []]]]
```

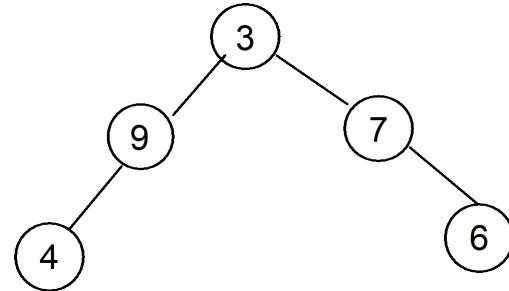


```

>>> getLeftChild(r)                      # mengambil sisi kiri root
[5, [4, [], []], []]
>>> getRightChild(r)                    # mengambil sisi kanan root
[7, [], [6, [], []]]
>>> L = getLeftChild(r)                  # L adakah sisi kiri dari root
>>> L
[5, [4, [], []], []]
>>> setRootVal(L,9)                     # ganti root L (sisi kiri) jadi
9
>>> print(r)
[3, [9, [4, [], []], []], [7, [], [6, [], []]]]

```

Hasil dari pohon biner adalah:



setelah node-5 diganti 9

Penelusuran Pohon Biner

Ada tiga macam cara penelusuran pohon biner, yaitu: preorder, inorder, postorder

- **preorder**, mengunjungi suatu node, kemudian mengunjungi sisi-kirinya, kemudian sisi-kanannya

```
preorder(node)
    visit(node)
    if left(node) <> null then
        preorder(left(node))
    if right(node) <> null then
        preorder(right(node))
```

- ***inorder***, mengunjungi sisi-kiri, kemudian node induk, lalu sisi-kanan

```
inorder(node)
    if left(node) <> null
        then inorder(left(node))
visit(node)
    if right(node) <> null
        then inorder(right(node))
```

- **postorder**, mengunjungi sisi-kiri, sisi-kanannya, lalu node induk

```
postorder(node)
```

```
    if left(node) <> null  
        then postorder(left(node))  
    if right(node) <> null  
        then inorder(right(node))  
    visit(node)
```

Preorder:

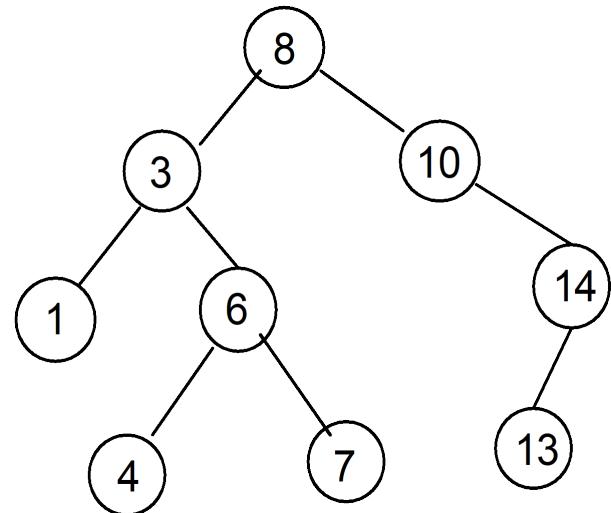
8->3->1->6->4->7->10->14->13

Inorder:

1->3->4->6->7->8->10->13->14

Postorder:

1->4->7->6->3->13->14->10-8



Implementasi BTNode

- Berikut ini disajikan implementasi sebuah pohon biner yang lebih lengkap, diberi nama BTNode.py
- Implementasi ini dilengkapi dengan beberapa fungsi ADT untuk memasukkan data, mencari data, menghapus data, menampilkan isi pohon, dan menampilkan struktur pohon

ADT Binary Tree

- `__init__()` : inisialisasi pohon biner
- `insert()` : menyisip data ke pohon
- `lookup()` : mencari posisi data
- `children_count()` : mengitung anak suatu node
- `delete()` : menghapus satu node
- `print_tree()` : menampilkan susunan node
- `pre_order()` : penelusuran preorder
- `in_order()` : penelusuran inorder
- `post_order()` : penelusuran post order

Implementasi class BTNode

```
1 #BTNode.py - Binary Tree implementation @Suarga
2 import sys
3
4 class BTNode:
5     """
6         Tree node: pointer left/kiri
7             pointer right/kanan
8             data : value dari node
9     """
10    def __init__(self, data):
11        """
12            Node constructor
13            @param data node data object
14        """
15        self.left = None
16        self.right = None
17        self.data = data
18
```

```
19 def children_count(self):
20     """
21     Returns the number of children
22     @returns number of children: 0, 1, 2
23     """
24     if (self.left is None) and (self.right is None):
25         return None
26     cnt = 0
27     if self.left:
28         cnt += 1
29     if self.right:
30         cnt += 1
31     return cnt
32
33 def insert(self, data):
34     """
35     Insert new node with data
36     @param data node data object to insert
37     """
38     if data < self.data:
39         if self.left is None:
40             self.left = BTNode(data)
41         else:
42             self.left.insert(data)
43     else:
44         if self.right is None:
45             self.right = BTNode(data)
46         else:
47             self.right.insert(data)
```

```
48
49     def lookup(self, data, parent=None):
50         """
51             Lookup node containing data
52             @param data node data object to look up
53             @param parent node's parent
54             @returns node and node's parent if found or None, None
55         """
56
57         if data < self.data:
58             if self.left is None:
59                 return None, None
60             return self.left.lookup(data, self)
61         elif data > self.data:
62             if self.right is None:
63                 return None, None
64             return self.right.lookup(data, self)
65         else:
66             return self, parent
```

```
67 def delete(self, data):
68     """
69     Delete node containing data
70     @param data node's content to delete
71     """
72     # get node containing data
73     node, parent = self.lookup(data)
74     if node is not None:
75         children_count = node.children_count()
76         if children_count == None:
77             #print(node.data, 'tdk ada child')
78             # if node has no children, just remove it
79             if parent.left is node:
80                 parent.left = None
81             else:
82                 parent.right = None
83             del node
84         elif children_count == 1:
85             #print(node.data, 'ada child 1')
86             # if node has 1 child
87             # replace node by its child
88             if node.left:
89                 n = node.left
90             else:
91                 n = node.right
```

```
92     if parent:
93         if parent.left is node:
94             parent.left = n
95         else:
96             parent.right = n
97     del node
98 else:
99     #print(node.data, 'ada child > 1')
100    # if node has 2 children
101    # find its successor
102    parent = node
103    successor = node.right
104    #while (successor is not None) and (susccesor.left):
105    while successor.left:
106        parent = successor
107        successor = successor.left
108        # replace node data by its successor data
109    node.data = successor.data
110    # fix successor's parent's child
111    if parent.left == successor:
112        parent.left = successor.right
113    else:
114        parent.right = successor.right
115    del node
116
```

```
117 def print_tree(self):
118     """
119     Print tree content inorder
120     """
121     if self.left:
122         #print('left',)
123         self.left.print_tree()
124     print (self.data,)
125     if self.right:
126         #print('right',)
127         self.right.print_tree()
128
129 def tree_data(self):
130     """
131     Generator to get the tree nodes data
132     """
133     # we use a stack to traverse the tree in a non-recursive way
134     stack = []
135     node = self
136     while stack or node:
137         if node:
138             stack.append(node)
139             node = node.left
140         else: # we are returning so we pop the node and we yield it
141             node = stack.pop()
142             yield node.data
143             node = node.right
```

```
144
145
146 def disp_tree(self, indent_char = '...', indent_delta=2):
147     node = self
148     def disp_tree_1(indent, node):
149         if node == None:
150             return None
151         else:
152             disp_tree_1(indent+indent_delta, node.right)
153             print(indent*indent_char+str(node.data))
154             disp_tree_1(indent+indent_delta, node.left)
155     disp_tree_1(0, node)
156
157
158 def pre_order(self):
159     node = self
160     def preorder(node):
161         if node is not None:
162             sys.stdout.write("%i -> " % node.data)
163             preorder(node.left)
164             preorder(node.right)
165     preorder(node)
166     print("None")
```

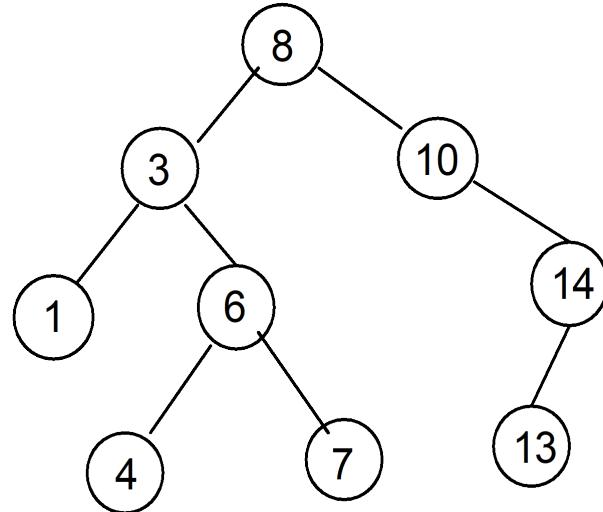
```
167
168     def in_order(self):
169         node = self
170         def inorder(node):
171             if node is not None:
172                 inorder(node.left)
173                 sys.stdout.write("%i -> " % node.data)
174                 inorder(node.right)
175             inorder(node)
176             print("None")
177
178     def post_order(self):
179         node = self
180         def postorder(node):
181             if node is not None:
182                 postorder(node.left)
183                 postorder(node.right)
184                 sys.stdout.write("%i -> " % node.data)
185             postorder(node)
186             print("None")
```

```
1 #BTNode_main.py
2 from BTNode import *
3 akar=8      # ada 8 elemen akan di-insert
4 def main():
5     root=BTNode(akar)
6     root.insert(3)
7     root.insert(10)
8     root.insert(1)
9     root.insert(6)
10    root.insert(4)
11    root.insert(7)
12    root.insert(14)
13    root.insert(13)
14    print('binary-tree root is ', akar)
15    root.disp_tree()
16    print('inorder: ')
17    root.in_order()
18    print()
19    print('preorder: ')
20    root.pre_order()
21    print()
22    print('postorder: ')
23    root.post_order()
24    print()
```

```

25 buang=3      #buang elemen bernilai 3
26 root.delete(buang)
27 print('setelah delete ',buang)
28 root.disp_tree()
29 print()
30 print('inorder: ')
31 root.in_order()
32 print()
33 print('preorder: ')
34 root.pre_order()
35 print()
36 print('postorder: ')
37 root.post_order()
38
39 main()

```



Hasil test BTNode

```
===== RESTART: D:\USER\Python\BTNode_main.py =
```

```
binary-tree root is 8
```

```
.....14
```

```
.....13
```

```
.....10
```

```
8
```

```
.....7
```

```
.....6
```

```
.....4
```

```
.....3
```

```
.....1
```

```
inorder:
```

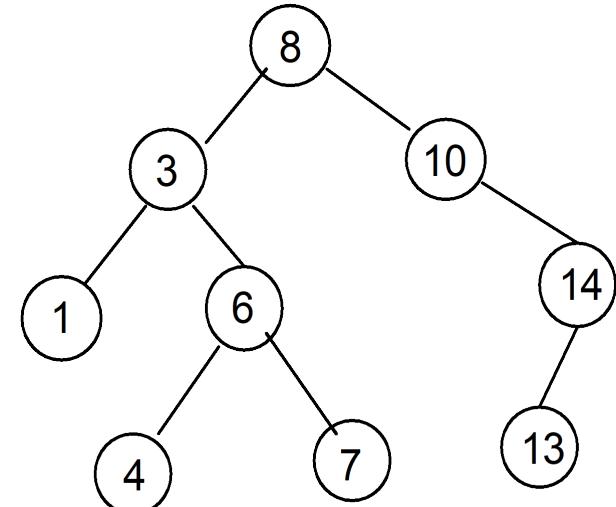
```
1 -> 3 -> 4 -> 6 -> 7 -> 8 -> 10 -> 13 -> 14 -> None
```

```
preorder:
```

```
8 -> 3 -> 1 -> 6 -> 4 -> 7 -> 10 -> 14 -> 13 -> None
```

```
postorder:
```

```
1 -> 4 -> 7 -> 6 -> 3 -> 13 -> 14 -> 10 -> 8 -> None
```



```
setelah delete 3
.....14
.....13
....10
8
.....7
.....6
....4
.....1
```

inorder:

1 -> 4 -> 6 -> 7 -> 8 -> 10 -> 13 -> 14 -> None

preorder:

8 -> 4 -> 1 -> 6 -> 7 -> 10 -> 14 -> 13 -> None

postorder:

1 -> 7 -> 6 -> 4 -> 13 -> 14 -> 10 -> 8 -> None

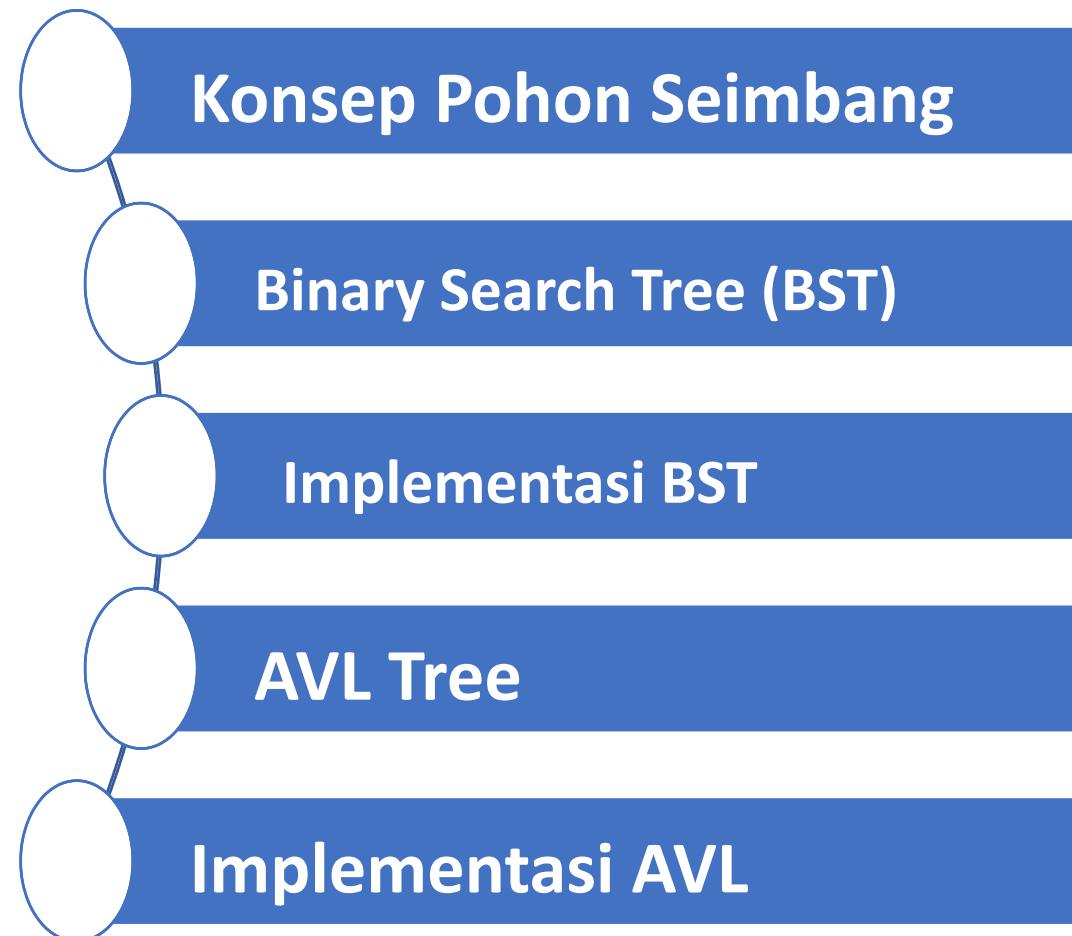


STRUKTUR DATA (PYTHON)

“Struktur BST dan Pohon Seimbang AVL”

[@SUARGA | [Pertemuan 14]]

OutLine





Konsep Pohon Seimbang

- Pohon Seimbang (Balanced Tree) adalah pohon yang tinggi-nya pada sisi kiri dan sisi kanan hampir sama.
- Apabila tidak diatur keseimbangan-nya maka mungkin terjadi suatu pohon dimana semua node ada di sisi kiri sementara sisi kanan-nya kosong, atau sebaliknya.
- Ada 2 contoh struktur pohon yang akan dibahas pada materi hari ini, yaitu:
 1. Binary Search Tree (BST) : memudahkan pencarian
 2. AVL Tree : sangat seimbang

Konsep Binary Search Tree

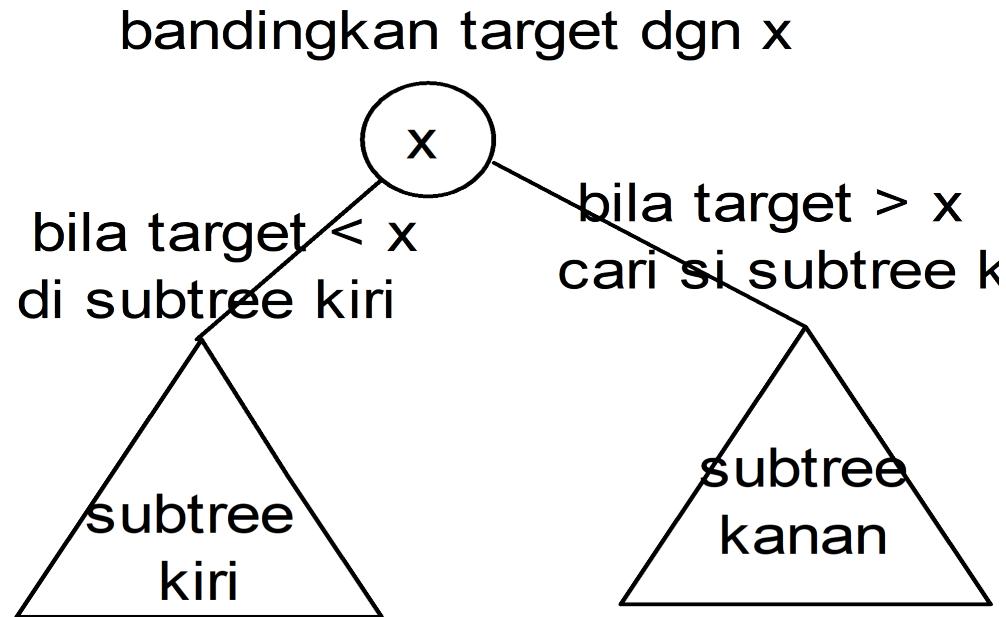
- Suatu Binary Search Tree atau BST adalah pohon biner dimana pada setiap node termuat suatu **kunci** (key), dan juga **data** (value) dan disusun mengikuti aturan pada pohon biner, agar suatu node mudah dicari, yaitu:
 1. Semua kunci yang lebih kecil dari kunci node-V akan disimpan disisi kiri dari node V.
 2. Semua kunci yang lebih besar atau sama dari kunci node-V akan disimpan di sisi-kanan dari node V
 3. Akan diatur agar kunci pada root memiliki nilai yang berada disekitar titik tengah vektor nilai.

BST Class

- Node dari BST bisa di-bangun dalam Python sebagai berikut:

```
class BST:  
    def __init__( self ):  
        self._root = None  
        self._size = 0  
    def __len__( self ):  
        return self._size  
  
class _BSTNode:  
    def __init__(self, key, value):  
        self.key = key  
        self.value = value  
        self.left = None  
        self.right = None
```

- Pada dasarnya akses ke BST tidak jauh berbeda dari pohon biner, misalnya dalam proses pencarian kunci, maka nilai kunci yang dicari dibandingkan dengan nilai x pada suatu node. Apabila kunci itu lebih kecil dari x maka cari di subtree sebelah kiri dan bila lebih besar maka cari di subtree sebelah kanan.



fungsi _bstSearch()

- Mencari suatu kunci target dalam BST
- Kode fungsi-nya bisa seperti berikut:

```
def _bstSearch( self, subtree, target ):  
    if subtree is None:  
        return None  
    elif target < subtree.key :  
        return self._bstSearch(subtree.left )  
    elif target > subtree.key :  
        return self._bstSearch(subtree.right )  
    else :  
        return subtree
```

- Berdasarkan pada fungsi `_bstSearch()` dapat dibuat dua fungsi baru yaitu fungsi `_contains_()` untuk memeriksa apakah suatu kunci ada dalam BST, dan juga fungsi `valueOf()` untuk mencari nilai data (value) yang berhubungan dengan suatu kunci (key).

```
def __contains__( self, key ):  
    return self._bstSearch(self._root, key )  
  
def valueof( self, key ):  
    node = self._bstSearch( self._root, key )  
    assert node is not None, "Invalid key... "  
    return node.value
```

- Mencari nilai key yang minimum maupun maksimum sering pula menjadi hal yang penting dalam BST. Nilai minimum diharapkan berada pada sisi-kiri dari root BST, karena bila root memuat key minimum maka pasti node lainnya semua akan berada di sisi kanan, dan sisi-kiri menjadi kosong. Demikian pula dengan nilai key maksimum, diharapkan berada di-sisi kanan dari root. Apabila berada di-posisi root, maka node lainnya semua berada di sisi-kiri.

mencari maximum dan minimum

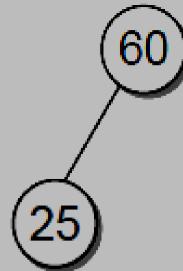
```
def _bstMinimum( self, subtree ):  
    if subtree is None:  
        return None  
    elif subtree.left is None:  
        return subtree  
    else :  
        return self._bstMinimum( subtree.left )  
  
def _bstMaximum( self, subtree ):  
    if subtree is None:  
        return None  
    elif subtree.right is None:  
        return subtree  
    else :  
        return self._bstMaximum( subtree.right )
```

menyisipkan node: insert()

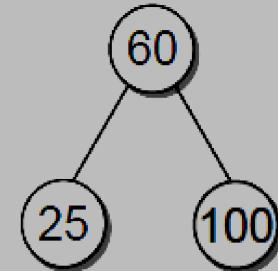
- Memasukkan suatu node ke dalam BST adalah dengan mengikuti aturan yang telah ditetapkan sebelumnya. Node pertama menjadi root selanjutnya bila key lebih kecil dari root maka tempatkan di sisi-kiri dan sebaliknya di sisi-kanan.



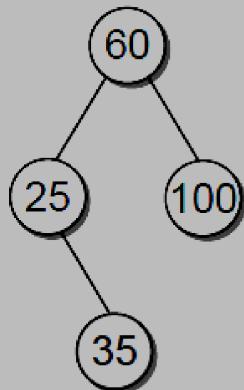
(a) Insert 60.



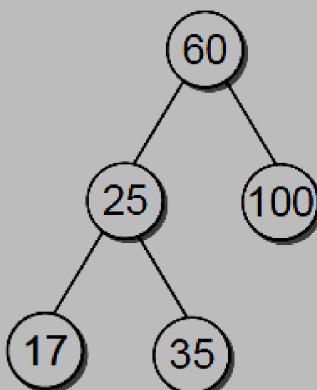
(b) Insert 25.



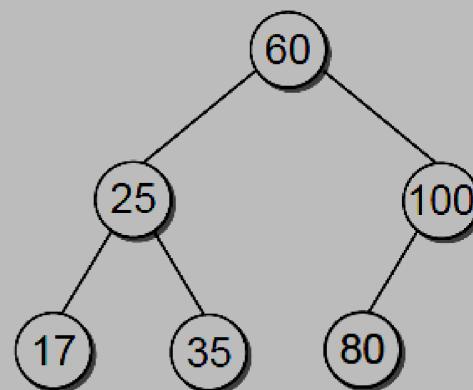
(c) Insert 100.



(d) Insert 35.



(e) Insert 17.



(f) Insert 80.

- Kode untuk menyisipkan atau memasukkan nilai baru ke dalam node adalah sebagai berikut.

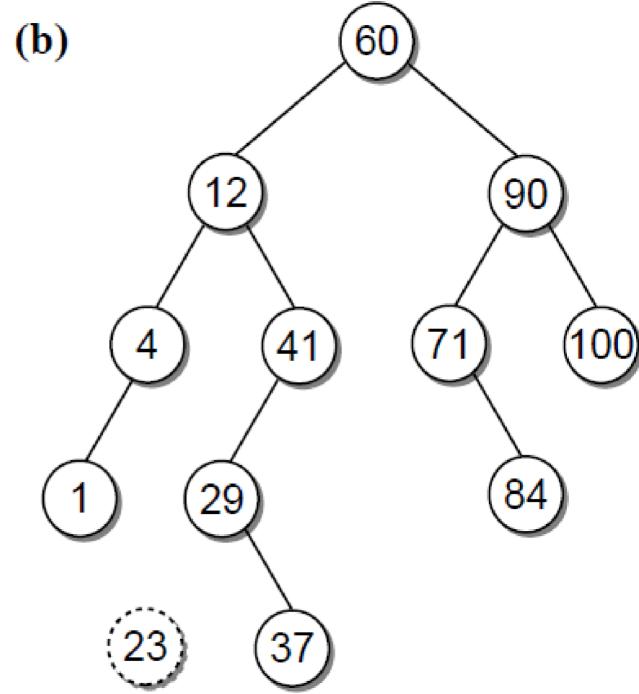
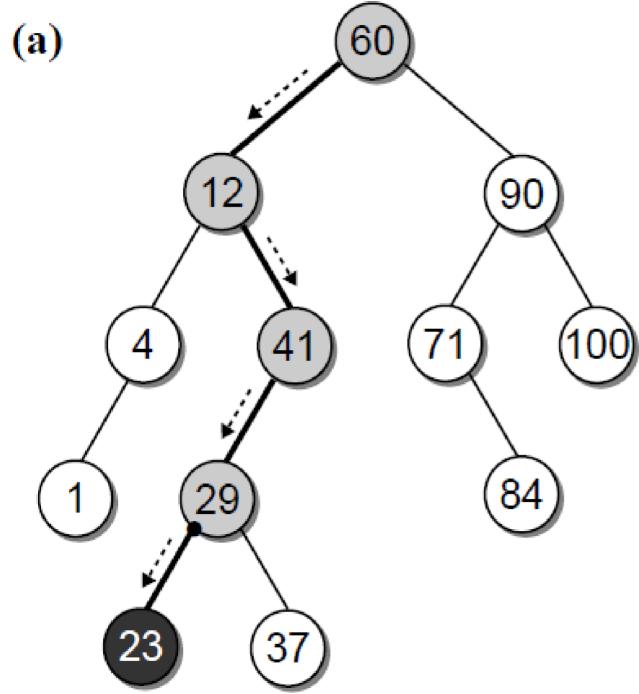
```
def _bstInsert( self, subtree, key, value ) :  
    if subtree is None :  
        subtree = _BSTNode( key, value )  
    elif key < subtree.key :  
        subtree.left = self._bstInsert(  
            subtree.left, key, value )  
    elif key > subtree.key :  
        subtree.right = self._bstInsert(  
            subtree.right, key, value )  
    return subtree
```

- Apabila key sudah ada tetapi data value-nya mau diganti maka kode-nya sebagai berikut:

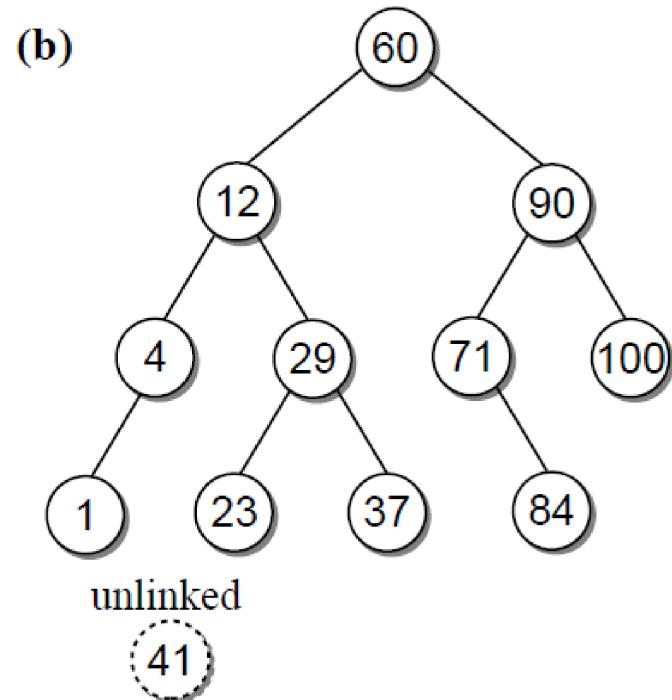
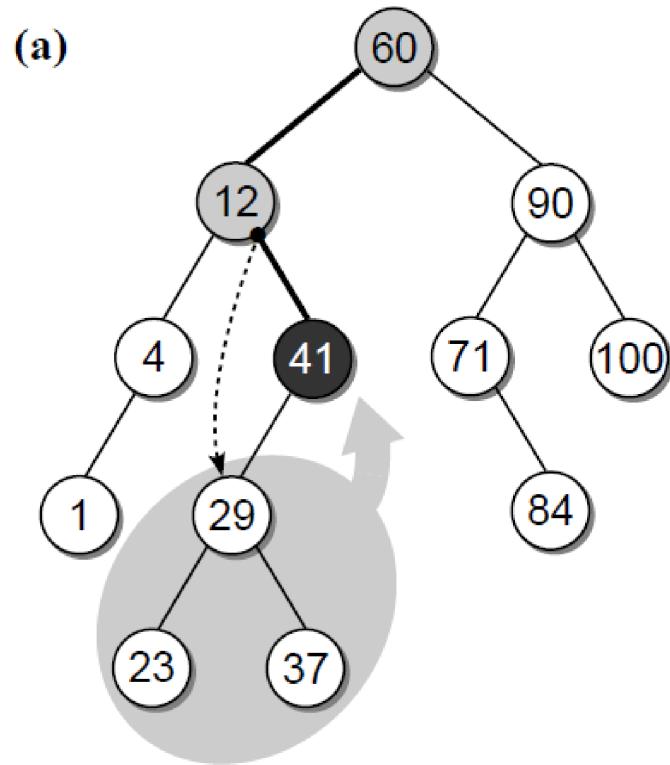
```
def add( self, key, value ):  
    node = self._bstSearch( key )  
    # apa kunci sdh ada  
    if node is not None:  
        # bila sdh ada  
        node.value = value  
        # ganti value-nya  
        return False  
    else:                      # kunci tidak ada  
        self._root = self._bstInsert( self._root, key,  
                                      value )  
        self._size += 1      # masukkan node-nya  
    return True
```

fungsi remove()

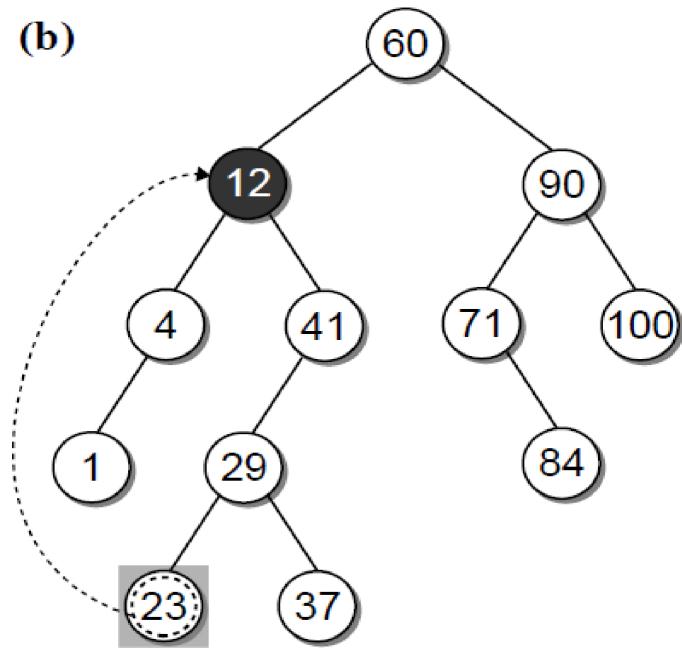
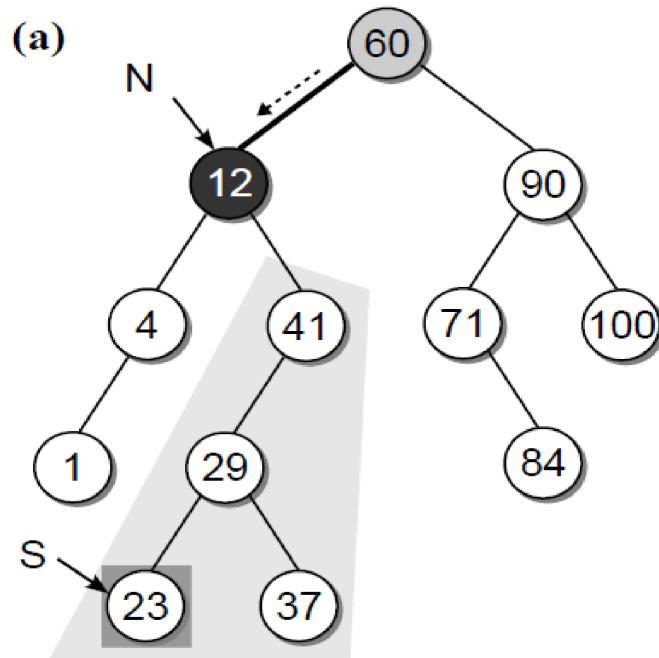
- Hal yang paling sulit dilakukan pada BST adalah menghapus suatu node (remove) berdasarkan suatu kunci. Ada tiga kemungkinan yang terjadi, yaitu:
 1. Node ini tidak memiliki anak, maka bisa langsung dihapus
 2. Node ini memiliki satu anak, maka node ini dihapus kemudian anak-nya ditarik menggantikannya
 3. Node ini memiliki dua anak, maka anak yang berada di-ujung kanan dari subtree kiri diangkat menggantikan node yang dihapus, kalau tidak ada, maka anak yang berada paling kiri dari subtree kanan-nya dipilih sebagai pengganti, kemudian anak sisi-kanannya dipasang dibawahnya.



Kasus 1, node 23 tidak memiliki anak
node 23 bisa langsung dihapus

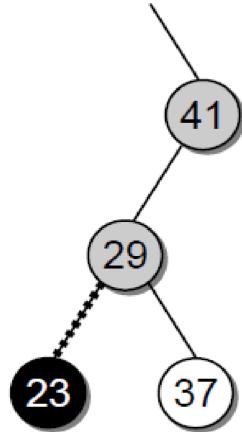


Kasus 2, node 41 memiliki satu anak
node 29 dapat diangkat menggantikan node 41

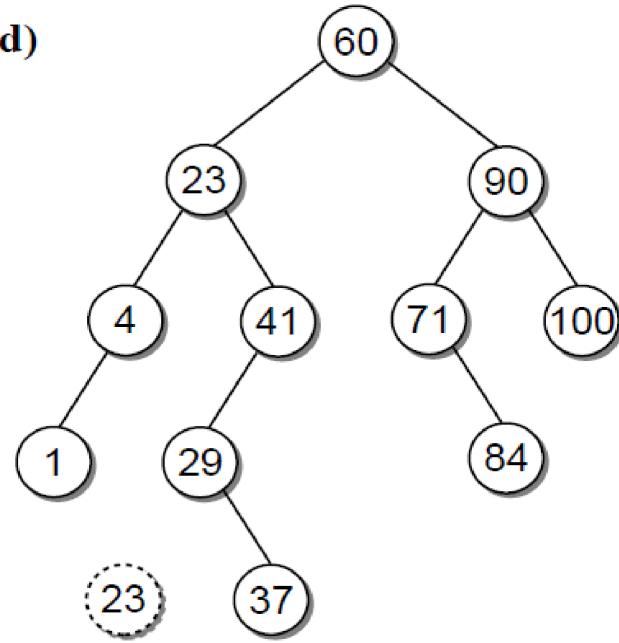


Kasus 3, node 12 memiliki 2 anak
cari node paling kanan dari turunan sisi-kiri
atau node paling kiri dari turunan sisi-kanan

(c)



(d)



Karena tidak ada node paling kanan dari sisi-kiri node 12
maka pilih node paling kiri dari sisi-kanan node 12, yaitu
node 23 untuk menggantikan node 12

- Kode fungsi remove() dapat ditulis sebagai berikut.

```
def remove( self, key ):  
    assert key in self, "Invalid key."  
    self._root = self._bstRemove( self._root, key )  
    self._size -= 1  
  
    # metoda yang melakukan ‘remove’ dari item recursif.  
def _bstRemove( self, subtree, target ):  
    # mencari item dalam tree.  
    if subtree is None :  
        return subtree  
    elif target < subtree.key :  
        subtree.left = self._bstRemove( subtree.left, target )  
    return subtree
```

```
elif target > subtree.key :  
    subtree.right = self._bstRemove( subtree.right, target )  
    return subtree  
# disini node yang memuat item ditemukan.  
else :  
    if subtree.left is None and subtree.right is None :  
        return None  
    elif subtree.left is None or subtree.right is None :  
        if subtree.left is not None :  
            return subtree.left  
        else :  
            return subtree.right  
    else :  
        successor = self._bstMinimum( subtree.right )  
        subtree.key = successor.key  
        subtree.value = successor.value  
        subtree.right = self._bstRemove( subtree.right,  
                                         successor.key )  
    return subtree
```

Implementasi BST

```
1 #my_BST.py == implementasi BST @Suarga
2 #definisi class untuk node BST
3 class _BSTNode:
4     def __init__(self, key, value):
5         self.key = key
6         self.value = value
7         self.left = None
8         self.right = None
9
10 #definisi| pohon BST
11 class BST:
12     #memulai BST
13     def __init__( self ):
14         self._root = None
15         self._size = 0
16
17     #menghitung banyaknya elemen/node
18     def __len__( self ):
19         return self._size
20
```

```
21 #mencari sebuah node target
22 def _bstSearch( self, subtree, target ):
23     if subtree is None:
24         return None
25     elif target < subtree.key :
26         return self._bstSearch( subtree.left, target )
27     elif target > subtree.key :
28         return self._bstSearch( subtree.right, target )
29     else :
30         return subtree
31
32 #memeriksa apakah key ada dalam BST
33 def __contains__( self, key ):
34     return self._bstSearch(self._root, key )
35
36 #menampilkan nilai yang berkaitan dengan kunci key
37 def valueOf( self, key ):
38     node = self._bstSearch( self._root, key )
39     assert node is not None, "Invalid key... "
40     return node.value
41
```

```
42 #menampilkan key minimum
43 def _bstMinimum( self, subtree ):
44     if subtree is None:
45         return None
46     elif subtree.left is None:
47         return subtree
48     else :
49         return self._bstMinimum( subtree.left )
50
51 #menampilkan key maksimum
52 def _bstMaximum( self, subtree ):
53     if subtree is None:
54         return None
55     elif subtree.right is None:
56         return subtree
57     else :
58         return self._bstMaximum( subtree.right )
59
60 #ADT tambahan ketika menyisipkan satu Node ke BST
61 def _bstInsert( self, subtree, key, value ):
62     if subtree is None :
63         subtree = _BSTNode( key, value )
64     elif key < subtree.key :
65         subtree.left = self._bstInsert( subtree.left, key, value )
66     elif key > subtree.key :
67         subtree.right = self._bstInsert( subtree.right, key, value )
68     return subtree
```

```
69
70     #ADT untuk memasukkan sebuah Node ke dalam BST
71     def add( self, key, value ):
72         node = self._bstSearch( self._root, key ) # apa kunci sdh ada
73         if node is not None:                  # bila sdh ada
74             node.value = value              # ganti value-nya
75             return False
76         else:                           # kunci tidak ada
77             self._root = self._bstInsert( self._root, key, value )
78             self._size += 1                 # masukkan node-nya
79             return True
80
81     #menghapus sebuah Node
82     def remove( self, key ):
83         assert key in self, "Invalid key."
84         self._root = self._bstRemove( self._root, key )
85         self._size -= 1
86
```

```
87 # ADT tambahan ketika menghapus sebuah key.
88 def _bstRemove( self, subtree, target ):
89     # Search for the item in the tree.
90     if subtree is None :
91         return subtree
92     elif target < subtree.key :
93         subtree.left = self._bstRemove( subtree.left, target )
94         return subtree
95     elif target > subtree.key :
96         subtree.right = self._bstRemove( subtree.right, target )
97         return subtree
98     # We found the node containing the item.
99     else :
100        if subtree.left is None and subtree.right is None :
101            return None
102        elif subtree.left is None or subtree.right is None :
103            if subtree.left is not None :
104                return subtree.left
105            else :
106                return subtree.right
107        else :
108            successor = self._bstMinimum( subtree.right )
109            subtree.key = successor.key
110            subtree.value = successor.value
111            subtree.right = self._bstRemove( subtree.right, successor.key )
112    return subtree
113
```

```
114 #menampilkan isi BST
115 def disp_tree(self, indent_char = '...', indent_delta=2):
116     node = self._root
117     def disp_tree_1(indent, node):
118         if node == None:
119             return None
120         else:
121             disp_tree_1(indent+indent_delta, node.right)
122             print(indent*indent_char+str(node.key)+ '|' + str(node.value))
123             disp_tree_1(indent+indent_delta, node.left)
124     disp_tree_1(0,node)
125
126 def disp_tree_rev(self, indent_char = '...', indent_delta=2):
127     node = self._root
128     def disp_tree_1(indent, node):
129         if node == None:
130             return None
131         else:
132             disp_tree_1(indent+indent_delta, node.left)
133             print(indent*indent_char+str(node.key)+ '|' + str(node.value))
134             disp_tree_1(indent+indent_delta, node.right)
135     disp_tree_1(0,node)
136
```

```
137 def main():
138     bt = BST()
139     bt.add(5, "Purple")
140     bt.add(3, "Black")
141     bt.add(7, "Red")
142     bt.add(2, "Yellow")
143     bt.add(9, "Green")
144     bt.disp_tree()
145     print('tambah beberapa node : ')
146     bt.add(8, "Blue")
147     bt.add(4, "White")
148     bt.disp_tree()
149     print('buang node key 4 : ')
150     bt.remove(4)
151     bt.disp_tree()
152     print('Cari key maksimum : ')
153     r = bt._root
154     node = bt._bstMaximum(r)
155     print(str(node.key) + ' | ' + str(node.value))
156     print('Cari key minimum : ')
157     node = bt._bstMinimum(r)
158     print(str(node.key) + ' | ' + str(node.value))
159
160 main()
```

Hasil Test

```
===== RESTART: D:/USER/
.....9|Green
.....7|Red
5|Purple
.....3|Black
.....2|Yellow
tambah beberapa node :
.....9|Green
.....8|Blue
.....7|Red
5|Purple
.....4|White
.....3|Black
.....2|Yellow
buang node key 4 :
.....9|Green
.....8|Blue
.....7|Red
5|Purple
.....3|Black
.....2|Yellow
Cari key maksimum :
9|Green
Cari key minimum :
2|Yellow
```

Balanced Tree / AVL Tree

- Pohon AVL pada prinsipnya adalah pohon BST dengan aturan yang lebih ketat.
- Pada BST akses bisa lebih efisien apabila pohon ini seimbang di sisi-kiri dengan sisi-kanan, karena pada kondisi ini maka tinggi pohon adalah $(\log n)$, dimana n adalah jumlah node dalam BST.
- Namun bisa saja terjadi kasus yang sangat buruk (worst case), yaitu ketika akar memiliki kunci yang nilainya minimum atau nilainya maksimum, dalam kondisi ini maka tinggi pohon adalah (n) .

- Berdasarkan pada kasus wors-case ini maka tiga orang peneliti yaitu: G.M. Adel'son, Velskii, dan Y.M. Landis merancang satu pohon biner dimana keseimbangan selalu di-perhatikan setiap kali satu node baru dimasukkan.
- Pohon biner ini kemudian diberi nama pohon AVL yang diambil dari huruf pertama nama belakang penemunya. Suatu pohon biner disebut seimbang apabila tinggi (height) sisi-kiri dan tinggi sisi-kanan maksimum berbeda satu.

- Pada setiap node dalam AVL di-hubungkan dengan suatu indikator yang disebut “balance factor” atau faktor keseimbangan, yaitu:

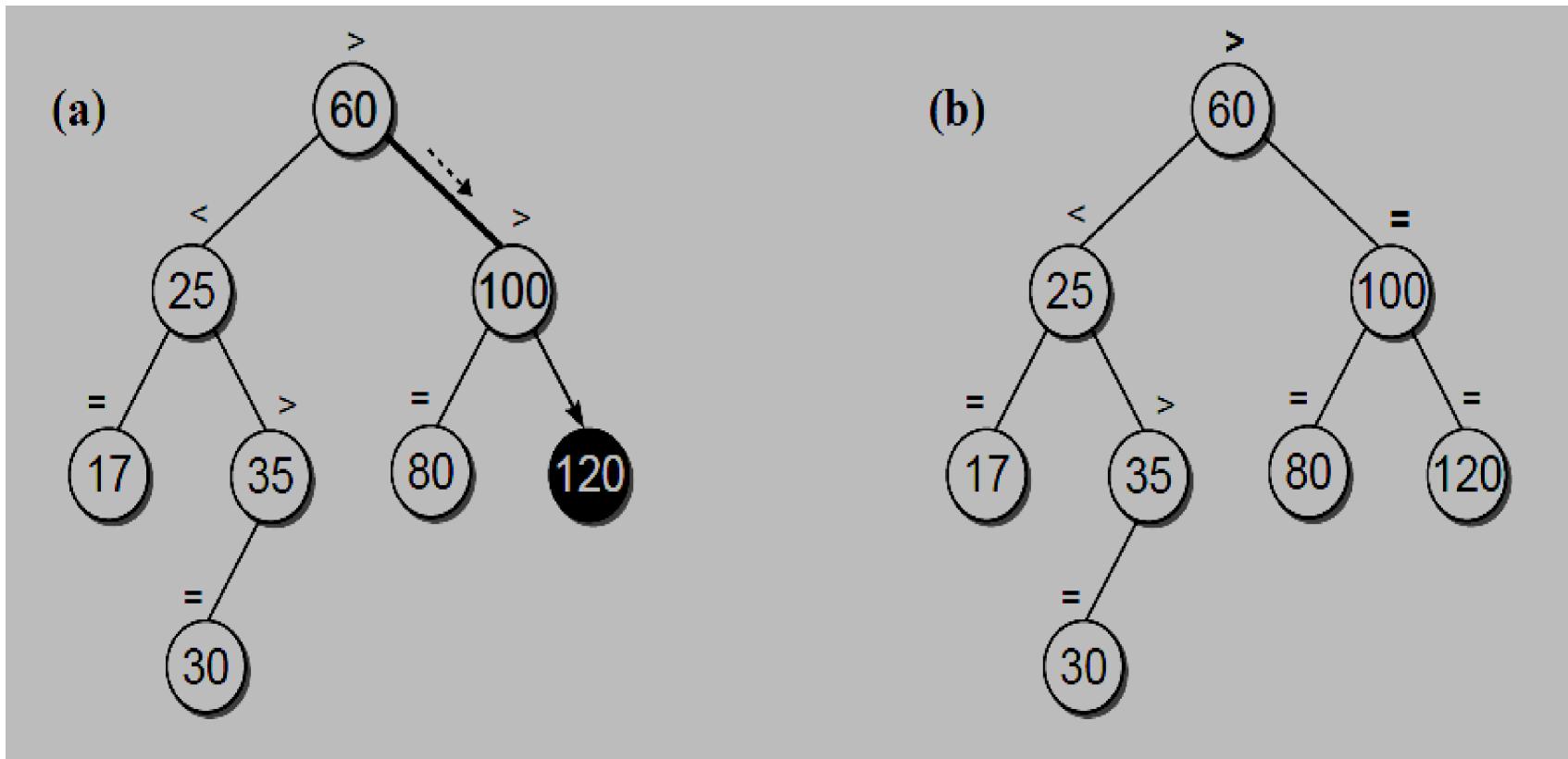
tanda > bila yang dikiri lebih tinggi

tanda = bila sama tinggi

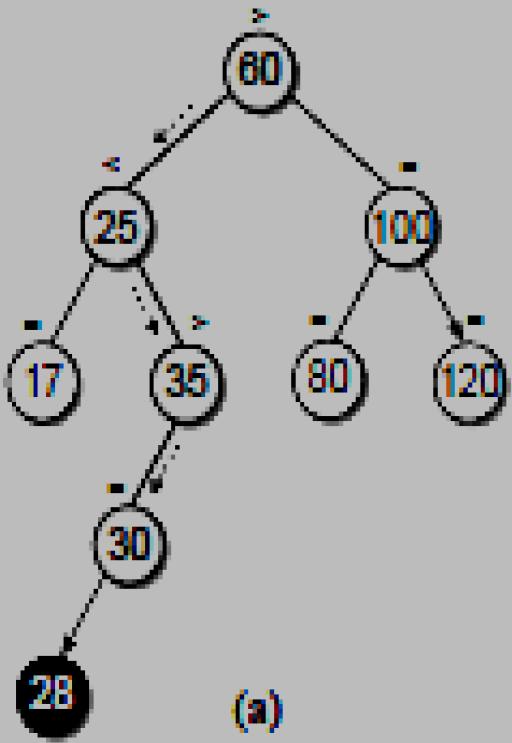
tanda < bila yang dikanan lebih tinggi

- Indikator ini bisa berubah ketika suatu node baru telah ditempatkan, sehingga susunan node dalam AVL bisa diubah menuju keseimbangan kiri dan kanan, agar bisa dijamin tinggi pohon maksimal adalah ($1.44 \log n$).

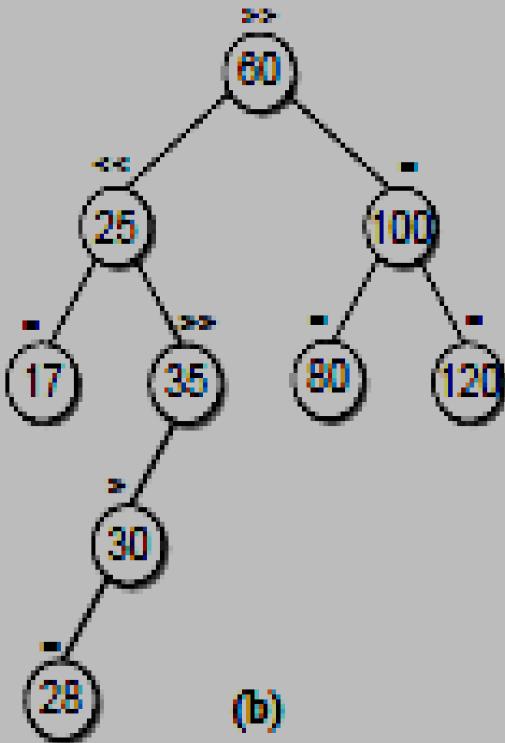
- Sebagai contoh andaikan node 120 akan ditambahkan ke dalam pohon AVL seperti pada gambar (a) berikut ini, maka sesuai dengan aturan pohon biner node 120 ditempatkan di-sisi-kanan node 100. Indikator keseimbangan berubah seperti pada gambar (b). Walau demikian antara sisi-kiri dan sisi-kanan masih tetap berselisih satu tinggi-nya, sehingga masih memenuhi syarat AVL.



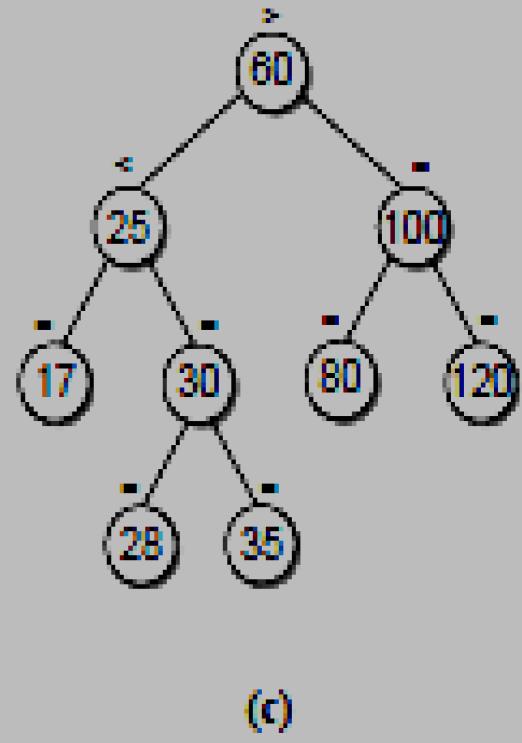
Bagaimana kalau node 28 ditambahkan, node ini ditempatkan di sisi-kiri node 30, seperti pada gambar berikut (a), maka antara sisi-kiri dan sisi-kanan tinggi-nya sudah berbeda dua (b), sehingga AVL ini perlu diseimbangkan lagi menjadi seperti gambar (c).



(a)



(b)



(c)

Implementasi pohon AVL

```
#####
#AVLTree.py  == @Suarga
import sys

class AVLNode:
    def __init__(self):
        self.data = 0
        self.ddata = ""
        self.left = None
        self.right = None

    def displayNode(self):
        print('{', self.data, ', ', self.ddata, '}')
```

Definisi node dari pohon AVL

```
- class AVLTree:
-     def __init__(self):
-         self.root = None
-
-     def avlinsert(self,data,ddata):
20        newItem = AVLNode()
21        newItem.data = data
22        newItem.ddata = ddata
23        if (self.root == None):
24            self.root = newItem
25        else:
26            self.root = self.RecursiveInsert(self.root, newItem)
```

Fungsi inisialisasi (`__init__()`) dan fungsi untuk menyisip sebuah node baru. (`avlinsert()`)

```
def RecursiveInsert(self, current, newDt):
    if (current == None):
        current = newDt
        return current
    elif (newDt.data < current.data): # go left
        current.left = self.RecursiveInsert(current.left, newDt)
        current = self.balance_tree(current) #balancing after insert
    elif (newDt.data > current.data): # go right
        current.right = self.RecursiveInsert(current.right, newDt)
        current = self.balance_tree(current)

    return current
```

fungsi menyisip node ke posisi-nya (di kiri atau di kanan)

```
40
def balance_tree(self, current):
    b_factor = self.balance_factor(current)
    if (b_factor > 1):
        if (self.balance_factor(current.left) > 0):
            current = self.RotateLL(current)
        else:
            current = self.RotateLR(current)
    elif (b_factor < -1):
        if (self.balance_factor(current.right) > 0):
            current = self.RotateRL(current)
        else:
            current = self.RotateRR(current)
    return current
50
```

fungsi untuk memeriksa keseimbangan AVL dengan menghitung “balance_factor”, bila tidak seimbang adakan suatu rotasi node:

- RotateLL : rotasi ke arah kiri
- RotateLR : rotasi ke kiri dulu lalu putar ke kanan
- RotateRL : rotasi ke kanan dulu lalu putar ke kiri
- Rotate RR: rotasi ke arah kanan

```
def getHeight(self, current):
    height = 0
    if (current != None):
        l = self.getHeight(current.left)
        r = self.getHeight(current.right)
        m = max(l, r)
        height = m + 1
    return height

60

def balance_factor(self, current):
    l = self.getHeight(current.left)
    r = self.getHeight(current.right)
    b_factor = l - r
    return b_factor
66
```

fungsi untuk menhitung tinggi pohon (getHeight()) dan fungsi yang menghitung faktor keseimbangan (balance_factor())

```
70 def RotateRR(self, parent):
    pivot = parent.right
    parent.right = pivot.left
    pivot.left = parent
    return pivot

80 def RotateLL(self, parent):
    pivot = parent.left
    parent.left = pivot.right
    pivot.right = parent
    return pivot

83 def RotateLR(self, parent):
    pivot = parent.left
    parent.left = self.RotateRR(pivot)
    return self.RotateLL(parent)

90 def RotateRL(self, parent):
    pivot = parent.right
    parent.right = self.RotateLL(pivot)
    return self.RotateRR(parent)
```

fungsi-fungsi rotasi untuk menyeimbangkan pohon AVL

```
def DisplayTree(self):
    self.InOrderDisplayTree(self.root)
    print()

def InOrderDisplayTree(self, current):
    if (current != None):
        self.InOrderDisplayTree(current.left)
        print('{',current.data,',',current.ddata,'} ')
        self.InOrderDisplayTree(current.right)

def find(self, key):
    ketemu = self.search(key)
    if (ketemu != None):
        print('Ditemukan: ')
    else:
        print(key, 'Tidak ditemukan')
```

Fungsi untuk menampilkan isi pohon (DisplayTree() + InOrderDisplayTree())
serta fungsi untuk menemukan satu kunci (find())

```
    def search(self, key):
        current = self.root
        while (current.data != key):
            if (key < current.data):
                current = current.left
            else:
                current = current.right
            if (current == None):
                return None
        current.displayNode()
        return current
```

Fungsi search() adalah ADT tambahan untuk pencarian key

```
def delete(self, current, target):
    if (current == None):
        return None
    else:
        if (target < current.data): # left subtree
            current.left = self.delete(current.left, target)
            if (self.balance_factor(current) == -2):
                if (self.balance_factor(current.left) <= 0):
                    current = self.RotateRR(current)
                else:
                    current = self.RotateRL(current)
        elif (target > current.data): # right subtree
            current.right = self.delete(current.right, target)
            if (self.balance_factor(current) == 2):
                if (self.balance_factor(current.right) <= 0):
                    current = self.RotateLL(current)
                else:
                    current = self.RotateLR(current)
        else: # target found
```

Fungsi dele te() untuk menghapus sebuah node, yang memanfaatkan fungsi rotasi RR, RL, LL, LR

```
140     else: # target found
141         if (current.right != None):
142             parent = current.right
143             while (parent.left != None):
144                 parent = parent.left
145             current.data = parent.data
146             current.right = self.delete(current.right, parent.data)
147             if (self.balance_factor(current) == 2):
148                 if (self.balance_factor(current.left) <= 0):
149                     current = self.RotateLL(current)
150                 else:
151                     current = self.RotateLR(current)
152             else:
153                 return current.left
154
155     return current
```

```
158     def Delete(self, target):
159         self.delete(self.root, target)
160
161     # display sideways
162     def disp_tree(self, indent_char = '...', indent_delta=2):
163         node = self.root
164         def disp_tree_1(indent, node):
165             if node == None:
166                 return None
167             else:
168                 disp_tree_1(indent+indent_delta, node.right)
169                 print(indent*indent_char+str(node.data))
170                 disp_tree_1(indent+indent_delta, node.left)
171
172         disp_tree_1(0, node)
173
174     # display normally
```

```
# display normally
def display_Tree(self):
    globalStack = []
    globalStack.append(self.root)
    nBlanks = 32
    isRowEmpty = False
    titik = '..'
    print(32*titik)
    while (isRowEmpty == False):
        localStack = []
        isRowEmpty = True
        spasi = ' '
        for j in range(nBlanks):
            sys.stdout.write("%s" % spasi)
        while (len(globalStack) != 0):
            temp = globalStack.pop()
            if (temp != None):
                sys.stdout.write("%d" % temp.data)
                localStack.append(temp.left)
                localStack.append(temp.right)
                if ((temp.left != None) or (temp.right != None)):
                    isRowEmpty = False
            else:
                isRowEmpty = True
    print('')

180
190
```

```
        else:
            sys.stdout.write('---')
            localStack.append(None)
            localStack.append(None)
            for j in range(nBlanks * 2 - 2):
                sys.stdout.write("%s" % spasi)
            print(' ')
            nBlanks = nBlanks//2
            while (len(localStack)!= 0):
                globalStack.append(localStack.pop())
            print(32*titik)

206
def main():
    theTree = AVLTree()
    print('Insert data : ')
210
    theTree.avlinsert(50, "Agus")
    theTree.avlinsert(25, "Beddu")
    theTree.avlinsert(75, "Chaidar")
    theTree.avlinsert(12, "Daud")
    theTree.avlinsert(37, "Erik")
    theTree.avlinsert(43, "Farid")
    theTree.avlinsert(30, "Gugun")
    theTree.avlinsert(33, "Harun")
    theTree.avlinsert(87, "Indah")
    theTree.avlinsert(93, "Jeny")
    theTree.avlinsert(97, "Kiki")
220
```

```
220     theTree.avlinsert(97, "Kiki")
        print('Insert selesai')
        print('Data dibaca inorder :')
        theTree.DisplayTree()
        print('Bentuk AVL : ')
        theTree.display_Tree()
        print('Menghapus node 43')
        theTree.Delete(43)
        print('AVL setelah 43 dihapus:')
        theTree.display_Tree()
        print('AVL tree : sideways')
        theTree.disp_tree()
        print('Mencari node 30')
        theTree.find(30)
        print('Mencari node 55')
        theTree.find(55)
        print('Menghapus node 87')
        theTree.Delete(87)
        print('AVL setelah 87 dihapus:')
        theTree.display_Tree()
        print('Insert node 56:')
        theTree.avlinsert(56, 'Lilik')
        print('AVL setelah 56 di-insert')
        theTree.display_Tree()
```

2-September-2022

Struktur Data-14, @Suarga

Hasil Test

```
Python Interpreter
*** Remote Interpreter Reinitialized ***
Insert data :
Insert selesai
Data dibaca inorder :
{ 12 , Daud }
{ 25 , Beddu }
{ 30 , Gugun }
{ 33 , Harun }
{ 37 , Erik }
{ 43 , Farid }
{ 50 , Agus }
{ 75 , Chaidar }
{ 87 , Indah }
{ 93 , Jeny }
{ 97 , Kiki }

Bentuk AVL :
.....
      37
    25           87
  12       30       50       93
--   --   --   33   43   75   --
.....
```


Python Interpreter

Ditemukan:

Mencari node 55

55 Tidak ditemukan

Menghapus node 87

AVL setelah 87 dihapus:

.....
37

25

93

12

30

50

97

--

--

--

33

--

75

--

--

Insert node 56:

AVL setelah 56 di-insert

.....
37

25

93

12

30

56

97

--

--

--

33

50

75

--

--

>>>