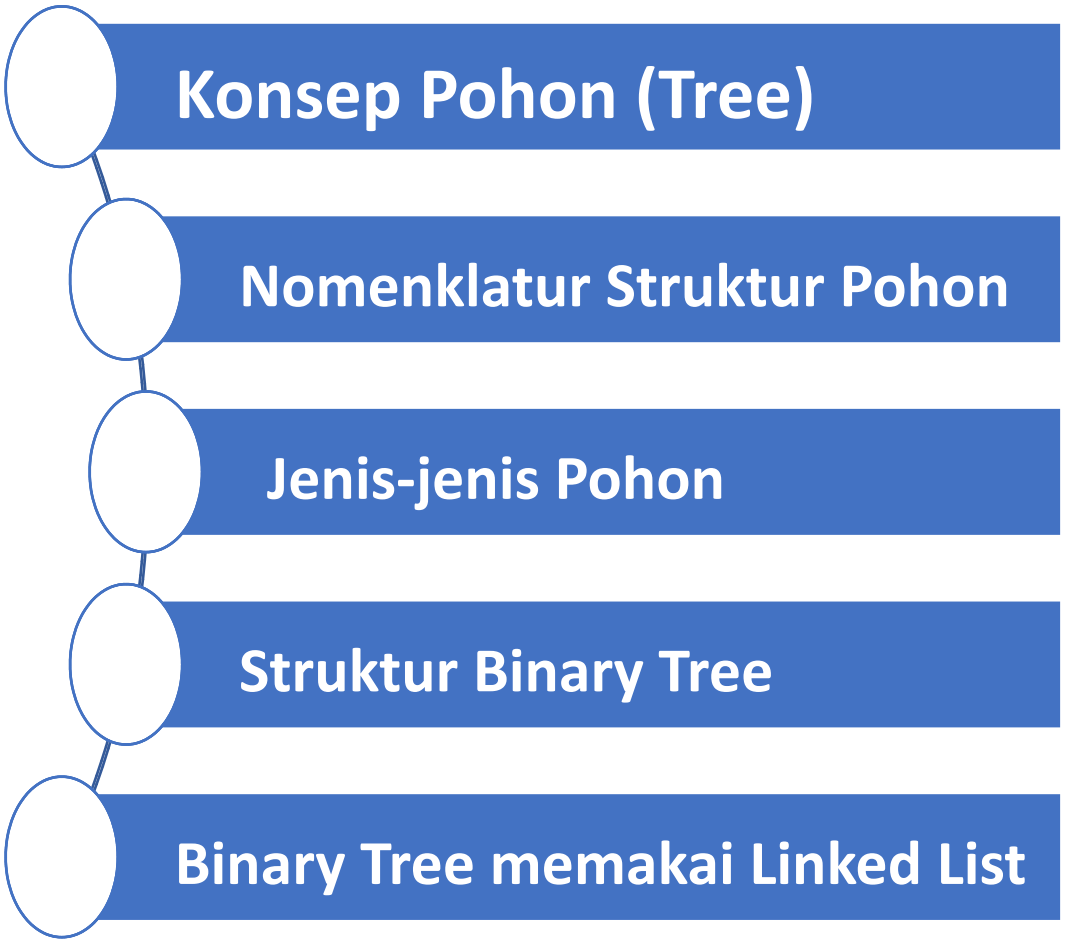
#KampusKuningGading

STRUKTUR DATA (PYTHON)

"Struktur Pohon Biner"

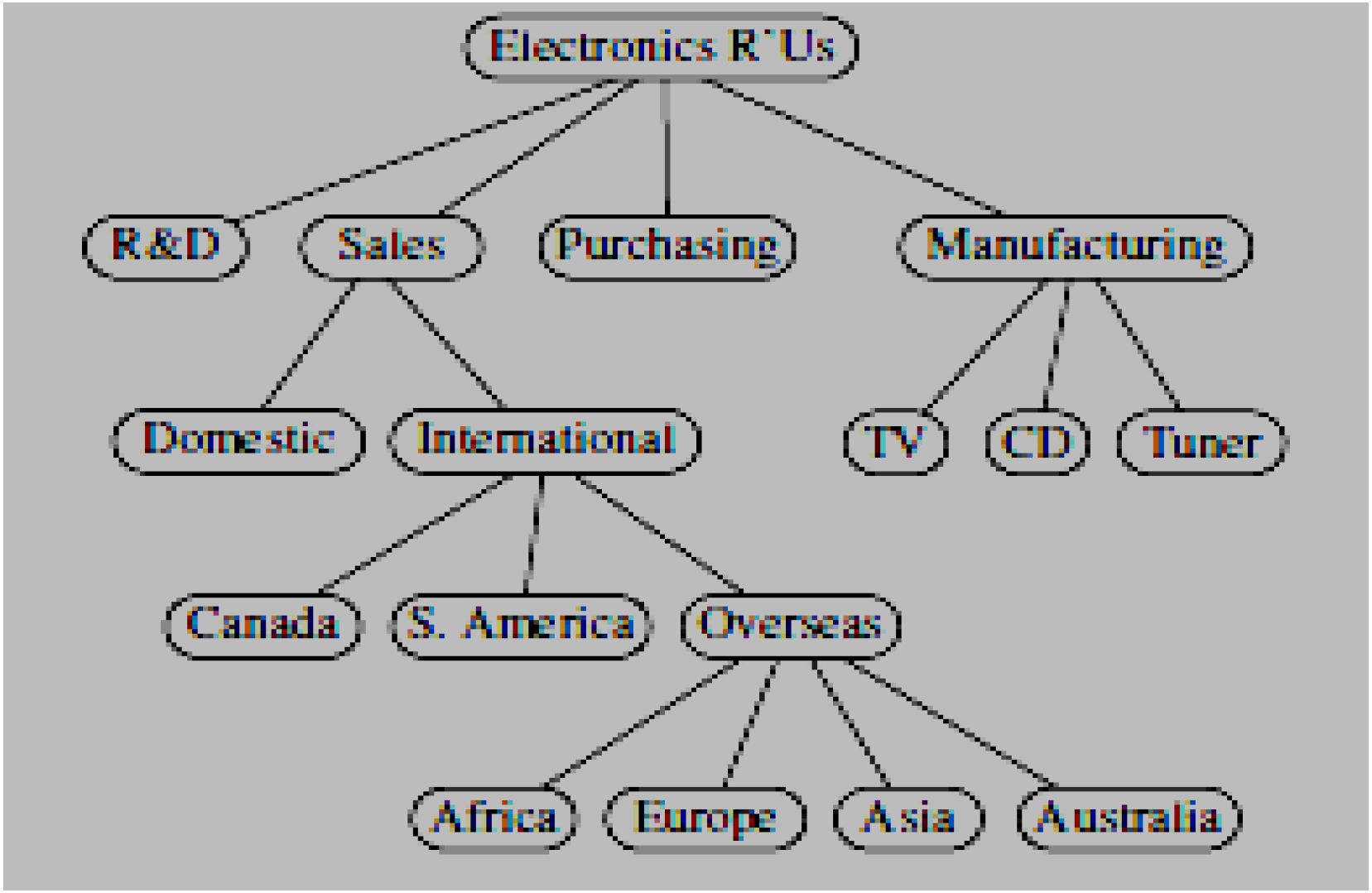
[@SUARGA| [Pertemuan 13]

OutLine


Konsep Pohon (Tree)
Nomenklatur Struktur Pohon
Jenis-jenis Pohon
Struktur Binary Tree
Binary Tree memakai Linked List

Konsep Struktur Pohon (Tree)

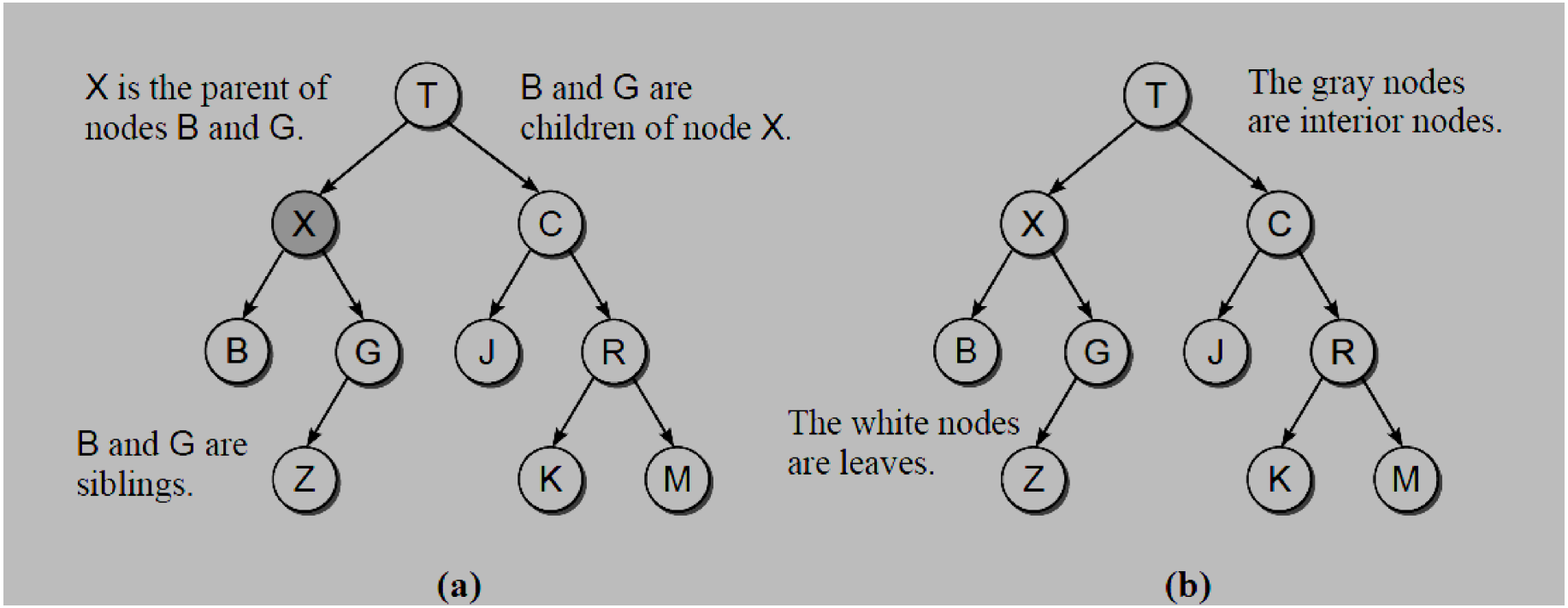
- Struktur pohon (tree) adalah abtraksi tipe data yang menyimpan elemen-nya secara hirarki, dari akar (root) hingga ke daun (leaves), namun secara grafis struktur pohon justru digambar sebagai pohon terbalik, dimana akar berada diatas dan daun berada dibawah.
- Struktur pohon digambarkan seperti hirarki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


Electronics R'Us
R&D
Sales
Purchasing
Manufacturing
Domestic
International
TV
CD
Tuner
Canada
S. America
Overseas
Africa
Europe
Asia
Australia

Nomeklatur Pohon

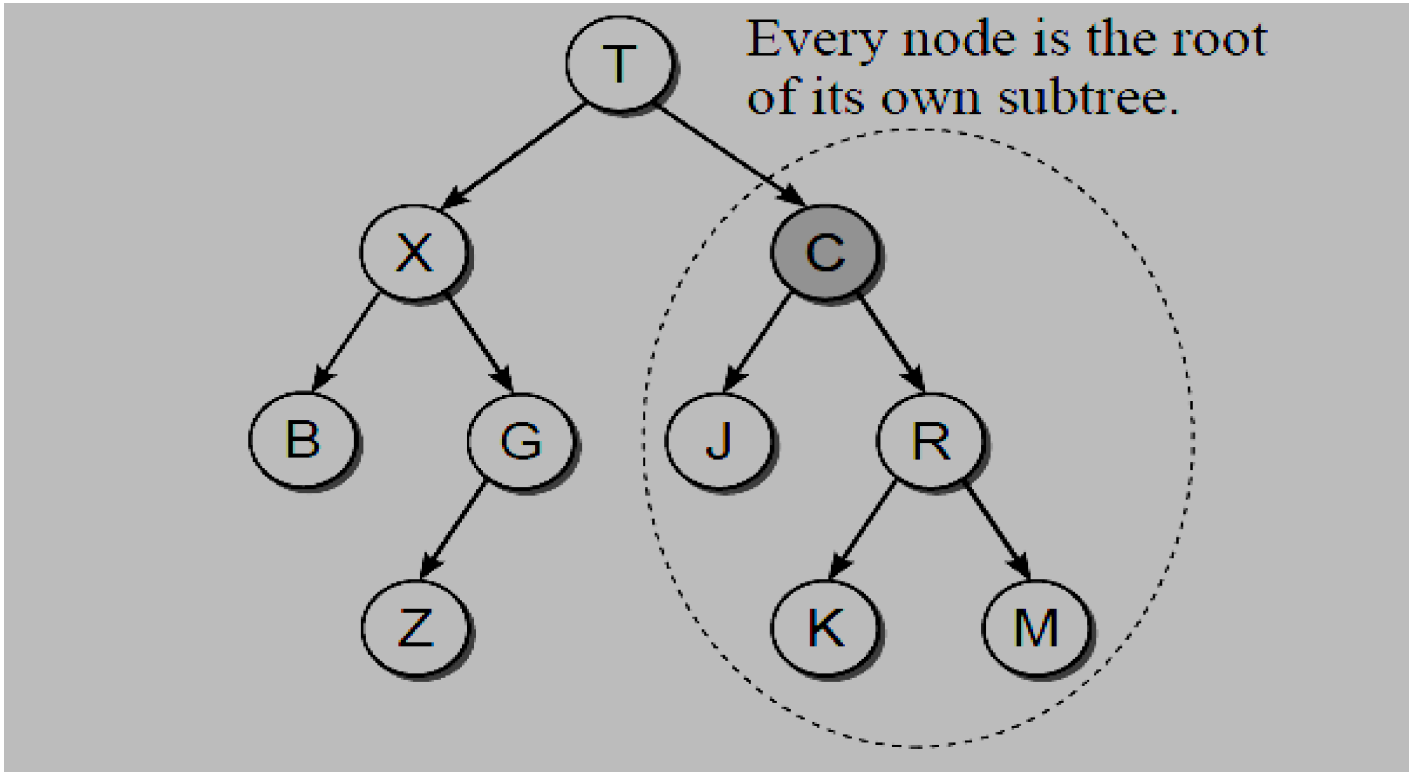
- **Root** : akar, adalah elemen (node) paling atas, merupakan pintu atau pointer untuk meng-akses elemen lain dalam pohon. Akar adalah satu-satunya node yang tidak memiliki ‘atasan’ atau parent. Ketika suatu pohon mulai dibentuk maka node pertama yang harus diciptakan adalah akar (root) ini.
- **Path** : 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).



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.

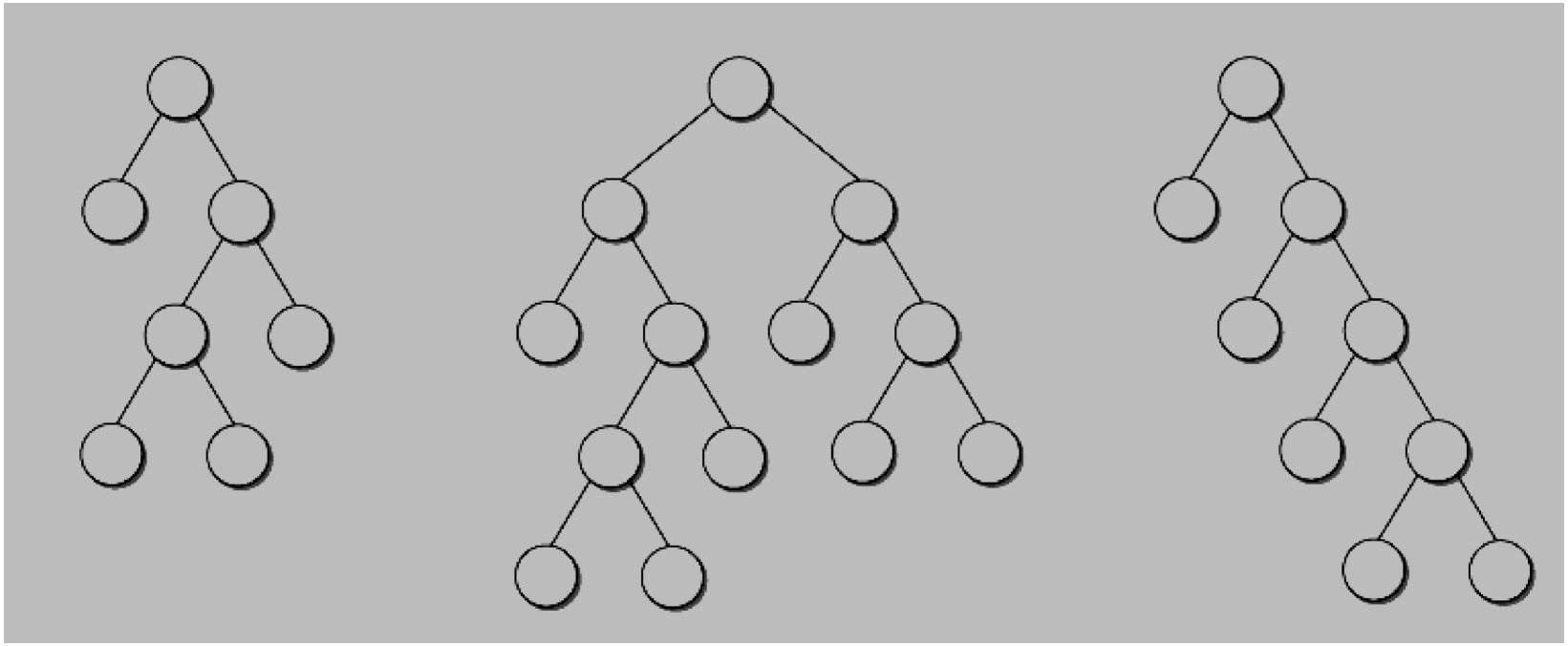
- **Red-Black Tree** : suatu pohon biner seimbang, dimana node-nya diberi warna merah atau hitam, mengikuti suatu aturan.

- **2-3- Tree** : suatu pohon seimbang yang memiliki aturan yang lebih ketat, setiap node boleh berisi dua kunci, dan maksimum tiga anak.

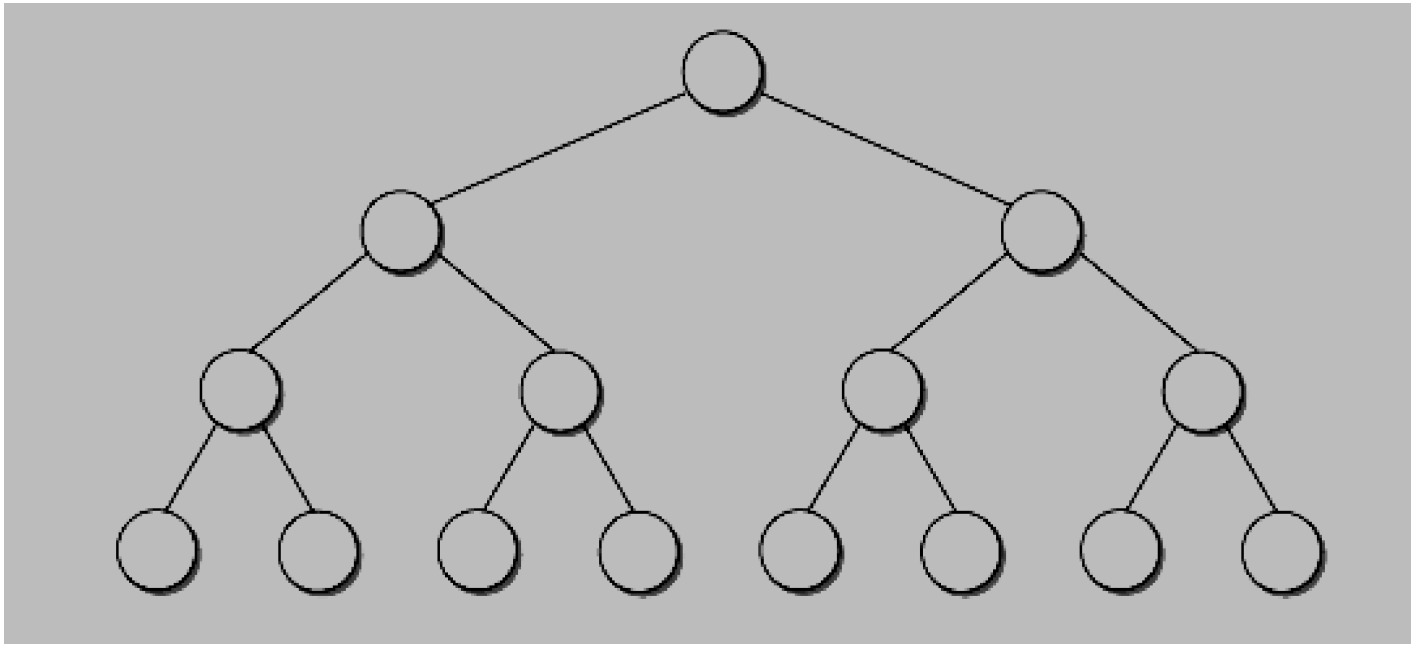
Binary-Tree

- 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:

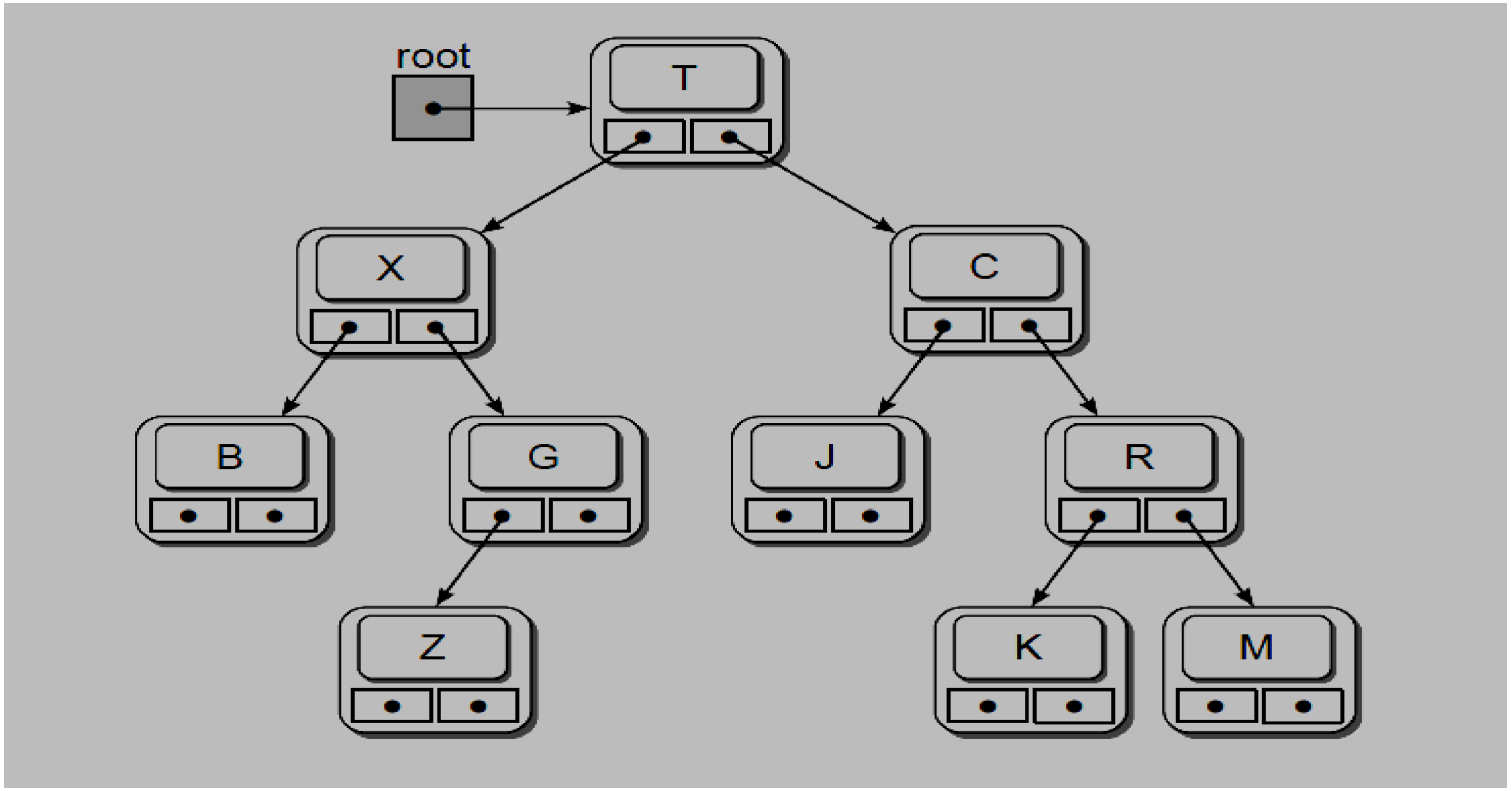
```
class _BinTreeNode :
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```



: **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:

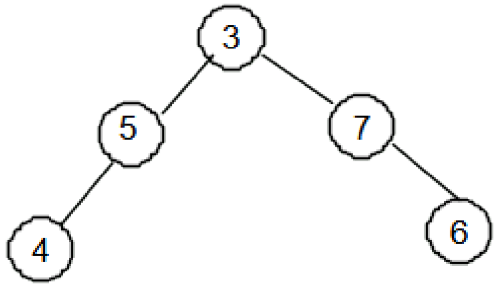
```
def getRightChild(root):
    return root[2]
```

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]
```

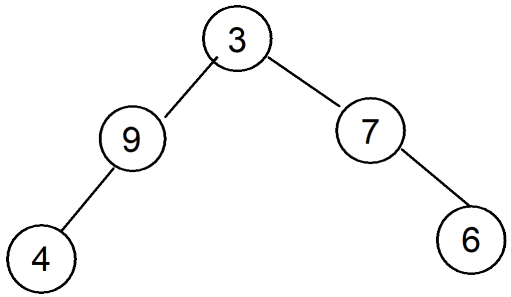
Contoh

```
>>> 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

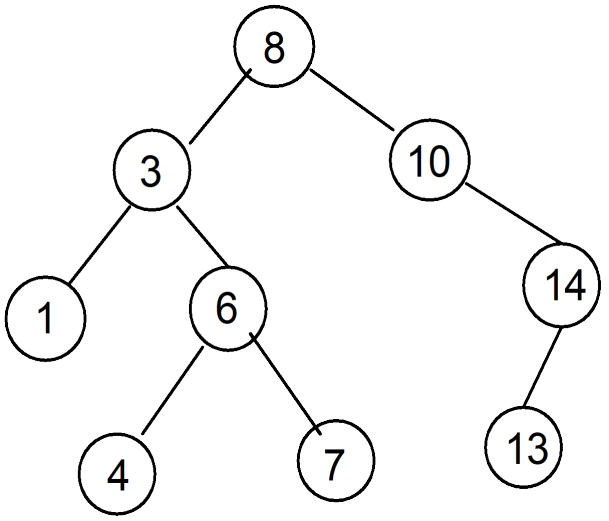
```
inorder(node)
   if left(node) <> null
       then inorder(left(node))
   visit(node)
   if right(node) <> null
      then inorder(right(node))
```

- ***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          if data < self.data:
57              if self.left is None:
58                  return None, None
59              return self.left.lookup(data, self)
60          elif data > self.data:
61              if self.right is None:
62                  return None, None
63              return self.right.lookup(data, self)
64          else:
65              return self, parent
66
```

```
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 (susccessor.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
```

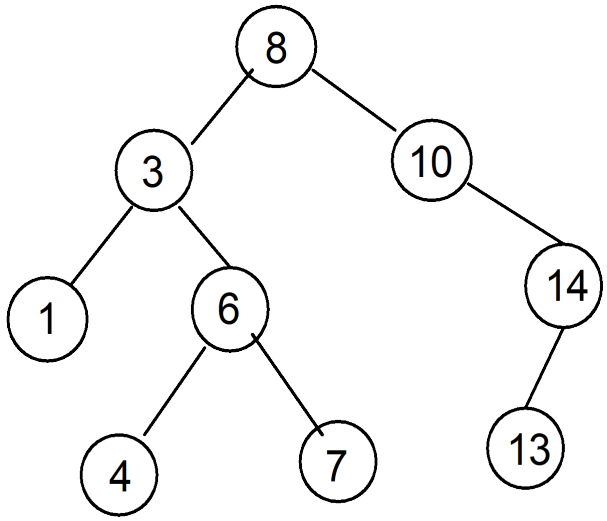
```
117     def print_tree(self):
118         """
119         Print tree content inorder
120         """
121         if self.left:
122             #print('left',)
123             self.left.print_tree()
124         print (self.data,)
125         if self.right:
126             #print('right',)
127             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")
```

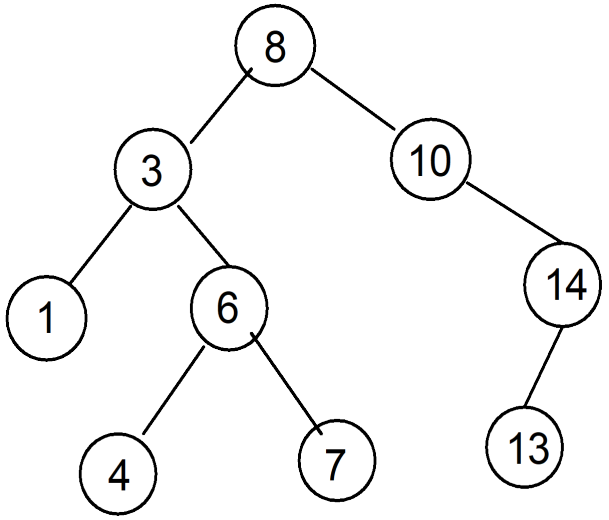
```python
#BTNode_main.py
from BTNode import *
akar=8      # ada 8 elemen akan di-insert
def main():
    root=BTNode(akar)
    root.insert(3)
    root.insert(10)
    root.insert(1)
    root.insert(6)
    root.insert(4)
    root.insert(7)
    root.insert(14)
    root.insert(13)
    print('binary-tree root is ', akar)
    root.disp_tree()
    print('inorder: ')
    root.in_order()
    print()
    print('preorder: ')
    root.pre_order()
    print()
    print('postorder: ')
    root.post_order()
    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
```