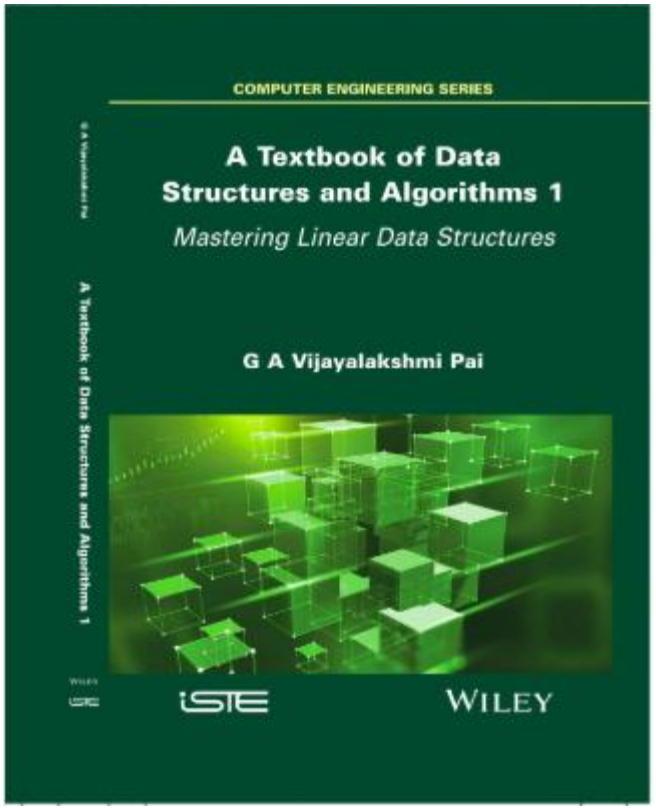
TIB29 – Struktur Data dan Algoritma

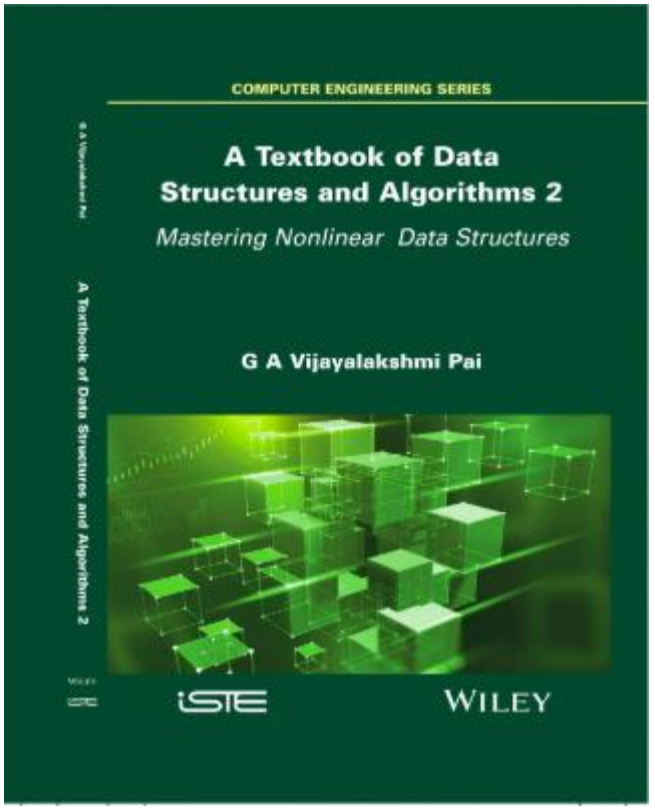
PERINGATAN HAK CIPTA

PERINGATAN HAK CIPTA

Tree dan *Binary Tree*

Diadopsi Dari Sumber:





Sub-CPMK

- Mahasiswa mampu membentuk binary tree dan melakukan penelusuran binary tree. (C3, A3)

Materi

1. Tree
2. Pengertian Binary Tree
3. Menambah Simpul
4. Binary Tree Traversal
5. Menemukan Induk Node
6. Menghapus simpul

1. *Tree*

1.1 Trees

- *Tree* adalah Struktur data yang diakses mulai dari simpul akar sampai ujung-ujung daun.
- *Tree* merupakan *graph* terhubung yang berurutan, tidak berputar dan tidak berarah.

1.1 Trees (Lanj.)

- Setiap simpul berupa daun atau simpul internal.
- Simpul internal memiliki satu atau lebih simpul anak dan disebut induk dari simpul anaknya.
- Semua anak dari simpul yang sama adalah saudara.
- *Tree* pada struktur data berlawanan dengan pohon secara fisik, akarnya biasanya digambarkan di bagian atas struktur, dan daunnya digambarkan di bagian bawah.

1.1 Trees (Lanj.)

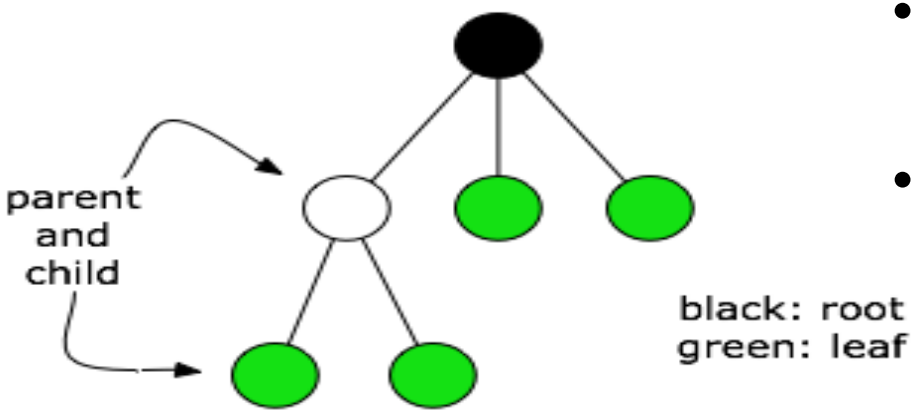


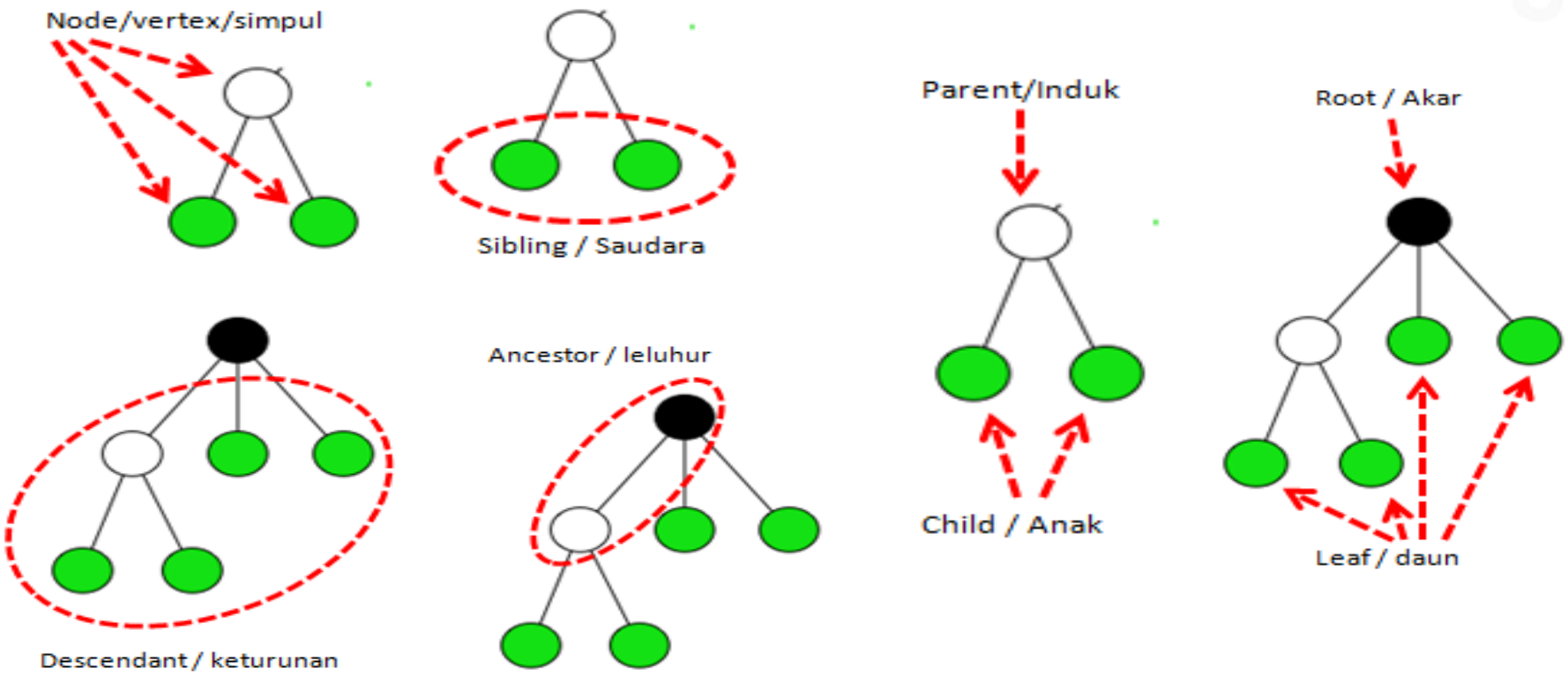
Figure: tree data structure

- Terdiri dari nodes/simpul dan panah
- Digambarkan terbalik dengan akar di bagian atas dan daun di bagian bawah

1.2 Terminologi

- Node / Vertex / Simpul
  - Referensi pada struktur data
  - Sekumpulan informasi yang berada pada sebuah lokasi memory
- *Parent*/Induk
- *Child*/Anak
- *Sibling*/Saudara
- *Descendant* / keturunan
- *Ancestor* / leluhur
- *Root* / Akar: node/simpul paling awal, hanya satu *item* dan tidak punya induk
- *Leaf* / Daun / Internal node: Node/simpul pada *tree* yang paling ujung / tidak punya anak

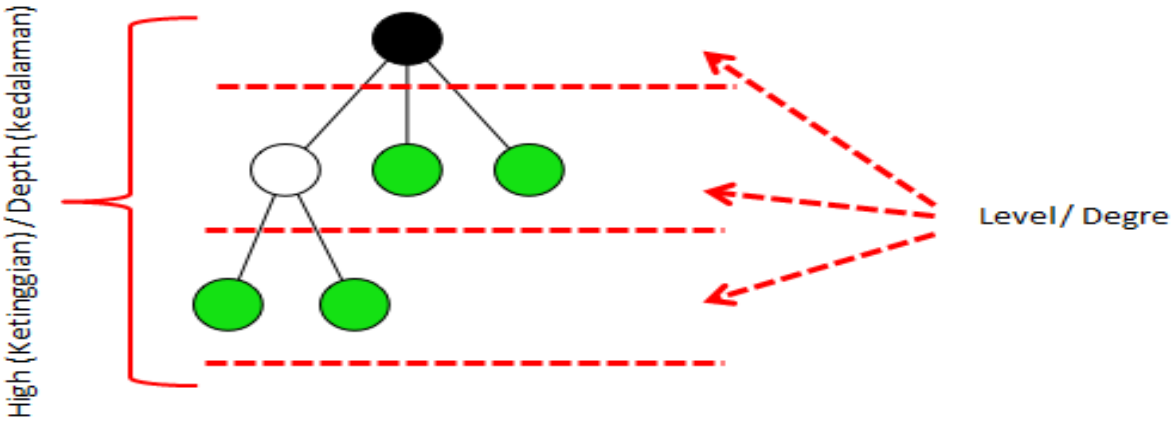
1.2 Terminologi (Lanj.)

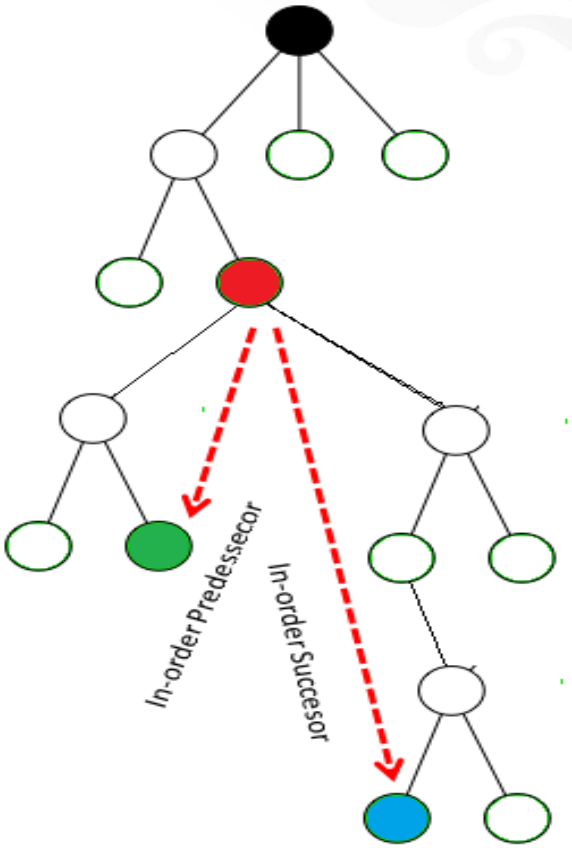


1.2 Terminologi (Lanj.)

- Level / *Degree*
- *Height* (ketinggian) / *depth* (kedalaman)
- Predesesor (*intermediate predecessor*) : Sel pendahulu, dalam *tree traversal* adalah yang dikunjungi terlebih dahulu
- Suksesor (*Successor*): sel berikut, dalam *tree traversal* adalah sel berikut yang akan dikunjungi

1.2 Terminologi (Lanj.)





1.3 Jenis *Tree*

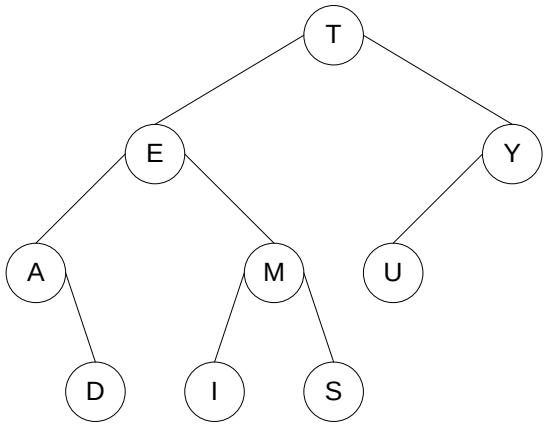
Dua jenis *tree*

- *Binary Tree*
- *Multiary Tree* (n-Ary *Tree)*

2. Pengertian *Binary Tree*

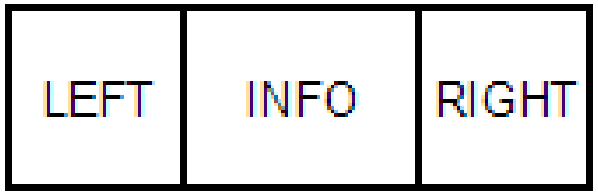
2.1 *Binary Tree*

- *Tree* dengan paling banyak dua anak pada tiap node nya disebut *binary Tree*.

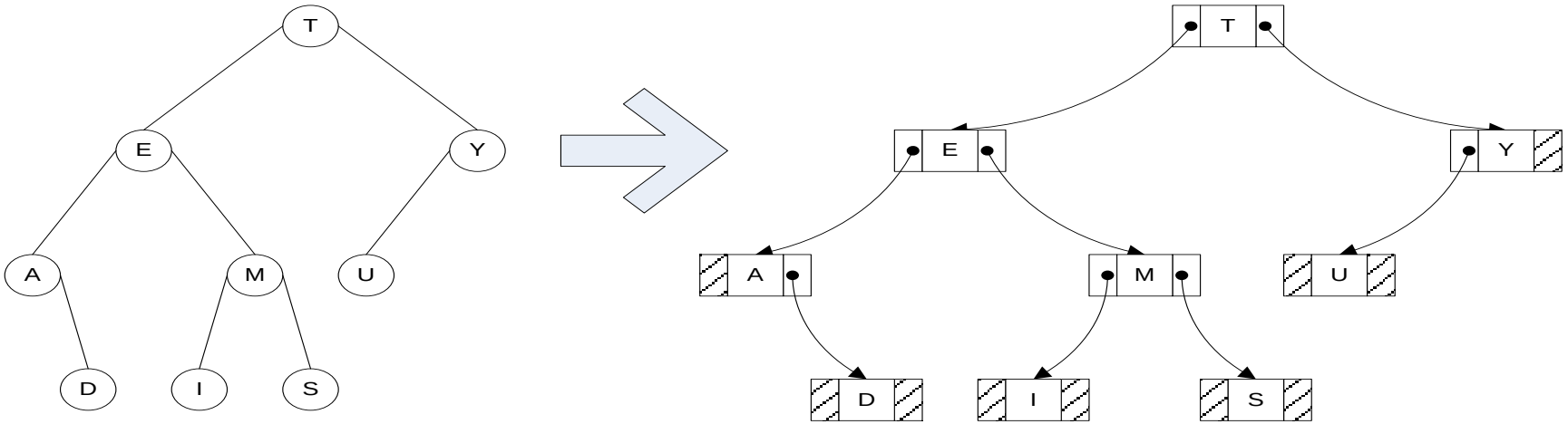


2.2 Implementasi *Binary Trees* Dengan *Linked List*

- *Element*
  - *Information*
  - *Left link → link to the child node at the left side*
  - *Right link → link to the child node at the right side*



2.2 Implementasi *Binary Trees* Dengan *Linked List (Lanj.)*



3. Menambah Simpul

3.1 Menambah Simpul

- Menempatkan sebuah *key* baru pada *binary tree*
- Menambah simpul ada dua orientasi yaitu *left-to-Right* dan *Right-to-Left*
- Untuk *Left-to-Right*: Setiap *key* pada kanan *subtree* harus lebih besar dari setiap *key* di kiri *subtree* (berlaku kebalikannya untuk *Right-to-Left*)
- *Rules*:
  - Jika *key* baru lebih besar dari node induk, arahkan ke node anak sebelah kanan
  - Jika *key* baru lebih kecil dari node induk, arahkan ke node anak sebelah kiri
  - Ulangi proses di atas sampai didapatkan node yang tidak memiliki *leaf*
  - Tempatkan *key* baru pada sebagai kanan atau kiri node berdasarkan dua kondisi pertama di atas

4. *Binary Tree Traversal*

4.1 *Tree Traversal*

- Proses mengunjungi setiap simpul pada *tree*/pohon biner tepat satu kali untuk setiap node
- Dapat juga ditafsirkan sebagai meletakkan semua node pada satu baris atau linearisasi *tree*/pohon
- Dapat dilakukan dari kiri ke kanan ataupun dari kanan ke kiri.
- Ada tiga metode *Traversal*:
  - *Pre-Order Traversal*
  - *In-Order Traversal*
  - *Post-Order Traversal*

4.2 Notasi *Tree Traversal*

- V – *Visiting a node* (mengunjungi Simpul)
- L – Melintasi *subtree* ke kiri (*Left*)
- R – Melintasi *subtree* ke kanan (*Right*)

4.3 *Left To Right Traversal*

Mengunjungi setiap *node* pada *tree* secara rekursif pada kiri dan kanan *subtrees* suatu simpul.

- *Preorder*: V-L-R
  - *visiting the node*
  - *traversing the left subtree*
  - *traversing the right subtree*
- *Postorder*: L-R-V
  - *traversing the left subtree*
  - *traversing the right subtree*
  - *visiting the node*.
- *inorder*: L-V-R
  - *visiting the left subtree*
  - *visiting the node*
  - *traversing the right subtree*.

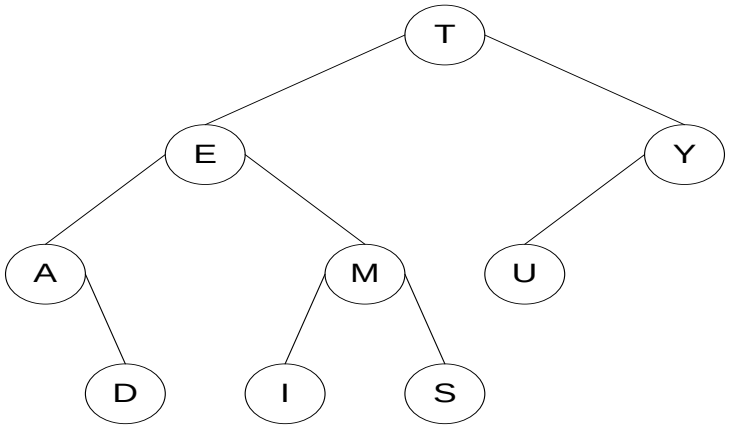
4.4 *Right to Left Traversal*

Kebalikan dari *Left-to-Right*, proses *traversal* dilakukan secara rekursif dari kanan ke kiri

- *Preorder*: V-R-L
  - *visiting the node*
  - *traversing the right subtree*
  - *traversing the left subtree*
- *Postorder*: R-L-V
  - *traversing the right subtree*
  - *traversing the left subtree*
  - *visiting the node.*
- *inorder*: R-V-L
  - *traversing the right subtree.*
  - *visiting the node*
  - *visiting the left subtree*

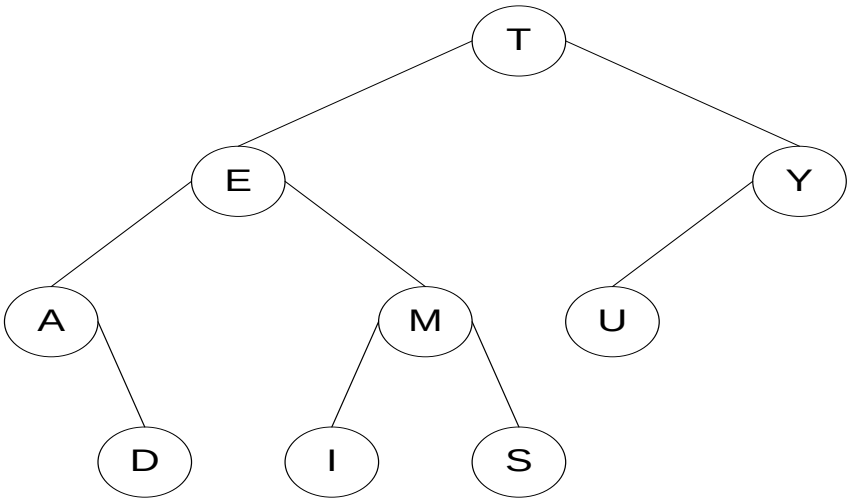
Pembahasan Tree Traversal berikut ini berdasarkan Left-To-Right Traversal, untuk penggunaan Right-To-Left Traversal dapat menyesuaikan

4.5 *Pre-order Traversal*



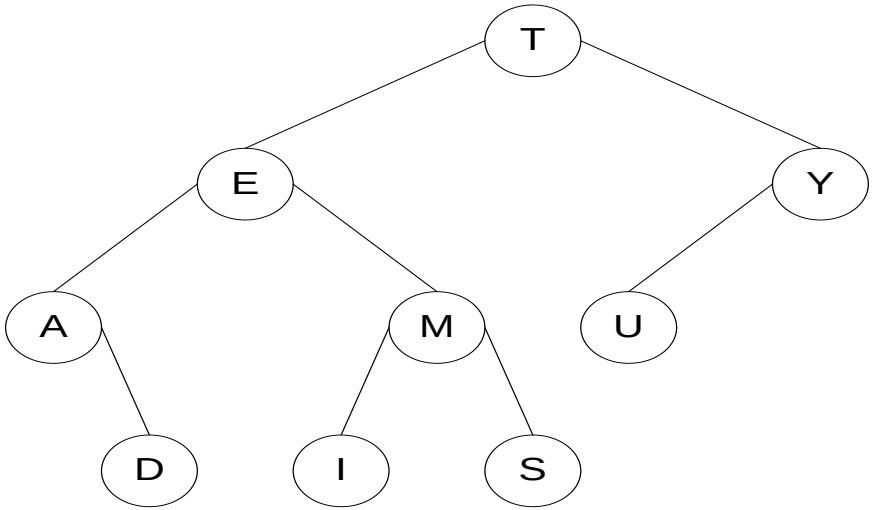
root - Visiting T – go left (E) - Visiting E – go left (A) – visiting A – go right (D) – visiting D – go parent (A) – go parent (E) - go right (M) – visiting M – go left (I) - visiting I – go parent (M) – go right (S) – visiting S – go parent (M) – go parent (E) – go parent (T) – go right (Y) - visiting Y – go left (U) – visiting U – finish

4.6 *Post-order Traversal*



root - go left (E) – go left (A) – go parent A – go right (D) – visiting D – go parent A – visiting A – go parent E – go right (M) – go left (I) – Visiting I - go parent M – go right (S) – visiting S – go parent M – visiting M – go parent E – visiting E – go parent T – go right (Y) – go left (U) – visiting U – go parent (Y) – visiting Y – go parent (T) – visiting T – finish

4.7 *In-Order Traversal*

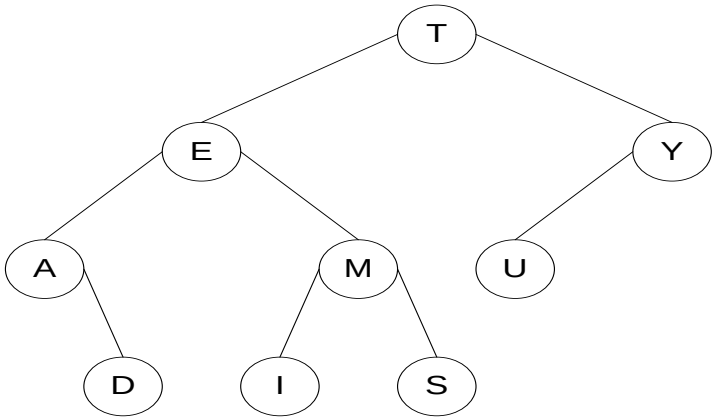


root (T) – go left (E) – go left (A) – visiting A – go right(D) - visiting D – go parent (A) – go parent (E) – visiting E – go right (M) – go left(I) - visiting I – go parent (M) – visiting M - go right (S) – visiting S – go parent (M) – go parent (E) – go parent (T) – visiting T – go right (Y) – go left (U) – visiting U – go parent (Y) – visiting Y – finish

5. Menemukan Induk Node

5.1 *Predecessor* dan *Successor*

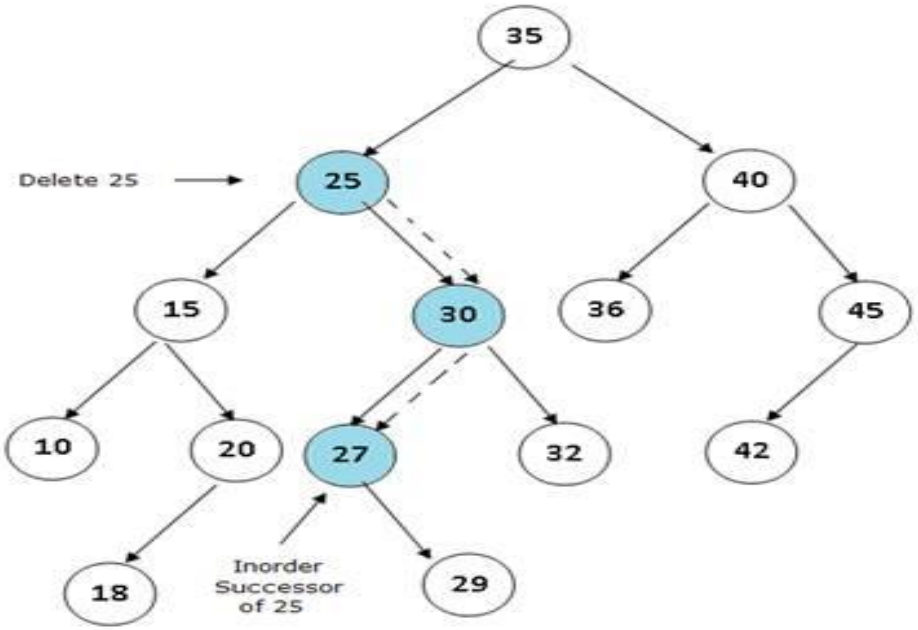
- Berdasarkan dari metode Traversal yang digunakan
- Predesesor (*Predecessor*) adalah node yang baru saja dikunjungi
- Suksesor (*Succesor*) adalah node yang akan dikunjungi
- Contoh pada *Traversal inorder* dari *tree* disamping:
- *Inorder-traversal*: A-D-E-I-M-B-T-U-Y
- Maka predesesor dari T adalah S, sedangkan suksesor dari T adalah U



4.1 *Predecessor* dan *Successor* (Lanj.)

Contoh Inorder Successor of node 25

Secara inorder, maka didapat successor dari 25 adalah 27



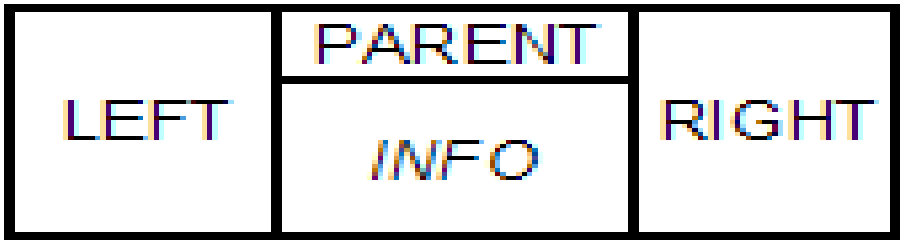
Sumber gambar: http://vle.du.ac.in/mod/book/view.php?id=5726&chapterid=3023

4.2 Menemukan Induk Dari Suatu Node

Dapat dilakukan dengan cara:

- Telusuri dari *root* sampai didapatkan induk dari node yang akan diproses → Pasti berhasil, tapi memakan waktu
- Gunakan *tree* dengan *link* ke induk → memerlukan *memory* yang lebih banyak, tapi setidaknya lebih baik daripada ide di atas
- Gunakan *Threaded Tree*

4.3 *Tree* Dengan Link Induk



- *Binary Tree* dimana setiap node nya memiliki informasi link ke induk
- *Elements*:
  - *Info*
  - *Left Child Link*
  - *Right Child Link*
  - *Parent Link*

4.4 *Threaded Tree*

- Pohon pencarian biner di mana masing-masing *node* menggunakan tautan anak kiri yang kosong untuk merujuk ke *node in-order predecessor* induk dan tautan anak kosong yang kanan untuk merujuk ke *node in-order successor* Induk dari Induk (*grand parent*)

http://www.nist.gov/dads/HTML/threadedtree.html

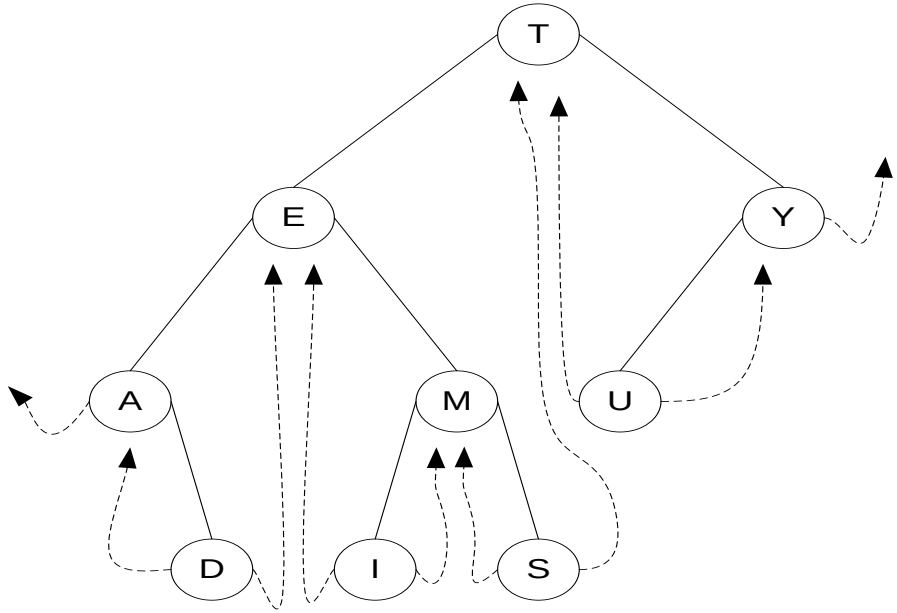
(October 17th 2008)

4.5 *Threaded Tree Elements*

- *Key*
- *Left Link*
- *Threaded Left* → Sebuah nilai boolean untuk mengindikasikan bahwa *left link* adalah *thread* ke *last visited node* (induk *in-order predecessor*)
- *Right Link*
- *Threaded Right* → Sebuah nilai boolean untuk mengindikasikan bahwa *right link* adalah *thread* ke *next node* yang akan dikunjungi (induk dari induk *in-order successor*)

4.5. *Threaded Tree Elements* (Lanj.)

4.6 Contoh *Threaded Tree*



6. Menghapus Simpul

6.1 Menghapus Simpul

Rules:

Setiap *key* pada *subtree* sebelah kanan harus lebih besar dari pada *key-key* pada *subtree* sebelah kiri.

5.2 Kondisi Penghapusan Simpul

Penghapusan tergantung pada kondisi

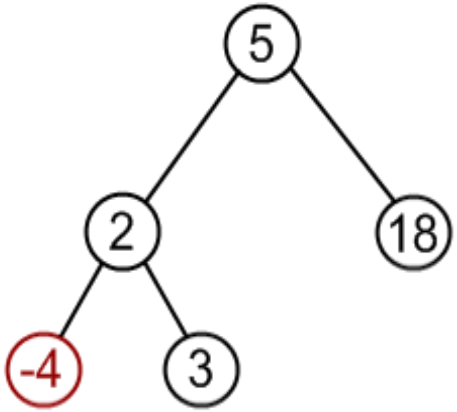
- Pada *Leaf* → hapus saja tanpa penangan masalah
- Pada node dengan satu *child* → *child* satu-satunya tersebut akan menjadi *child* dari *grandparent* menggantikan *node* yang dihapus
- Pada *Node* dengan dua *child* dapat dilakukan dengan cara:
  - *Deletion by merging*
  - *Deletion by copying*

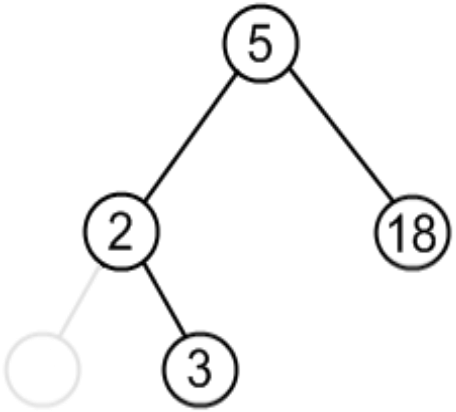
Gambar dapat dilihat pada masing-masing pembahasan di slide berikutnya

5.3 Hapus Pada *Leaf*

- Penghapusan simpul pada *leaf* dapat dilakukan tanpa perlakuan khusus apapun, karena sebagai *leaf*, suatu simpul tidak memiliki keturunan yang harus diatur agar memenuhi persyaratan node sebelah kiri harus lebih kecil daripada *node* sebelah kanan

5.3 Hapus Pada *Leaf* (Lanj.)



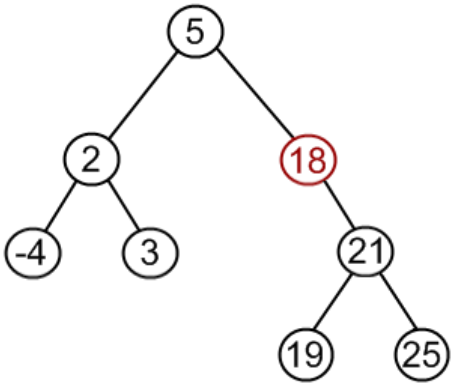


5.4 Hapus Simpul Dengan Satu Anak

- Node dengan satu anak hanya memiliki masalah menjadi anak dari induk manakah node dari anak yang dihapus tersebut
- Sedangkan anak dari anak node yang dihapus akan mengikuti anak dari node yang dihapus
- Maka perlakuan penghapusan node dengan satu anak dilakukan dengan anak satu-satunya tersebut akan menjadi anak dari *ancestor*/leluhur node yang dihapus

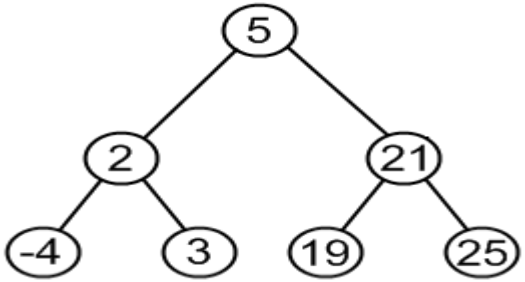
5.4 Hapus Simpul Dengan Satu Anak (Lanj.)

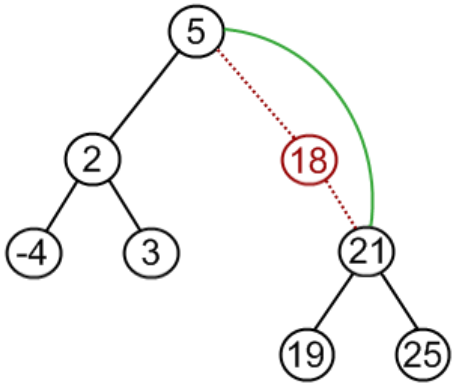
- Misal akan menghapus node 18



- Maka node 21 akan menjadi anak dari node 5 menggantikan node 18

- Hasil akhir





5.5 Penghapusan Pada Simpul Yang Memiliki 2 Anak

- Pada *Node* dengan dua *child* dapat dilakukan dengan cara:
  - *Deletion by merging*
  - *Deletion by copying*

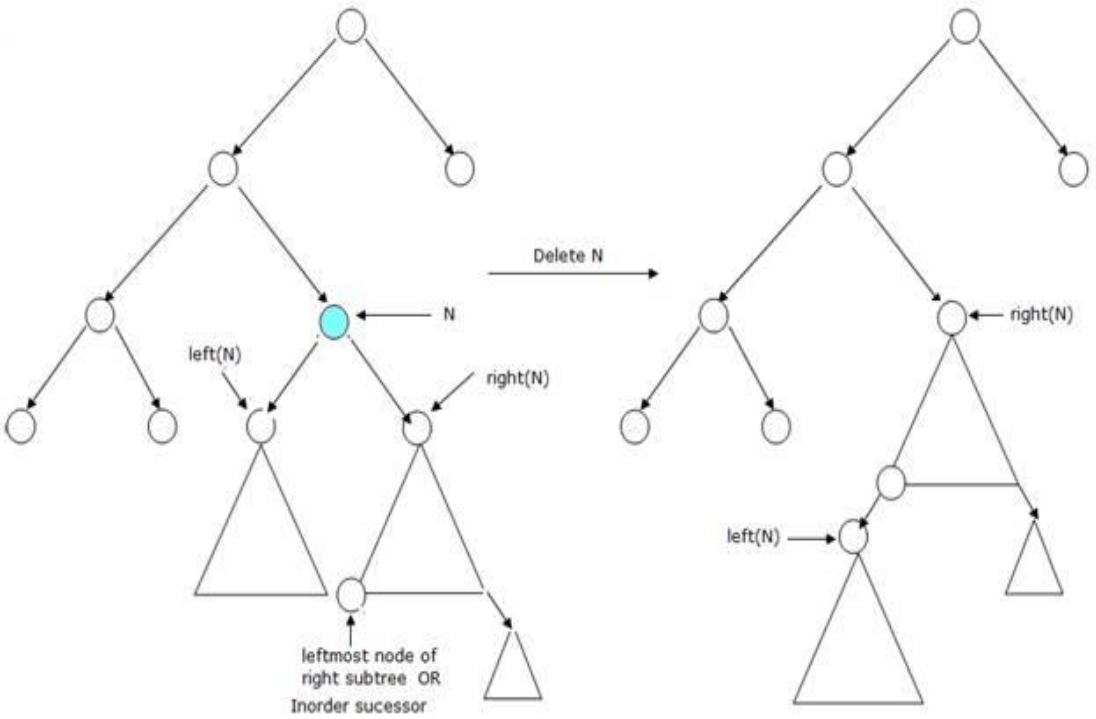
5.5.1 Delete By Merging

- *replace* ***right node*** sebagai *node* induk dan gabungkan *node* kiri ke *leftmost node* dari *subtree* kanan

  Atau lakukan kebalikannya

- *replace* ***left node*** sebagai *node* induk dan gabungkan *node* kanan ke *rightmost node* dari *subtree* kiri

5.5.1 Delete By Merging (Lanj.)



Sumber gambar:
http://vle.du.ac.in/mod/book/view.php?id=5726&chapterid=3011

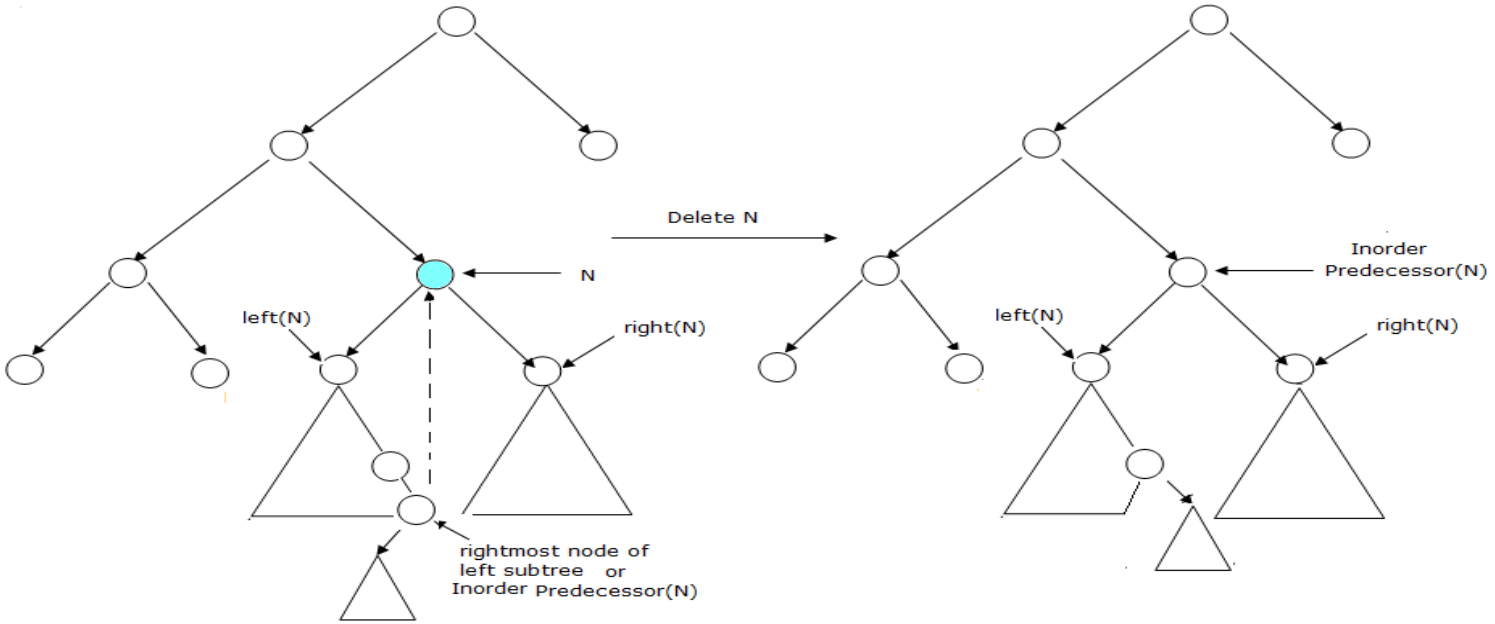
- *Deleting node* N dengan dua *children by merging right subtree into left subtree*
- *Node yang ditunjuk oleh N akan dihapus,*
- *Maka sub –tree yang ada di kanan node N akan menjadi pewaris menggantikan N*
- *Sedangkan sub tree di kiri Node N akan menjadi anak node terkiri dari sub tree yang semula di kanan Node N*

5.5.2 Delete by Copying

- Ganti ***key*** yang akan dihapus dengan *immediate predecessor* atau *immediate successor* (bisa juga dengan cara sebaliknya) dg cara:
  - Ambil *Subtree* sebelah kanan dari *node* yang akan dihapus, cari *leaf* yang paling kiri dari *node* yang harusnya dikunjungi secara *in-order*
  - Copykan *leaf* ke posisi *node* yang akan dihapus
  - Hapus *Leaf* yang sudah dicopykan menjadi node pengganti yang dihapus tadi

5.5.2 Delete by Copying (Lanj.)

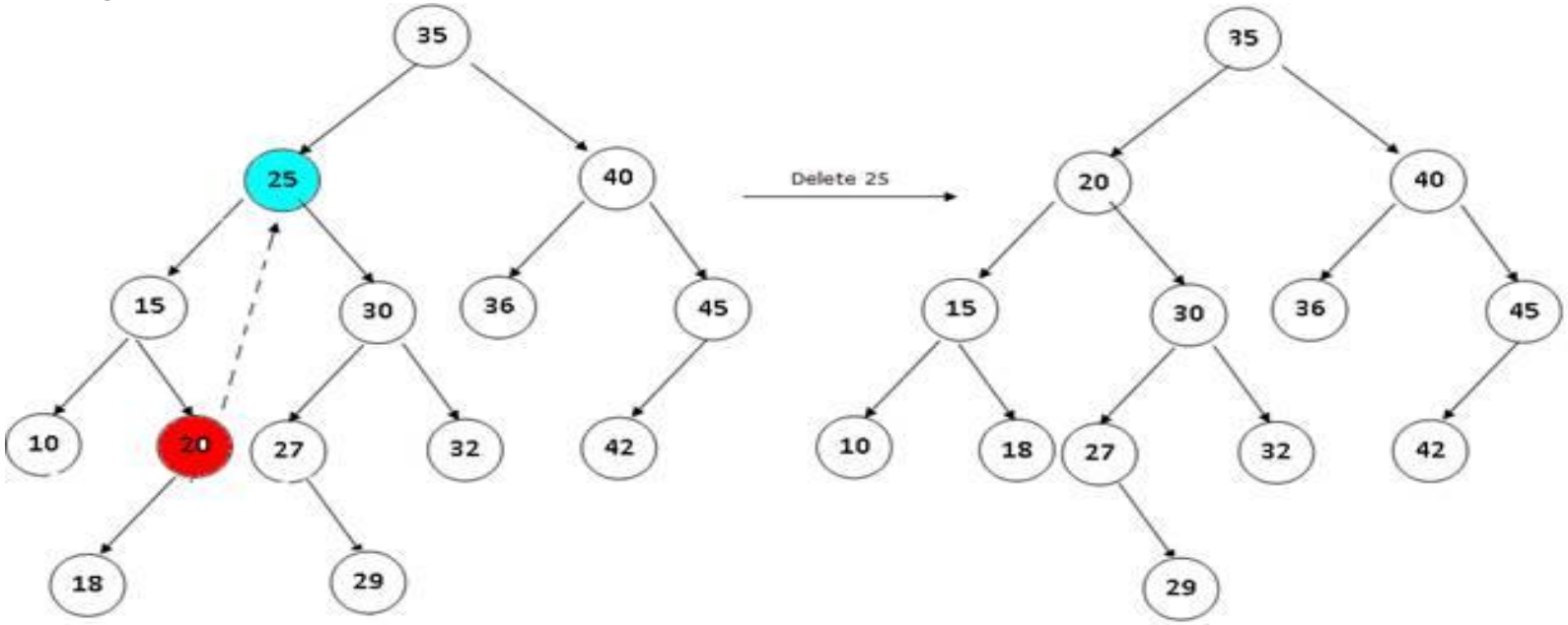
Deleting N *by copying In-order Predecessor*



Sumber gambar: http://vle.du.ac.in/mod/book/view.php?id=5726&chapterid=3023

5.5.2 Delete by Copying (Lanj.)

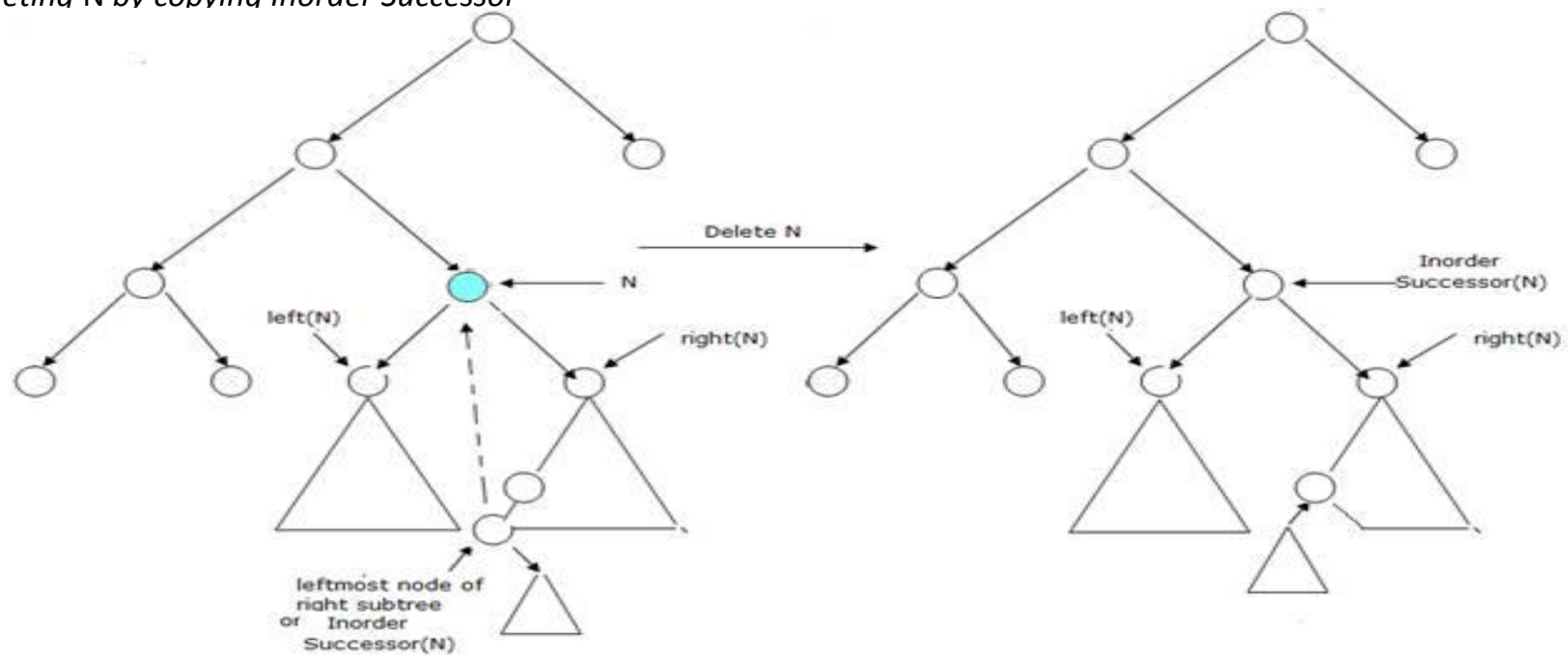
Contoh Hapus 25 dg *Deletion by copying* menggunakan *right-most inorder predecessor*



Sumber gambar: http://vle.du.ac.in/mod/book/view.php?id=5726&chapterid=3023

5.5.2 Delete by Copying (Lanj.)

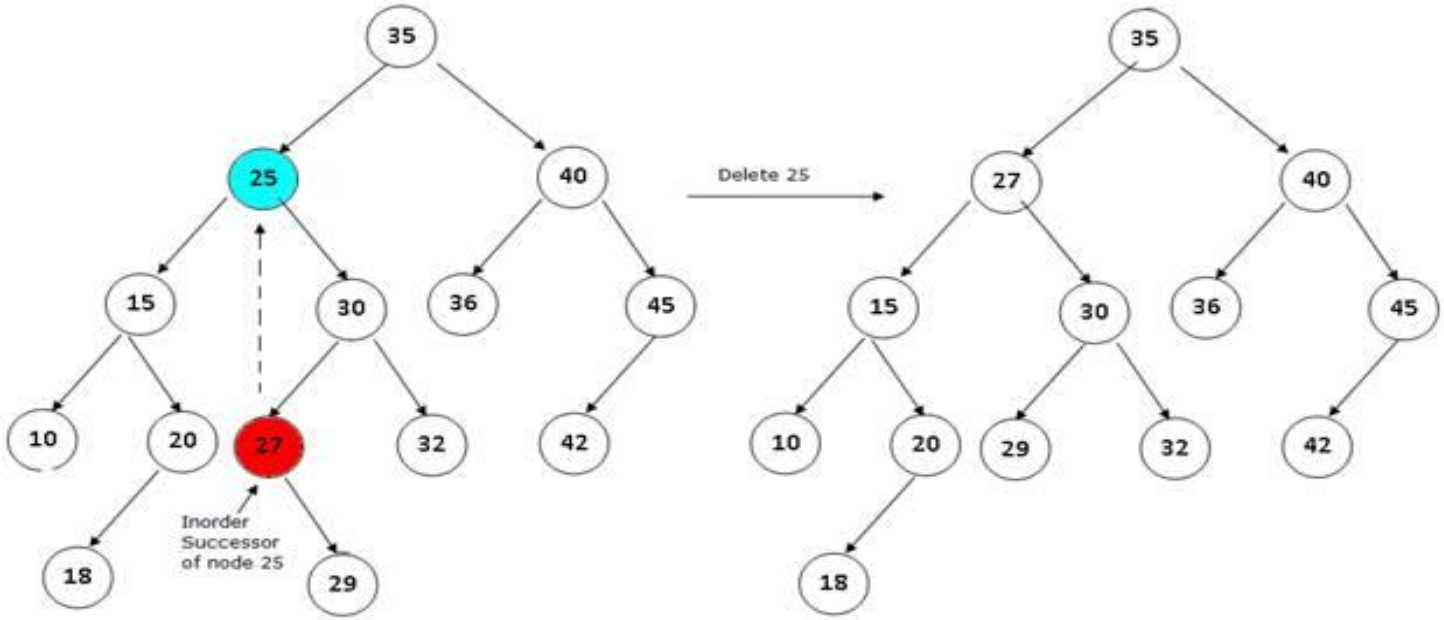
Deleting N *by copying Inorder Successor*



Sumber gambar: http://vle.du.ac.in/mod/book/view.php?id=5726&chapterid=3023

5.5.2 Delete by Copying (Lanj.)

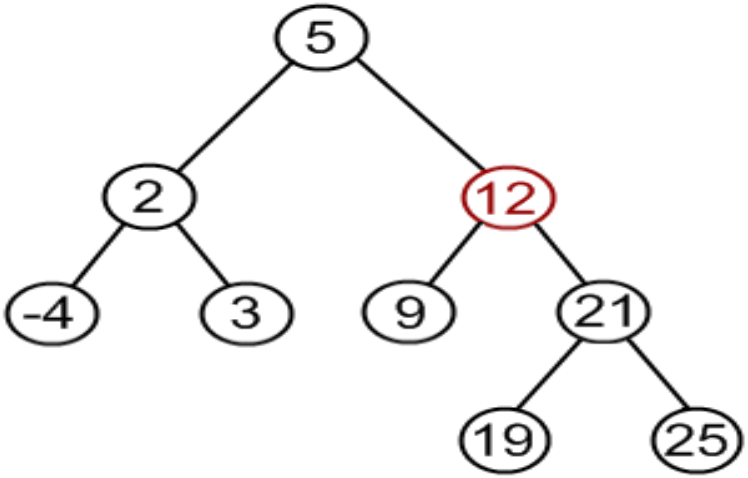
Contoh *Deleting* 25 dg *Deletion by copying* menggunakan *left-most inorder succesor*



Sumber gambar: http://vle.du.ac.in/mod/book/view.php?id=5726&chapterid=3023

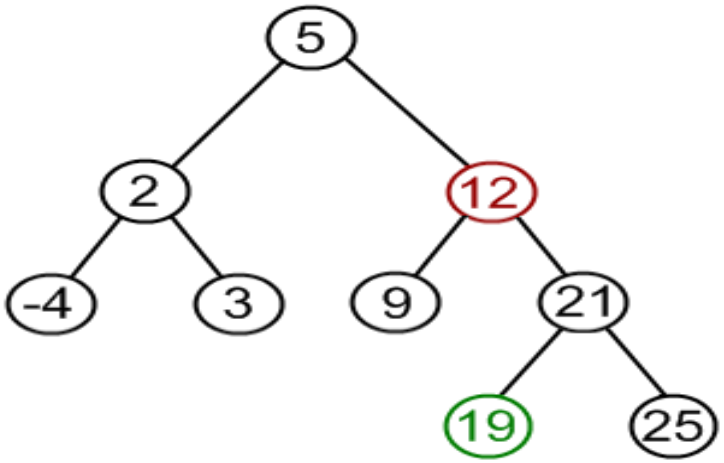
5.5.2 Delete by Copying (Lanj.)

Contoh lain: Misalkan Akan hapus *node* 12



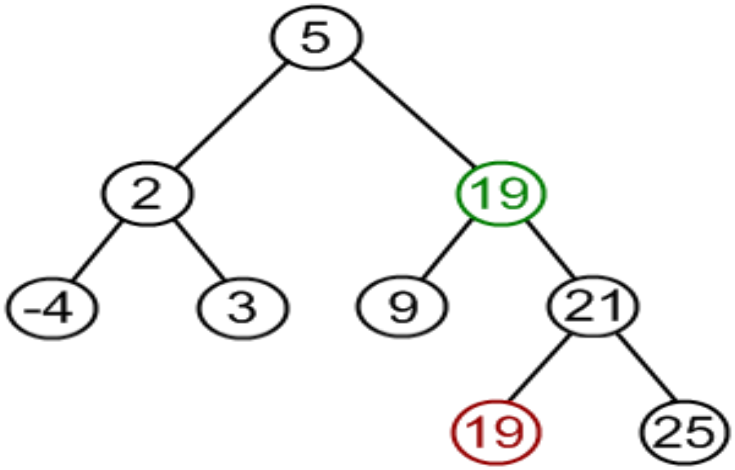
5.5.2 Delete by Copying (Lanj.)

Node Subtree Kanan yang paling kiri adalah 19



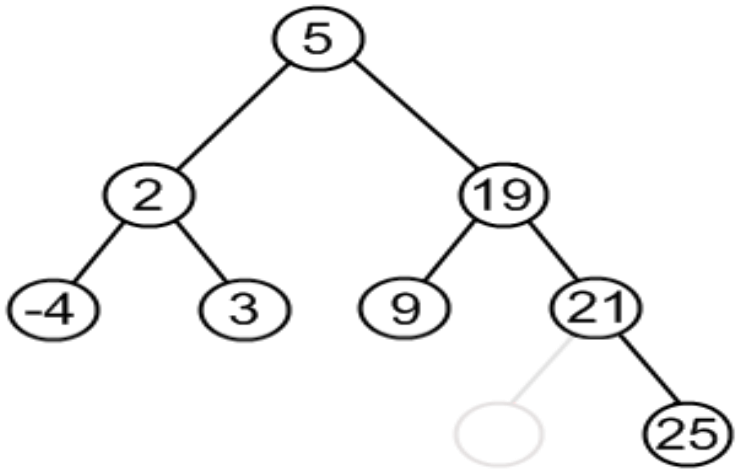
5.5.2 Delete by Copying (Lanj.)

Copykan *Leaf* Paling Kiri Tersebut ke *Node* Yang Akan Dihapus



5.5.2 Delete by Copying (Lanj.)

Musnahkan *Node* Yang Sudah Dicopykan Ke *Node* Yang Dihapus tersebut



Contoh In Order Traversal Dengan C++

```
void inOrderTrav(struct TheCell *travCell)
{
  if (travCell->kiri != NULL)
    inOrderTrav(travCell->kiri);
  cout<<travCell->dat<<" | ";
  if (travCell->kanan != NULL)
    inOrderTrav(travCell->kanan);
}
```

Contoh In Order Traversal Dengan Python

```
class TheCell:
    def __init__(self, dat):
        self.dat = dat
        self.kiri = None
        self.kanan = None

def in_order_trav(trav_cell):
    if trav_cell.kiri is not None:
        in_order_trav(trav_cell.kiri)
    print(trav_cell.dat, "| ", end="")
    if trav_cell.kanan is not None:
        in_order_trav(trav_cell.kanan)

# Example usage:
# Assuming 'TheCell' class is defined elsewhere, and you have a root_node
as the starting point
# You can call the in_order_trav function like this:
# in_order_trav(root_node)
```

PERINGATAN HAK CIPTA

Ringkasan

- *Tree* dengan paling banyak dua anak pada tiap *node* nya disebut *binary Tree*.
- Menambah simpul ada dua orientasi yaitu *left-to-Right* dan *Right-to-Left*
- Untuk *Left-to-Right*: Setiap *key* pada kanan *subtree* harus lebih besar dari setiap *key* di kiri *subtree* (berlaku kebalikannya untuk *Right-to-Left*)
- *Tree Traversal* adalah Proses mengunjungi setiap simpul pada *tree*/pohon biner tepat satu kali untuk setiap node
- Ada tiga metode *Traversal*:
  - *Pre-Order Traversal (VLR)*
  - *In-Order Traversal (LVR)*
  - *Post-Order Traversal (LRV)*

PERINGATAN HAK CIPTA

Terimakasih

TUHAN Memberkati Anda

Teady Matius Surya Mulyana (tmulyana@bundamulia.ac.id)