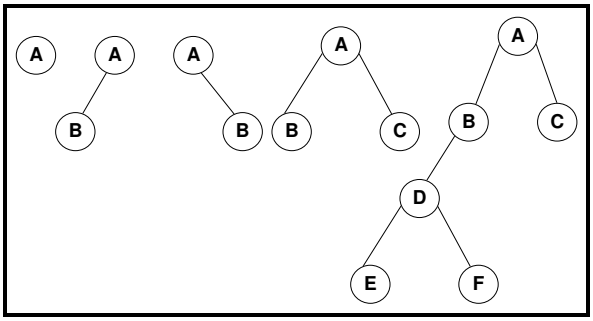
3.5 Binary Tree/Pohon Biner

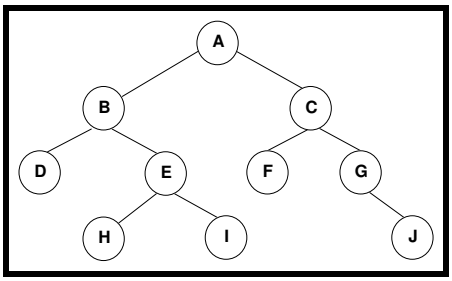
Definisi: *Merupakan pohon yang setiap simpulnya memiliki paling banyak dua cabang (anak).*



Gambar 23. Binary Tree

Banyak simpul maksimum pada tingkat ke-n dari *binary tree* adalah (2^{n}).
Banyak simpul maksimum dengan kedalaman n dari *binary tree* adalah ($2^{n+1} - 1$).

Contoh :



Tabel 3. Depth Binary Tree

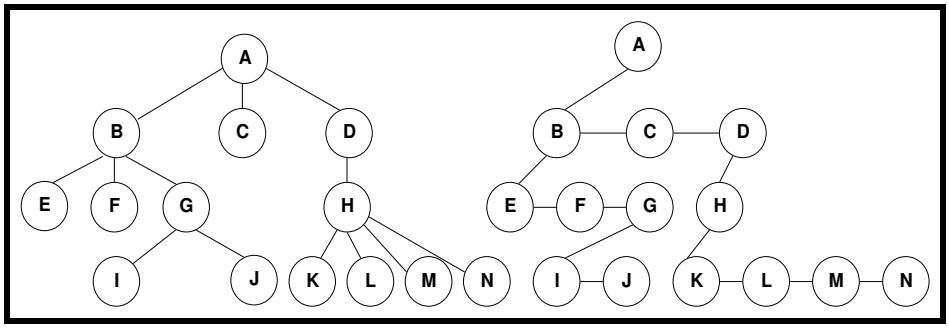
Tingkat	Banyak simpul
0	$2^{n} = 2^{0} = 1$
1	$2^{n} = 2^{1} = 2$
2	$2^{n} = 2^{2} = 4$
3	$2^{n} = 2^{3} = 8$

Tree disamping memiliki depth = 3 (~n), sehingga simpul maksimumnya adalah : ($2^{n+1} - 1$) = 15.

3.6 Mengubah tree non binary menjadi binary tree :

Mekanisme untuk melakukan konversi tree dari bentuk yang tidak biner (*Non Binary Tree*) menjadi tree biner, adalah:

- Cabang paling kiri (anak pertama) dari *tree non binary* merupakan anak sebelah kiri dari *binary tree ;*
- Saudara (*siblings*) dari cabang yang sudah menjadi anak sebelah kiri dari *binary tree,* menjadi cabang kanan dari cabang yang baru dibuat dalam *binary tree ;*
- Anak (*son*) pertama dari cabang yang sudah menjadi anak sebelah kiri dari *binary tree*, menjadi anak sebelah kiri dari *binary tree,*
- dan seterusnya.



Gambar 24. Konversi Non Biner ke Binary Tree

3.7 Traversal Binary Tree :

Pengertian : Traversal adalah proses mendatangi setiap simpul dari *Binary Tree* secara sistematik masing-masing satu kali.
Proses dilakukan terhadap isi simpul, dapat berupa :

- Pencetakan informasi (isi simpul), atau
- Perhitungan matematika.

Cara pemrosesan ini terbagi menjadi :

1. ***PREORDER,*** yaitu dengan memproses simpul tsb., kemudian proses anak sebelah kiri dan dilanjutkan ke simpul anak sebelah kanan (~ Urutannya : S L R);
2. ***POSTORDER,*** yaitu dengan memproses simpul anak sebelah kiri, kemudian proses anak sebelah kanan dan terakhir proses simpul tersebut (~ Urutannya : L R S);
3. ***INORDER,*** yaitu dengan memproses simpul anak kiri, kemudian proses simpul tersebut dan terakhir proses simpul anak sebelah kanan (~ Urutannya : L S R);
4. ***LEVELORDER,*** yaitu dengan memproses simpul ber-dasarkan tingkat dari simpul yang dilalui. Pemrosesan dimulai dari simpul tingkat satu sampai tingkat n (akhir).

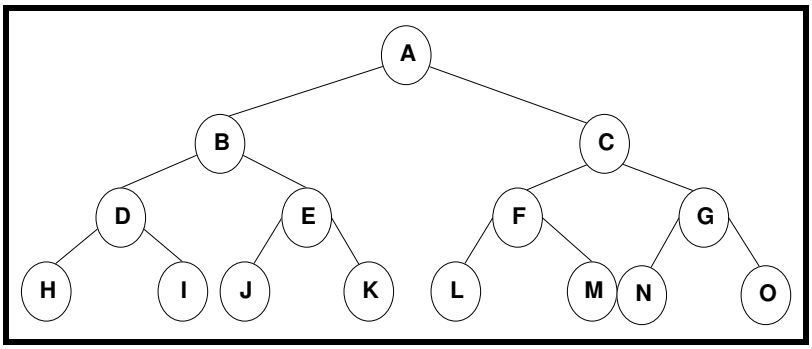
Untuk cara nomor 1 sampai 3 dapat juga dimulai dari simpul yang di kanan (SRL, RLS, RSL).

3.8 Representasi Binary Tree

Binary Tree dapat direpresentasikan melalui :

- Cara Sekuensial yaitu menggunakan Array.
- Cara Linked List.

3.8.1 Representasi Secara Sequensial



Gambar 25. Representasi Sequensial dari Binary Tree

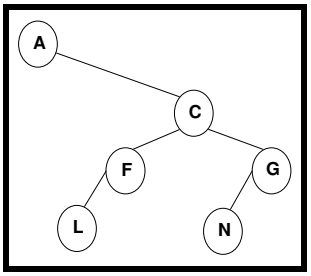
Maka diperlukan array dengan 15 elemen untuk menyimpannya, dengan aturan :

- Root selalu berada pada elemen pertama;
- Simpul-simpul pada tingkat ke-2 berada di elemen ke-2 sampai dengan elemen ke-3;
- Simpul-simpul pada tingkat ke-3 berada di elemen ke-4 sampai dengan elemen ke-7;
- Simpul-simpul pada tingkat ke-4 berada di elemen ke-8 sampai dengan elemen ke-15.

Secara umum :

- Simpul pada tingkat ke - i terletak pada array dengan index 2^{i} sampai dengan $2^{I+1} - 1$;
- Anak-anak elemen ke - i adalah (2*i) dan (2*i + 1) dan parent dari elemen ke - i adalah (i div 2) atau $\lfloor i/2 \rfloor$

Permasalahan : Penyimpanan jadi tidak hemat jika *binary tree*nya miring/*skewed.*

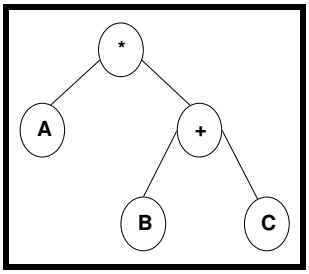


Gambar 26. Skewed Binary Tree

karena masih diperlukan array sebanyak
(2^4 - 1) = 15 elemen.

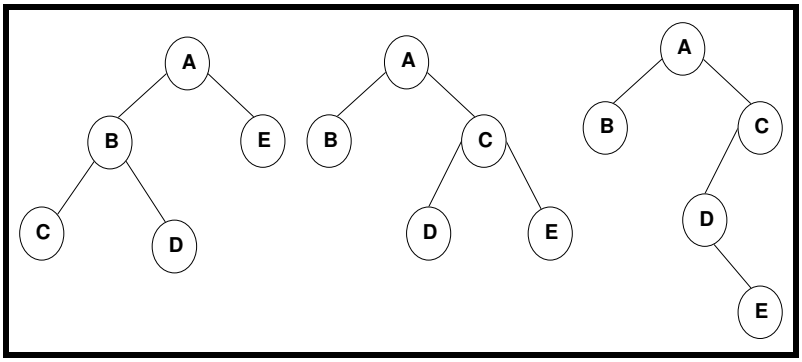
Secara umum *binary tree*, dengan berbagai jenis traversal, dapat disajikan secara sequensial, misalnya :

Kasus 1:



Notasi prefix : *A+BC
Notasi postfix : ABC+*
Notasi Infix : A*B+C

Kasus 2 :
Jika diberikan notasi prefix ABCDE, maka dapat dibuat binary tree sebagai berikut :



Gambar 27. Variasi Binary Tree

Dalam traversal preorder, untuk ekspresi matematika (=Kasus 1), OPERATOR menjadi ROOT dan OPERAND menjadi CABANG.
Dalam Kasus 2 tidak diketahui mana yang menjadi ROOT dan mana yang menjadi CABANG, sehingga menghasilkan banyak variasi tree yang terbentuk. Untuk itu, diperlukan informasi lain yaitu jumlah anak (derajat) dari setiap simpul.

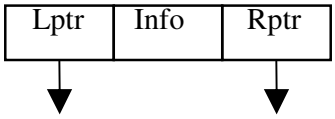
Contoh traversal preorder dan jumlah anaknya :

- Preorder : A B C D E F G H I J K L
  Jml. Anak: 4 0 3 0 0 0 0 2 2 0 0 0
- Preorder : P Q R S T U V W X Y Z
  Jml. Anak: 3 2 0 0 4 0 1 0 0 0 0

Representasi dalam arraynya adalah dengan mengambil simpul yang memiliki anak dan terkanan, secara rekursif.

3.8.2 Representasi Binary Tree Secara Linked List

Dengan menggunakan struktur Node dalam linked list sebagai berikut :

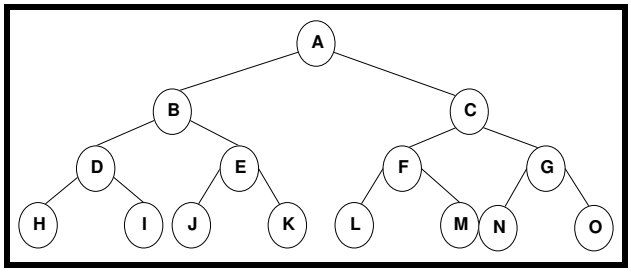


Gambar 28. Representasi Berkait dari Node Tree

maka pohon biner dengan N simpul akan memiliki :

- Jumlah pointer seluruhnya adalah : 2 N;
- Banyaknya pointer yang terpakai adalah (N - 1), yaitu pointer percabangannya;
- Banyaknya pointer yang tidak terpakai (karena menunjuk ke NIL) adalah (N + 1).

Ilustrasi rumusan di atas :

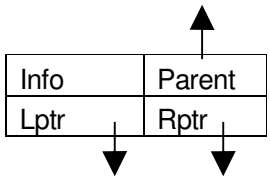


Gambar 29. Binary Tree Representation

Selain banyaknya pointer yang tidak terpakai, ada juga permasalahan lainnya, yaitu bagaimana caranya untuk mencari parent dari suatu simpul (yang bukan ROOT), hal ini diperlukan untuk memproses suatu simpul agar tidak selalu dimulai dari simpul ROOT, yang membutuhkan waktu akses tersendiri (mengurangi pemborosan waktu).

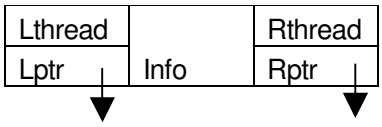
3.9 Pohon Threaded

Definisi : Pohon yang setiap simpulnya memiliki pointer yang menunjuk ke simpul *parent* (ayahnya), selain memiliki pointer untuk menunjuk ke cabangnya.
Pointer yang menunjuk ke *parent* (ayahnya) disebut dengan **THREAD.** Struktur data pada pohon threaded :



Gambar 30. Struktur data Node Binary Tree

Ide pohon threaded di atas menambah alamat pointer sebanyak 50 %, untuk mengatasinya dapat dilakukan :
Penambahan dua field *Boolean* pada setiap simpul, yaitu *Lthread* dan *Rthread*, sehingga pointer yang dibutuhkan hanya dua buah



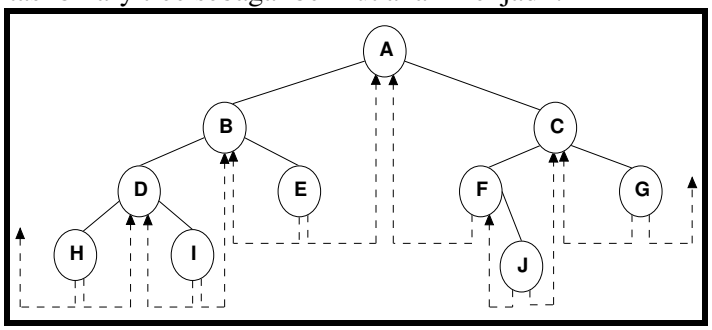
Gambar 31. Struktur data Node Threaded Binary Tree

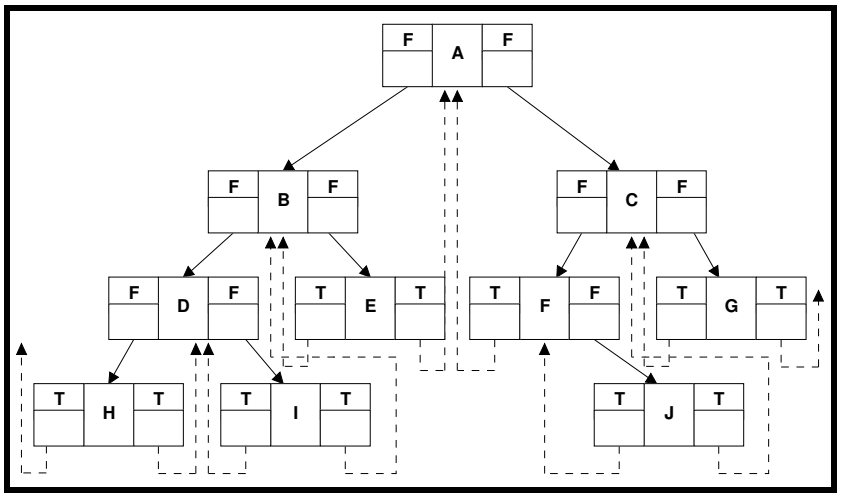
Lthread dan *Rthread* diisi dengan "TRUE" bila pointer Lptr dan Rptr menunjuk ke simpul *parent,* dan diisi dengan "FALSE" bila Lptr dan Rptr menunjuk ke pencabangannya (anaknya).

Aturan lengkap pohon Threaded :

- Jika p^.rptr = NIL, maka isilah dengan alamat simpul yang segera akan dikunjungi secara inorder (*inorder sucessor*);
- Jika p^.lptr = NIL, maka isilah dengan alamat simpul yang akan dikunjungi tepat sebelumnya secara inorder (*inorder predecessor*);

Sehingga representasi binary tree sebagai berikut akan menjadi :





Gambar 32. Konversi Binary Tree – Threaded Binary Tree

Gambar Pohon Threaded yang terkatung-katung

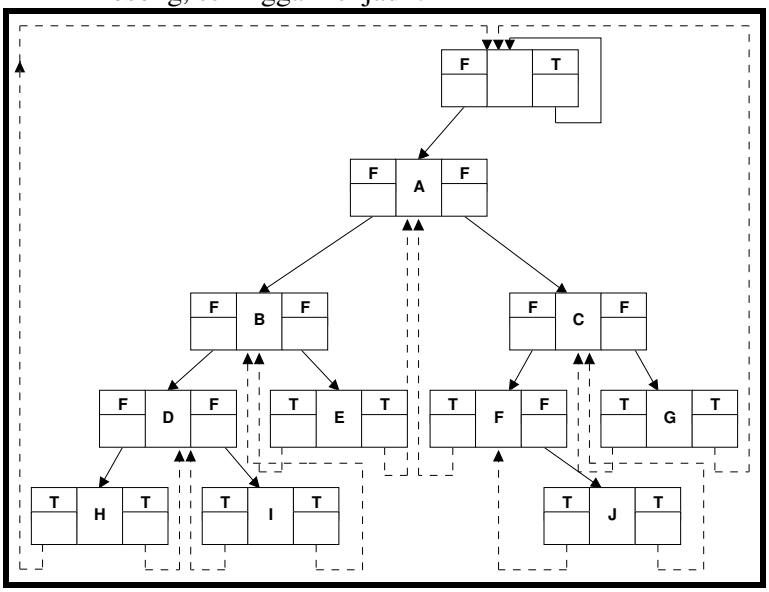
Hal-hal yang perlu diperhatikan dalam penggunaan pohon *Threaded* adalah :

- Konversi tentang *Thread* simpul awal dan akhir traversal;
- Algoritma pencarian simpul penerus (*sucessor*) dan simpul pendahulu (*predecessor*) secara *inorder* suatu simpul yang diketahui (begitu pula untuk *preorder* dan *post order*);
- Algoritma pencarian *parent* (ayah) dari suatu simpul;
- Penyisipan sebuah simpul ke dalam pohon *Threaded.*

Penanganan masalah di atas :

Kasus 1 :

Dibuat simpul HEADER kosong, sehingga menjadi :

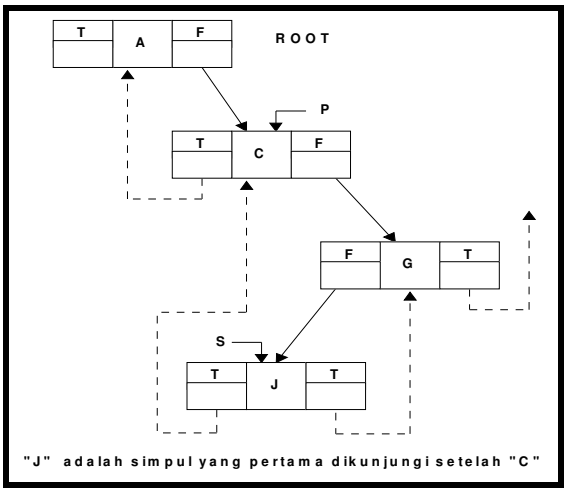


Gambar 33. Optimasi Threaded Binary Tree

Kasus 2 :

Algoritma untuk menentukan simpul penerus (*suces-sor*) dan simpul pendahulu (*predecessor*) secara *inorder* dari suatu simpul yang diketahui adalah :

- Diketahui simpul p;
- Jika p^.rptr ≠ NIL dan p^.Rthread = TRUE, maka p^.rptr adalah penerus simpul p;
- Jika p^.rptr ≠ NIL dan p^.Rthread = FALSE, maka p^.rptr adalah cabang kanan dari simpul p;
- p dapat berupa simpul pertama dari bagian kanan pohon tersebut. Simpul (*inorder*) berikutnya dapat diketahui dengan menelusuri bagian kiri p^.rptr sampai simpul s, sehingga s^.lthread = TRUE, misalkan :



Gambar 34. Successor dan Predecessor dalam Threaded Binary Tree

Kasus 3 :

Pencarian *parent* (ayah) dapat dilakukan dengan fungsi sebagai berikut :

```
  Type     tTree = ^tree;
           tree = record
                info : char;
                Lthread, Rthread : boolean;
                Lptr, Rptr : tTree;
           End;

Function ParentThread(p : tTree) : tTree;
Var q : tTree;
Begin
q ← p
While (not q^.Lthread) do q ← q^.Lptr

{Melihat ke simpul pendahulu /inorder}
q ← q^.Lptr
if (q^.Rptr = p)
  then ParentThread ← q
  else begin   {p sebagai  anak kiri}
       q ← p
       while (not Q^.Rthread) do q ← q^.Rptr
       ParentThread ← q^.Rptr
  end
End
```

Beberapa hal menarik dari manfaat pohon *Threaded* :

- Proses menambah simpul baru atau menghapus simpul mudah untuk diterapkan;
- Pencarian simpul penerus dan pendahulu dari traversal *postorder/preorder* tidak semudah pada traversal *inorder*.

3.10 Lexicographic Binary-Tree

Data pada tabel dibawah ini merupakan objek proses searching dengan ***Nama*** sebagai key.

Tabel 4. Searching berbantukan Binary Tree

Nomor-Induk	Nama
001	B
002	E
003	C
004	G
005	A
006	F
007	D

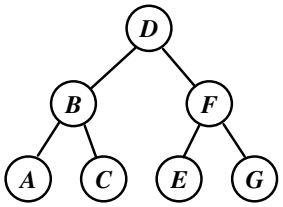
Diasumsikan pula bahwa frekwensi relatif proses searching adalah sbb :

key yang dicari	frekwensi-relatif
A	1
B	1
C	1
D	1
E	1
F	1
G	1
lebih besar G	1

Data tersebut akan disusun sebagai node node dalam struktur binary tree yang biasa disebut *Lexicographic-Binary-Tree*. Susunan node dibuat sedemikian rupa sehingga proses searching dapat dilakukan dengan aturan sbb :

- searching dimulai dari *Root*
- bandingkan *Key yang dicari* dengan *Key node ybs*,
  - jika (*Key yang dicari*) > (*Key node ybs*)
    maka *pergi ke Right-Son* dari node ybs
    jika *Right-Son tidak ada* maka *GAGAL*
  - jika (*Key yang dicari*) < (*Key node ybs*)
    maka *pergi ke Left-Son* dari node ybs
    jika *Left-Son tidak ada* maka *GAGAL*
  - jika (*Key yang dicari*) = (*Key node ybs*)
    maka berarti *SUDAH KETEMU* di node ybs
- demikian seterusnya lakukan proses searching sehingga *KETEMU*, atau *GAGAL*.

Maka bentuk binary tree ideal (yang menghasilkan kecepatan searching rata rata yang paling tinggi) adalah bentuk yang menghasilkan internal path length minimum seperti pada gambar disamping ini (pada node hanya dicantumkan key node ybs).



bentuk lexicographic binary tree yang ideal

Nilai internal path length = 0 + (2x1) + (4x2) = 10
Path length rata rata = 10/7

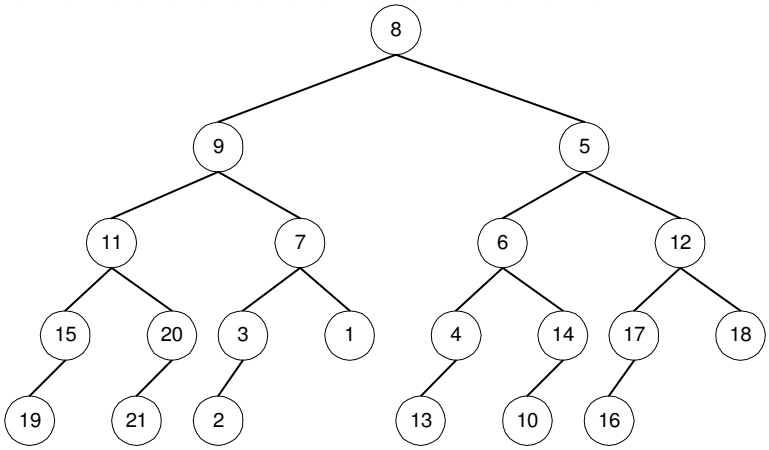
3.11 Extended Binary tree

Beberapa bentuk *extend* dari *binary tree* adalah:

1. *Lexicographic binary tree,* yaitu *binary tree* yang disusun secara terurut dengan patokan nilai yang lebih kecil ke *subtree kiri* dan nilai yang lebih besar ke *subtree* kanan;
2. *Balanced binary tree,* yaitu *binary tree* yang seimbang. Sebuah *binary tree* dikatakan seimbang jika selisih ketinggian *subtree kiri* dan *subtree kanan* dari setiap node yang dimiliki tidak lebih dari 1; Tree yang seimbang ini biasa juga disebut dengan AVL-tree (*Adelson-Velskii dan Landis*; yang menemukan metoda penyeimbangan tree)
3. *Perfectly balance binary tree,* yaitu *binary tree* yang *balance* dan memiliki kedalaman yang optimal untuk jumlah seluruh node yang dimiliki.

   Pembuatan *binary tree* yang *perfectly balance* ini dapat dilakukan dengan mendefinisikan rumusan berikut:
   - Definisikan nilai-nilai elemen yang dimiliki oleh *binary tree;*
   - Dari n buah elemen, ambil elemen pertama sebagai *Root;*
   - Bikin subtree kiri dengan jumlah elemen sesuai formula → nl = $\lceil$ n div 2 $\rceil$
   - Bikin subtree kanan dengan jumlah elemen sesuai formula → nr = n-nl-1

   Contoh *perfectly balance tree* untuk 21 elemen data dibawah ini:
   8, 9, 11, 15, 19, 20, 21, 7, 3, 2, 1, 5, 6, 4, 13, 14, 10, 12, 17, 16, 18 adalah



Gambar 35. Perfectly Balanced Binary Tree

Procedure *Create Balance Tree,* adalah sebagai berikut:

```
Program BuildTree
Type BTree = ^node
  Node = record
       Info : integer;
       LS, RS : BTree;
  End;
Var  n : integer;
  Root : BTree;
```

```
Function tree(n:integer): Btree;
{Membuat perfectly balanced tree dengan n buah node}
Var NewNode : Btree;
  X, nl, nr : integer;
Mulai
  If n = 0 then tree ← nil
            Else nl ← n div 2;  nr ← n – nl – 1;
                  Read(keyboard) X
                  NewNode ← Alloc (1, Node)
                  If (NewNode <> Nil)
                  Then NewNode^.Info ← X
                        NewNode^.LS ← tree(nl)
                        NewNode^.RS ← tree(nr)
                        Tree ← NewNode
                  EndIf
  EndIf
End_function Tree

Procedure PrintTree(t:Btree; h:integer);
{Menampilkan tree dengan indentation h}
Var i : integer;
Mulai
  If t <> Nil
     Then
PrintTree (t^.LS, h+1)
For i←1 to h do
  Write (layar)' '
  Writeln(layar)t^.Info
  PrintTree(t^.RS, h+1)
  EndIf
End_procedure PrintTree

Mulai
  Read(keyboard) n
  Root ← Tree(n)
  PrintTree(Root,0)
End_program BuildTree
```

4. Multiway binary tree → 2-3 tree, dll.

3.11.1 Insert dan Delete pada Binary Tree

3.11.1.1 Insert Tree

Operasi Insert pada sebuah binary tree, dapat dilakukan di posisi :

- Node yang merupakan *Daun/Leaf*
- Node yang memiliki sebuah cabang (anak kiri saja atau anak kanan saja) ➔ Melengkapi jumlah anak jadi dua

Insert pada binary tree (yang umum) dapat dilakukan dengan menentukan posisi elemen tree yang baru untuk kemudian diaturkan posisinya dengan "menyambungkan" pointer LS dan RS dari elemen tree yang sudah ada.

Insert pada Lexicographic Binary tree:

Untuk melakukan penambahan elemen pada sebuah binary tree yang terurut (seperti *lexicographic binary tree)*, maka proses insert dapat dilakukan dengan memperhatikan:

- Boleh duplikasi/tidaknya data dalam lexicographic (*Uniq/tidak*); jika boleh tidak uniq maka perlu ditambahkan satu field yang menampung data *jumlah* elemen yang sama;
- Posisi Insert ditentukan dengan menggunakan aturan (untuk sort Ascending) jika nilai elemen yang akan diinsertkan lebih kecil dari nilai elemen pada node aktif, maka telusuri anak kiri; dan jika lebih besar maka telusuri anak kanan.
- Insert dilakukan di posisi yang tepat dengan mengubah pointer LS dan RS dari node yang terlibat.

Procedure pencarian posisi untuk *Insert data* pada *Lexicographic binary tree* adalah sebagai berikut:

```
Procedure Search(x:integer; Root:Btree);
{Mencari posisi yang sesuai untuk Insert Lexicographic BT}
Var d : integer;  {-1 = 'L', 0 = 'B', 1 = 'R'}
  p1, p2 : Btree;
Mulai
  p2 ← Root; p1 ← p2^.RS; d ← 1;
  while (p1 <> Nil) and (d <> 0) do
     p2 ← p1
     if (x < p1^.info)
        then p1 ← p1^.LS; d ← -1
        else if (x > p1^.info)
              then p1 ← p1^.RS; d ← 1
              else d ← 0
              EndIf
     EndIf
  EndWhile

  If (d = 0) then p1^.count := p1^.count + 1 {if Not uniq}
             Else p1 ← Alloc (1,Node)
                  If (p1 <> Nil)
                     Then p1^.info ← x
                        p1^.LS ← nil
                        p1^.RS ← nil
                        p1^.count ← 1
                        If d < 0 then p2^.LS ← p1
                           Else p2^.RS ← p1
                        EndIf
                  EndIf
  EndIf

End_procedure Search
```

Insert pada Balanced Binary tree:

Untuk melakukan penambahan elemen pada sebuah binary tree yang seimbang, maka proses insert dapat dilakukan dengan tetap memperhatikan balancing dari tree tersebut. Proses balancing dapat dilakukan dengan memperhatikan setiap node apakah 'L' (miring ke kiri), 'R' (miring ke kanan) atau 'B' (balance). Jika ada node yang level nya berturutan dan ditemui LL maka dilakukan **rotasi** ke Kanan, begitu pula untuk RR maka rotasi ke kiri.

Setelah itu baru dilakukan penghitungan apakah setiap node selisih ketinggian subtree kiri dan subtree kanannya maksimal 1, jika terpenuhi syarat tersebut maka sudah balance. Jika syarat tidak terpenuhi maka cari node yang berturutan levelnya yang LR atau RL untuk dilakukan **rotasi ganda**.
Jika rotasi sampai melibatkan *root* maka dikenal dengan istilah ***Restrukturisasi* Tree.**

3.11.1.2 Delete Elemen Tree

Operasi Delete elemen pada sebuah binary tree, dapat dilakukan di posisi :

- Node yang merupakan *Daun/Leaf* ➔ Langsung Delete
- Node yang memiliki sebuah cabang (anak kiri saja atau anak kanan saja) ➔ Sehingga anaknya akan menggantikan posisi parent yang dihapus.
- Node dengan jumlah anak 2 (*complete node*) ➔ posisi yang dihapus digantikan oleh elemen terkanan (jika dikunjungi InOrder) dari *subtree Kiri* atau oleh elemen terkiri (jika dikunjungi InOrder) dari *subtree Kanan,* yang memiliki 1 anak. Jika kedua buah *subtree* memiliki 2 anak, dapat dipilih salah satu.

Delete elemen pada binary tree (yang umum) dapat dilakukan dengan menentukan posisi elemen tree yang akan di delete, untuk kemudian diaturkan posisinya dengan "menyambungkan" pointer LS dan RS dari elemen tree yang sudah ada.

Delete elemen pada Balanced Binary tree:

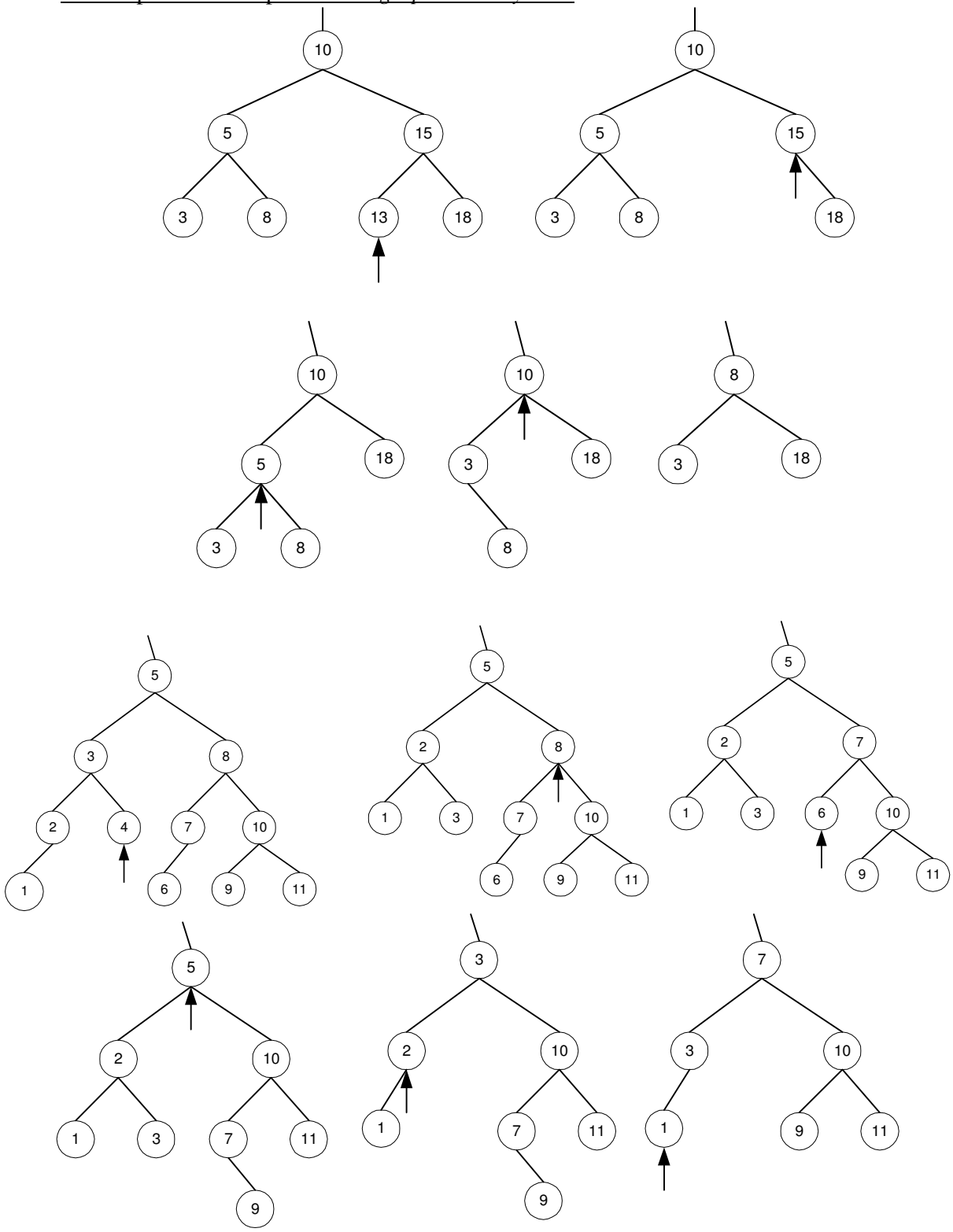
Untuk melakukan penghapusan elemen pada sebuah binary tree yang terurut dan *balanced*, maka proses insert dapat dilakukan dengan memperhatikan procedure di bawah ini:

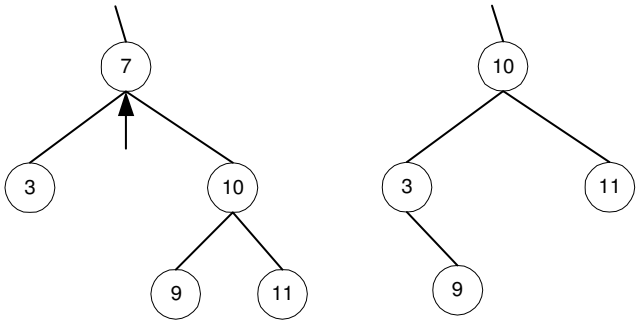
```
Procedure Delete (x:integer; Var p:Btree);
Var q : Btree;

  Procedure del (var r : Btree);
  Mulai
     If (r^.RS <> Nil
  Then  del (r^.RS)
  Else q^.Info ← r^.Info
     q ← r
     r ← r^.LS
  EndIf
  End_Procedure_Del
Mulai
  if (p = Nil)then write(layar) 'Tidak ada tree'
     else if x < p^.info then delete(x, p^.LS)
               else if x > p^.info then delete(x, p^.RS)
                         else {delete yang ditunjuk p}
                            q ← p
                            if (q^.RS = Nil) then p ← q^.LS
                               else  if (q^.LS=Nil)
                                        then p ← q^.RS
                                        else del(q^.LS)
                                     EndIf
                            EndIf
                            Dealloc(q)
                      EndIf
            EndIf
  EndIf

End_procedure Delete
```

Contoh proses delete pada *Lexicographic Binary tree:*





Gambar 36. Delete Node pada Balanced Binary Tree

{*Catatan: Procedure Balanced Tree Deletion, dapat diacu di buku 5,* halaman 223 – 225}

3.12 Kesimpulan

Struktur Pohon (*Tree*) adalah struktur yang penting dalam bidang informatika, yang memungkinkan proses:

- Pengorganisasian informasi berdasarkan suatu struktur lojik
- Melalui cara pengaksesan khusus terhadap suatu elemen.

Algoritma yang dipentingkan dalam sebuah struktur data tree adalah bagaimana menyusun/mengorganisasikan struktur Tree sesuai dengan kebutuhan dunia nyata , juga bagaimana mengakses setiap elemen data dalam struktur data tree tersebut (proses Traversal).
Representasi organisasi data hirarki dilakukan dengan memperhatikan keterkaitan data/informasi yang dimiliki dalam proses leveling yang dibutuhkan. Leveling dapat dilakukan dengan menerapkan struktur Non biner dan untuk optimalisasi algoritma yang lebih sederhana maka dapat dilakukan transformasi ke dalam bentuk Binary tree.
Binary tree dapat dimodifikasi lebih lanjut agar proses *searching*/pencarian data dan *sorting*/pengurutan data dapat dilakukan, yaitu dengan menerapkan *balancing binary tree*.