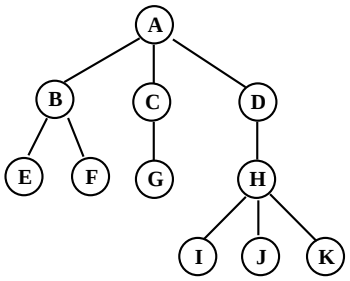
# 4.6. PELAKSANAAN TRAVERSING & ALGORITMAnya.

## TRAVERSING pada REPRESENTASI LINKED LIST.

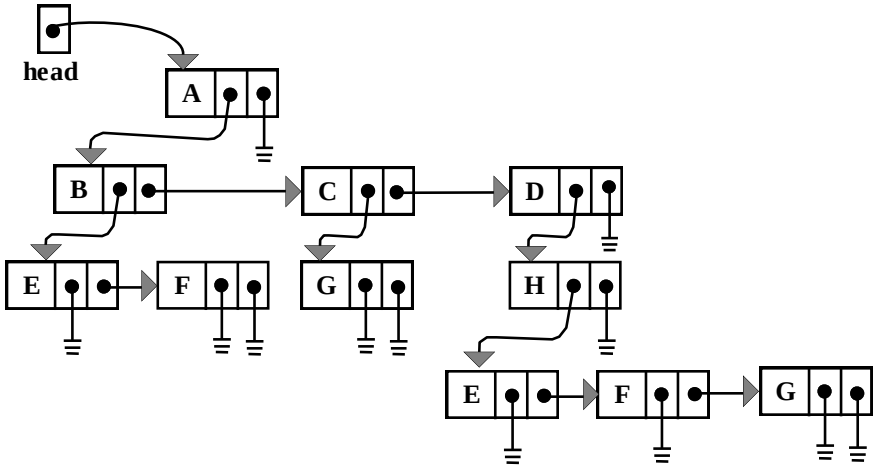
Pada definisi traversing ( lihat bab 4.2) disebutkan bahwa traversing adalah mengunjungi setiap node pada sebuah tree masing masing 1x saja. Definisi ini bersifat ideal, karena pada pelaksanaannya traversing kadang kadang terpaksa mengunjungi beberapa node lebih dari 1x. Meskipun demikian tetap hanya 1x kunjungan yang "resmi", karena kunjungan yang selebihnya bersifat "numpang-lewat" ( terpaksa melalui node ybs untuk mengunjungi node yang lain ).
Sebagai ilustrasi perhatikan kasus contoh berikut untuk traversing post-order.




Pada tree disamping ini akan dilakukan traversing post-order. Maka secara ideal urutan kunjungan adalah sbb :

E, F, B, G, C, I, J, K, H, D, A

Dengan cara representasi linked; dimana setiap node mempunyai 2 pointer; masing masing first-son dan next-brother maka diperoleh gambar berikut.




1. Pelaksanaan traversing post-order akan dimulai dengan mempergunakan nilai pointer head untuk "mengunjungi" node A. Kunjungan ini bersifat "numpang-lewat", karena harus dilakukan dalam rangka mengunjungi node E ( node yang seharusnya pertama dikunjungi pada traversing post-order ).
   Secara singkat dapat ditulis dengan notasi sbb :

   head
   $\Rightarrow$ A

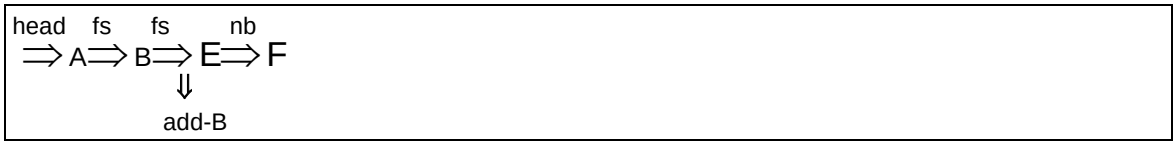
2. Selanjutnya untuk mencapai node E, harus dikunjungi secara "numpang-lewat" node B. Node B dicapai dengan menggunakan isi pointer first-son dari node A. Dapat ditulis :


3. Dari node B, dengan menggunakan isi pointer first-son, akan dicapai node E. Kunjungan ke node E ini bersifat "resmi" ( sesuai dengan bentuk ideal traversing post-order ). Dapat ditulis :

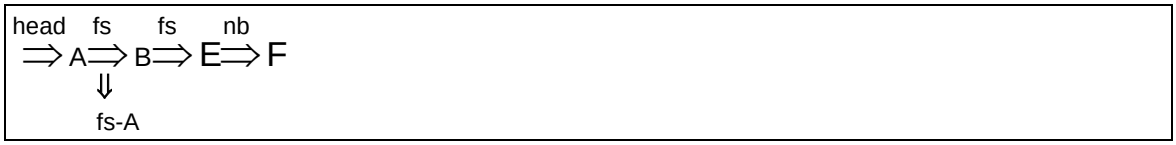

4. Selanjutnya dengan menggunakan isi pointer next-brother E, akan dikunjungi node F secara resmi. Proses total dapat ditulis :


5. Node berikut yang harus dikunjungi adalah node B. Pointer di node F tidak ada yang dapat digunakan untuk membantu mencapai node B. Akan tetapi ingat bahwa sebenarnya node B telah dikunjungi secara "numpang-lewat". Jika pada saat itu alamat node B "disimpan"/ disalin kedalam variabel tertentu, maka sekarang akan dapat dipergunakan untuk mencapai node B.
   Penyimpanan dapat dilakukan melalui salah satu cara sbb :
   - Pada saat sedang di node B : simpan alamat node B.
   - Atau, pada saat sedang di node A : simpan nilai pointer first-sonnya ( = alamat node B ).

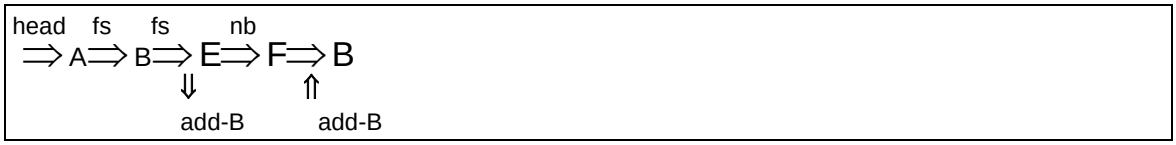
   Dapat ditulis sbb :




   atau

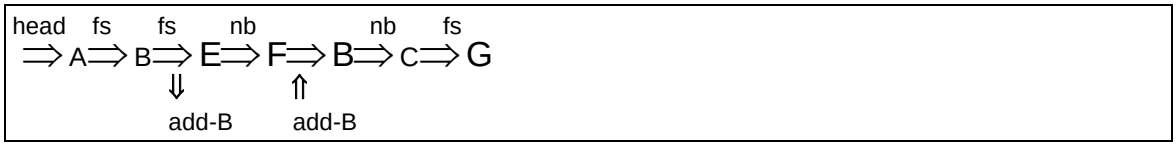



   Jika diambil cara yang pertama, maka pemanfaatannya untuk melanjutkan proses traversing dapat ditulis sbb :

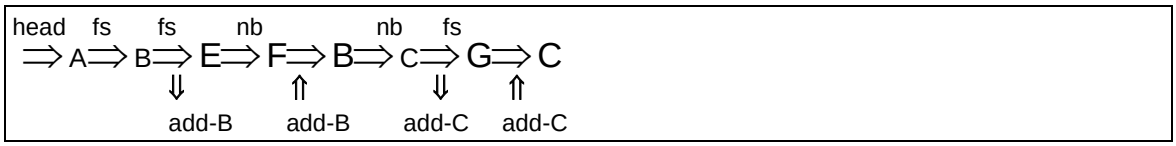



6. Untuk melanjutkan proses traversing akan dikunjungi node G, karena sekarang kita ada di node B, maka harus dilakukan hal sbb :
   - gunakan nilai pointer next-brother B untuk mengunjungi C, kunjungan bersifat "numpang-lewat"
   - kemudian gunakan nilai pointer first-son C untuk mengunjungi node G

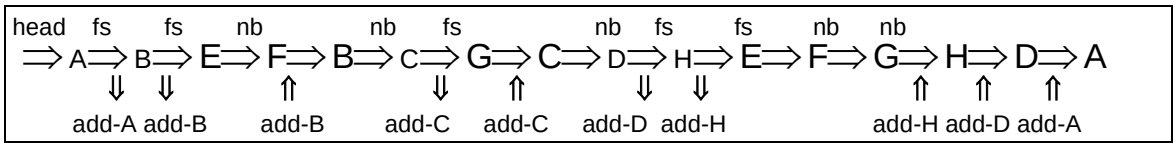
   atau :



7. Kunjungan berikut ke node C ditangani dengan cara yang sama dengan masalah kunjungan resmi terhadap B, sehingga diperoleh :

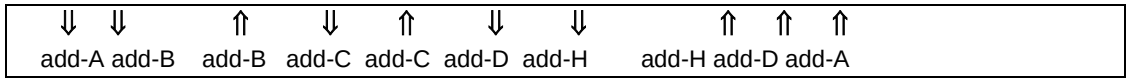



8. Demikian seterusnya sehingga secara lengkap proses traversing post-order dapat ditulis sbb :

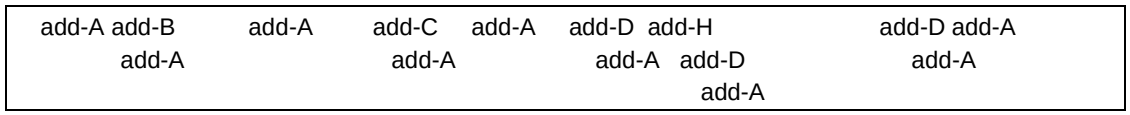



Amatilah proses penyimpanan dan pengambilan kembali nilai alamat ! Terlihat ada sifat sbb : *alamat yang paling akhir disimpan akan menjadi alamat pertama yang diambil,* atau *Last-In-First-Out*.
Dapat disimpulkan bahwa secara konseptual struktur tempat penyimpanan sementara tersebut adalah sebuah ***stack***.
Sejarah isi stack tersebut adalah sbb :

operasi




isi stack




catatan : ⇓ berarti *push*, dan ⇑ berarti *pull/pop*.

Operasi yang dilakukan pada setiap tahapan traversing diatas dapat dituliskan secara lebih formal. Untuk itu perlu terlebih dahulu didefinisikan hal hal sbb :

- Satuan alokasi : pointer first-son disebut sebagai *fs*, dan next-brother disebut sebagai *nb*.
- Penyimpanan alamat ke stack dilakukan dengan melalui pemanggilan prosedur *push(alamat)*.
- Pengambilan alamat dari stack dilakukan dengan melalui pemanggilan prosedur *pull()*.
- Variabel pointer yang bernama *pcur* berisi alamat node yang sedang dikunjungi.

Maka dapat dibuat ikhtisar operasi dalam bentuk tabel sbb :

| tahap | posisi di | status kunjungan | operasi yang dilakukan | node yang dituju |
|---|---|---|---|---|
| 1. | - | | pcur ← head | A |
| 2. | A | ⇨ | push(pcur)<br>pcur ← pcur↑.fs | B |
| 3. | B | ⇨ | push(pcur)<br>pcur ← pcur↑.fs | E |
| 4. | E | ☑ | pcur ← pcur↑.nb | F |
| 5. | F | ☑ | pcur ← pull() | B |
| 6. | B | ☑ | pcur ← pcur↑.nb | C |
| 7. | C | ⇨ | push(pcur)<br>pcur ← pcur↑.fs | G |
| 8. | G | ☑ | pcur ← pull() | C |

| | | | | |
|---|---|---|---|---|
| dst | C | ☑ | pcur ← pcur↑.nb | D |
| | D | ⇨ | push(pcur)<br>pcur ← pcur↑.fs | H |
| | H | ⇨ | push(pcur)<br>pcur ← pcur↑.fs | I |
| | I | ☑ | pcur ← pcur↑.nb | J |
| | J | ☑ | pcur ← pcur↑.nb | K |
| | K | ☑ | pcur ← pull() | H |
| | H | ☑ | pcur ← pull() | D |
| | D | ☑ | pcur ← pull() | A |
| | A | ☑ | | |

*Tabel Ikhtisar Traversing Post-Order*

⇨ status kunjungan : "numpang-lewat"
☑ status kunjungan : sesuai urutan post-order

Analisis terhadap ikhtisar operasi pada tabel tersebut diatas menghasilkan kesimpulan sbb :

- Ada 4 macam kumpulan operasi yang mungkin dilakukan pada traversing, ialah.
  - (a) pcur ← head ( inisiasi )
  - (b) push(pcur) ( berarti bergerak "*kebawah*" )
    pcur ← pcur↑.fs
  - (c) pcur ← pcur↑.nb ( berarti bergerak "*kekanan*" )
  - (d) pcur ← pull() ( berarti bergerak "*keatas*" )

- Kecuali kelompok ke a yang berfungsi inisiasi, pemilihan kelompok operasi yang harus dilaksanakan di suatu node tergantung dari 2 macam kondisi sbb :
  - "*arah*" proses traversing sebelumnya : *kebawah*, *keatas* atau *kekanan*.
  - "*karakteristik*" node ybs : punya atau tidaknya *first-son* dan *next-brother*.

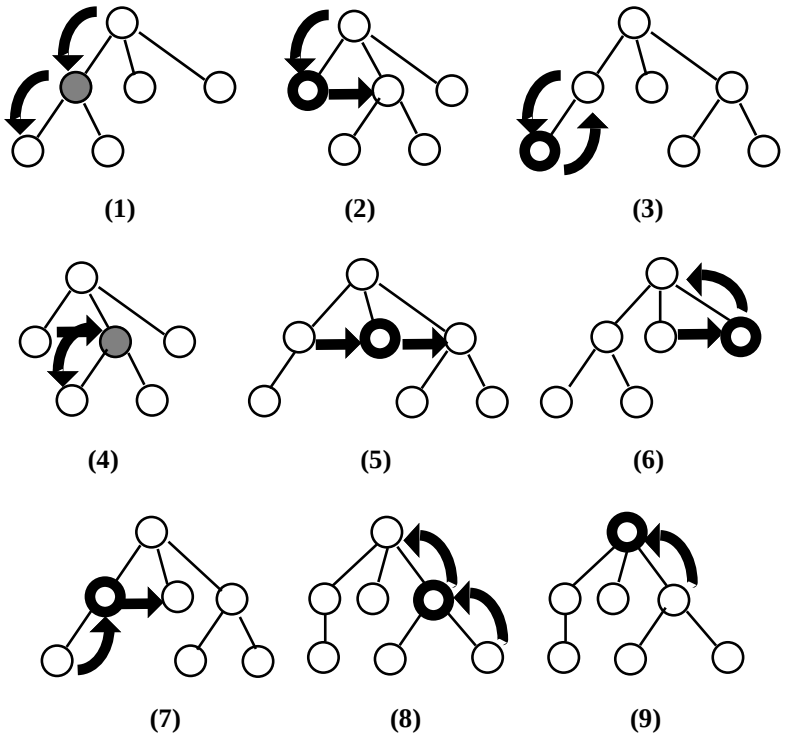
  Hanya ada beberapa kombinasi arah 2 buah gerak proses traversing yang berturutan.

- Kunjungan pada node current dapat ditentukan jenisnya ( "*numpang-lewat*" atau "*resmi*" ) setelah ditentukan kumpulan operasi yang dipilih untuk dilaksanakan ( atau berarti arah gerak berikutnya ).

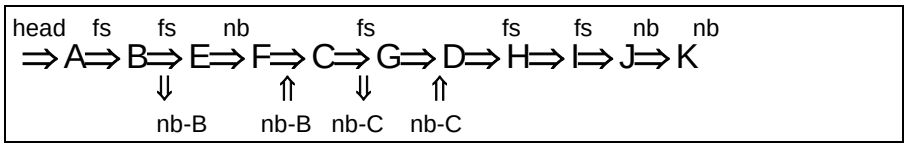
Ikhtisarnya dapat dilihat pada tabel dibawah ini. Tabel hasil analisis ini nantinya akan dipergunakan untuk membuat *algoritma non-rekursif traversing post-order* tree.

***Tabel Ikhtisar Pemilihan Kumpulan-Operasi pada Traversing Post-order***

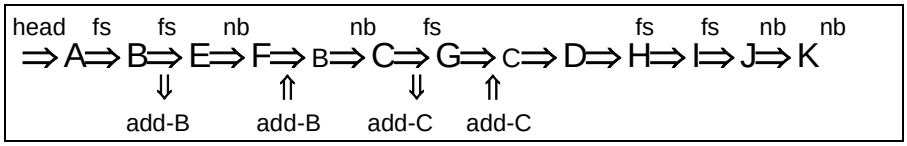
| *no* | *arah gerak terakhir* | *karakteristik node current* | *arah gerak selanjutnya* | *kumpulan operasi yang dipilih* | *status kunjungan pada node current* |
|---|---|---|---|---|---|
| 1 | kebawah | first-son ada | kebawah | (b) | numpang-lewat |
| 2 | | first-son tdk-ada, next-brother ada | kekanan | (c) | resmi |
| 3 | | first-son tdk ada, next-brother tdk-ada | keatas | (d) | resmi |
| 4 | kekanan | first-son ada | kebawah | (b) | numpang-lewat |
| 5 | | first-son tdk-ada, next-brother ada | kekanan | (c) | resmi |
| 6 | | first-son tdk ada, next-brother tdk-ada | keatas | (d) | resmi |
| 7 | keatas | next-brother ada | kekanan | (c) | resmi |
| 8 | | next-brother tdk-ada | keatas | (d) | resmi |
| 9 | | =root | stop | - | resmi |




Untuk contoh tree yang sama dengan diatas, pelaksanaan traversing pre-order dapat diikhtisarkan sbb :
Traversing pre-order ideal : A, B, E, F, C, G, D, H, I, J, K, L
Pelaksanaannya :




atau




Ikhtisar operasi untuk versi pertama :

| tahap | posisi di | status kunjungan | operasi yang dilakukan | node yang dituju |
|---|---|---|---|---|
| 1. | - | | pcur ← head | A |
| 2. | A | ☑ | pcur ← pcur↑.fs | B |
| 3. | B | ☑ | push(pcur↑.nb)<br>pcur ← pcur↑.fs | E |
| 4. | E | ☑ | pcur ← pcur↑.nb | F |
| 5. | F | ☑ | pcur ← pull() | C |
| 6. | C | ☑ | push(pcur↑.nb)<br>pcur ← pcur↑.fs | G |
| 7. | G | ☑ | pcur ← pull() | D |

| | | | | |
|---|---|---|---|---|
| 8. | D | ☑ | pcur ← pcur↑.fs | H |
| 9. | H | ☑ | pcur ← pcur↑.fs | I |
| 10. | I | ☑ | pcur ← pcur↑.nb | J |
| 11. | J | ☑ | pcur ← pcur↑.nb | K |
| 12. | K | ☑ | | - |

*Tabel Ikhtisar Traversing Pre-Order*

⇨ status kunjungan : "numpang-lewat"
☑ status kunjungan : sesuai urutan pre-order

Kumpulan operasinya :

(a) pcur ← head ( inisiasi )
(b) pcur ← pcur↑.fs ( berarti bergerak "*kebawah jenis 1*" )
(c) push(pcur↑.nb) ( berarti bergerak "*kebawah jenis 2*" )
pcur ← pcur↑.fs
(d) pcur ← pcur↑.nb ( berarti bergerak "*kekanan*" )
(e) pcur ← pull() ( berarti bergerak "*menyeberang*" )

Sedangkan ikhtisarnya adalah sbb :

***Tabel Ikhtisar Pemilihan Kumpulan-Operasi pada Traversing Pre-order***

| *arah gerak terakhir* | *Karakteristik node current* | *isi stack* | *arah gerak selanjutnya* | *kumpulan operasi yang dipilih* |
|---|---|---|---|---|
| tidak perlu diperiksa | first-son ada, next-brother ada | - | kebawah (2) | (c) |
| | first-son ada, next-brother tdk-ada | - | kebawah (1) | (b) |
| | first-son tdk ada, next-brother ada | - | kekanan | (d) |
| | first-son tdk ada, next-brother tdk-ada | ada | menyeberang | (e) |
| | first-son tdk ada, next-brother tdk-ada | kosong | stop | - |

Perhatikan bahwa :

- Status kunjungan *selalu resmi*, tidak ada kunjungan numpang-lewat !
- Pemilihan operasi *tidak perlu menyertakan pemeriksaan arah gerak terakhir* !

## ALGORITMA TRAVERSING ( non-rekursif ).

Dengan menggunakan tabel ikhtisar hasil analisis pelaksanaan traversing pada bagian terdahulu, maka bentuk algoritma untuk post-order dan pre-order pada tree bukan biner.

### Algoritma non-rekursif traversing post-order pada tree non-biner

Dengan asumsi bahwa :

- *NIL, KEATAS, KEBAWAH* dan *KEKANAN* adalah *konstanta* yang sesuai.
- *arah* adalah variabel yang isinya mewakili arah traversing.

Bentuk kondisi ( merujuk pada tabel ikhtisar traversing post-order ) :

| | jenis-node | | Kondisi | | |
|---|---|---|---|---|---|
| arah | fs | nb | kode | Deskripsi | operasi |
| ⇩ | ✓ | - | *(k1)* | (arah=*KEBAWAH* ) and (pcur↑.fs<>*NIL*) | *(b)* |
| ⇩ | × | ✓ | *(k2)* | (arah=*KEBAWAH*) and ((pcur↑.fs=*NIL*) and (pcur↑.nb<>*NIL*)) | *(c)* |

| arah | jenis-node | | Kondisi | | operasi |
|---|---|---|---|---|---|
| | fs | nb | kode | Deskripsi | |
| ⇩ | × | × | *(k3)* | (arah=*KEBAWAH* ) and ((pcur↑.fs=*NIL*) and (pcur↑.nb=*NIL*)) | *(d)* |
| ⇨ | ✓ | - | *(k4)* | (arah=*KEKANAN*) and (pcur↑.fs<>*NIL*) | *(b)* |
| ⇨ | × | ✓ | *(k5)* | (arah=*KEKANAN* ) and ((pcur↑.fs=*NIL*) and (pcur↑.nb<>*NIL*)) | *(c)* |
| ⇨ | × | × | *(k6)* | (arah=*KEKANAN* ) and ((pcur↑.fs=*NIL*) and (pcur↑.nb=*NIL*)) | *(d)* |
| ⇧ | - | ✓ | *(k7)* | (arah=*KEATAS*) and (pcur↑.nb<>*NIL*) | *(c)* |
| ⇧ | - | × | *(k8)* | (arah=*KEATAS*) and (pcur↑.nb=*NIL*) | *(d)* |
| ⇧ | root | | *(k9)* | (arah=*KEATAS*) and (pcur=head) | - |

- kondisi untuk pemilihan kumpulan operasi *(b)* ( "kebawah" ) :
  *(k1)* or *(k4)* ⇒ (arah<>*KEATAS*) and (pcur↑.fs<>*NIL*)

- kondisi untuk pemilihan kumpulan operasi *(c)* ( "kekanan" ) :
  *(k2)* or *(k5)* or *(k7)* ⇒

- kondisi untuk pemilihan kumpulan operasi *(d)* ( "keatas" ) :
  *(k3)* or *(k6)* or *(k8)* ⇒

- kondisi untuk "stop"/ menghentikan pengulangan :
  *(k9)* or ("*tree-kosong*") ⇒ *(k9)* or *(head=NIL)*
  atau dapat disederhanakan dengan jalan sbb :
  tambahkan operasi pcur ← *NIL* pada cabang untuk *(k9)*
  kondisi menjadi (pcur <> *NIL)*

bentuk algoritma versi ke 1 :

```
proc Post_nR_v1(head)
    pcur ← head                                  {operasi a}
    arah ← KEBAWAH                               {inisiasi variabel arah}
    while (pcur <> NIL)                          {belum stop}
    do  if (k1) or (k4)
        then arah ← KEBAWAH
             push(pcur)                          {operasi b}
             pcur ← pcur↑.fs
        else if (k2) or (k5) or (k7)
             then arah ← KEKANAN
                  write (layar) pcur↑.info
                  pcur ← pcur↑.nb                {operasi c}
             else if (k3) or (k6) or (k8)
                  then arah ← KEATAS
                       write (layar) pcur↑.info
                       pcur ← pull()             {operasi d}
                  else pcur ← NIL                {supaya stop}
                  endif
             endif
        endif
    endwhile
endproc
```

- versi lain dapat mengambil bentuk dasar :

```
proc Post_nR_v2(head)
    ..{operasi a}
    arah ← KEBAWAH                          {inisiasi variabel arah}
    while (pcur <> NIL)                     {belum stop}
      do  case of (arah)
            KEBAWAH :
                if (pcur↑.fs<>NIL)
                then arah ← KEBAWAH
                     ..{operasi b}
                else if (pcur↑.fs=NIL) and (pcur↑.nb<>NIL)
                     then arah ← KEKANAN
                          ..{operasi c}
                     else arah ← KEATAS
                          ..{operasi d}
                     endif
                endif
            KEKANAN :
                if (pcur↑.fs<>NIL)
                then arah ← KEBAWAH
                     ..{operasi b}
                else if (pcur↑.fs=NIL) and (pcur↑.nb<>NIL)
                     then arah ← KEKANAN
                          ..{operasi c}
                     else arah ← KEATAS
                          ..{operasi d}
                     endif
                endif
            KEATAS :
                if (pcur=head)
                then pcur ← NIL
                else if (pcur↑.nb<>NIL)
                     then arah ← KEKANAN
                          ..{operasi c}
                     else arah ← KEATAS
                          ..{operasi d}
                     endif
                endif
            endcase
    endwhile
endproc
```

## Algoritma non-rekursif traversing pre-order pada tree non-biner

Merujuk pada tabel ikhtisar ada 5 pilihan kumpulan operasi. Dapat dilakukan pengurangan pilihan menjadi 4 saja, dimana pilihan (e) direkayasa sehingga sama dengan pilihan (d). Caranya adalah sbb :

- Stack diinisiasi dengan mem"push" nilai NIL.
- Pada pilihan (e) dilakukan = operasi pada (d), maka akan di"pull" nilai NIL dari stack.

```
proc Pre_nR(head)
    push(NIL)                                  {inisiasi isi stack dengan nilai NIL}
    pcur ← head                                {inisiasi}
    while (pcur <> NIL)                        {belum stop}
      do write (layar) pcur↑.info
         if (pcur↑.fs<> NIL)                   {punya first-son}
           then if (pcur↑.nb<> NIL)            {first-son ada, next-brother ada}
                  then push(pcur↑.nb)
                  else
                endif
                pcur ← pcur↑.fs
           else if (pcur↑.nb<> NIL)            {first-son tdk-ada, next-brother ada}
                  then pcur ← pcur↑.nb
                  else pcur ← pull()
                endif
         endif
    endwhile
endproc
```

## ALGORITMA TRAVERSING ( rekursif ).

## Algoritma rekursif traversing post-order pada tree non-biner

Post_r adalah prosedur yang menangani traversing post-order, sedangkan Utama adalah prosedur yang memanggil Post_r. Bentuknya sangat ringkas, karena kerumitan penanganan stack diserahkan pada sistem pemrograman yang dipakai. Harap diingat bahwa pemanggilan secara rekursif akan memboroskan pemakaian stack-area pada sistem !

```
proc Utama
    if (head <> NIL)
       then Post_R(head)
    endif
endproc

proc Post_R(pcur)
  Repeat
     if (pcur↑.fs <> NIL)              {sub-tree pertama ada}
       then Post_R(pcur↑.fs)           {kunjungi sub-tree pertama}
     endif
     write(layar) pcur↑.info           {disinilah kunjungan resmi pada node pcur}
     pcur ← pcur↑.nb                   {sub-tree berikut di kanan}
  until (pcur = NIL)                   {selama sub-tree tsb. ada}
endproc
```

## Algoritma rekursif traversing pre-order pada tree non-biner

```
proc Utama
|       Pre_R(head)
endproc

proc Pre_R(pcur)
|       .....                          {disinilah kunjungan resmi pada node pcur}
|       if (pcur↑.fs <> NIL)           {sub-tree pertama ada}
|       | then  Pre_R(pcur↑.fs)        {kunjungi sub-tree pertama}
|       |       px ← pcur↑.nb          {sub-tree berikut di kanan}
|       |       while (px<>NIL)        {selama sub-tree tsb. ada}
|       |       |  do   Pre_R(px)      {kunjungi sub-tree tsb.}
|       |       |       px ← pcur↑.nb  {sub-tree berikut di kanan}
|       |       endwhile
|       | else
|       endif
endproc
```

## ALGORITMA TRAVERSING BINARY TREE.

Dengan asumsi struktur data yang digunakan adalah :

| ↑.lptr | data | ↑.rptr |
|---|---|---|

## Algoritma rekursif traversing post-order pada binary tree

```
Proc PostOrder(P : BinTREE)
begin
      if P ≠ nil then
      begin
            PostOrder(P↑.lptr)
            PostOrder(P↑.rptr)
            write(P↑.data)
      endif
endproc
```

## Algoritma non rekursif traversing Inorder pada tree biner

```
proc Utama
|       InOrder (head)
endproc

proc InOrder (pcur : BinTREE)
|   if (pcur <> NIL)                                  {sub-tree pertama ada}
|   |   then    Repeat
|   |           |   While (pcur <> NIL) do
|   |           |       push(pcur)
|   |           |       pcur ← pcur↑.lptr            {sub-tree berikut di kiri}
|   |           |   EndWhile
|   |           |
|   |           |   pcur ← Pop( )
|   |           |   write(layar) pcur↑.data          { kunjungan resmi pada node pcur}
|   |           |   pcur ← pcur↑.rptr                {sub-tree berikut di kanan}
|   |           Until (pcur = NIL) and (stack = NIL)     {selama sub-tree tsb. ada}
|   endif
endproc
```

## Algoritma non rekursif traversing Level order pada tree biner

```
proc Utama
|       Level_R(head)
endproc

proc Level_R (pcur : BinTREE)
|   if (pcur <> NIL)                                  {sub-tree pertama ada}
|   |   then    write(layar) pcur↑.data              { kunjungan resmi pada node pcur}
|   |           While (pcur <> NIL) or (queue <> NIL) do
|   |           |   If (pcur↑.lptr <> NIL)
|   |           |   then write(layar) pcur↑.lptr↑.data
|   |           |       If (pcur↑.lptr↑.lptr <> NIL) or (pcur↑.lptr↑.rptr <> NIL)
|   |           |       then INSERT(pcur↑.lptr)
|   |           |       Endif
|   |           |   EndIf
|   |           |
|   |           |   If (pcur↑.rptr <> NIL)
|   |           |   then write(layar) pcur↑.rptr↑.data
|   |           |       If (pcur↑.rptr↑.lptr <> NIL) or (pcur↑.rptr↑.rptr <> NIL)
|   |           |       then INSERT(pcur↑.rptr)
|   |           |       Endif
|   |           |   EndIf
|   |           |
|   |           |   If (queue <> NIL)
|   |           |       then pcur ← DELETE( )
|   |           |       else pcur ← NIL
|   |           |   EndIf
|   |           EndWhile
|   endif
endproc
```

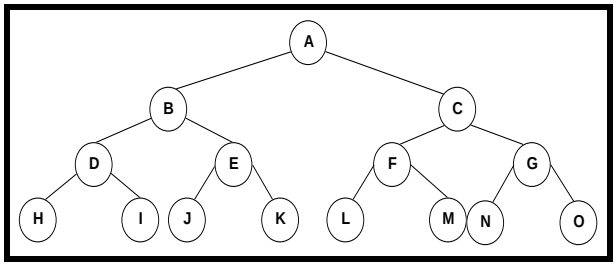
Dengan menggunakan struktur linked list sebagai berikut :

| ↑.Lptr | Info | ↑.Rptr |
|---|---|---|

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 :




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).

# 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 :

| Info | ↑.Parent |
|---|---|
| ↑.Lptr | ↑.Rptr |

Ide pohon threaded di atas menambah alamat pointer sebanyak 50 % (menunjuk Parent), untuk mengatasinya dapat dilakukan :

1. Penambahan dua field *Boolean* pada setiap simpul, yaitu *Lthread* dan *Rthread*, sehingga pointer yang dibutuhkan hanya dua buah

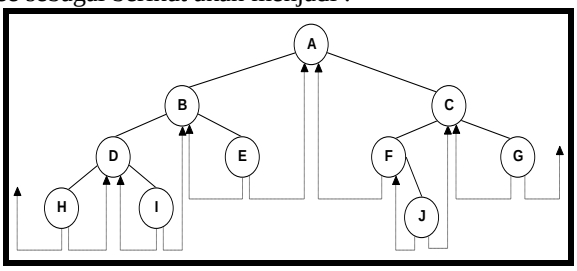
| Lthread | Info | Rthread |
|---|---|---|
| ↑.Lptr | | ↑.Rptr |

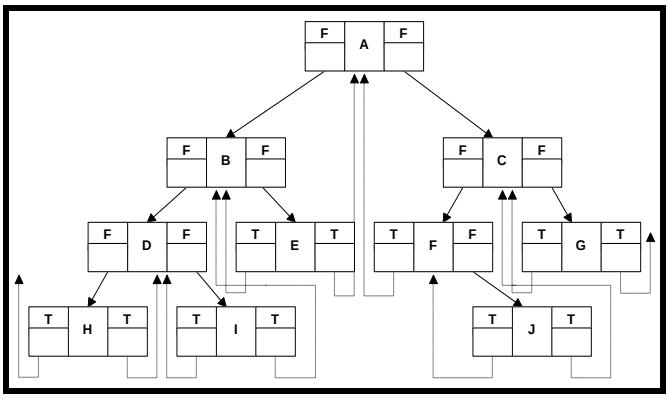
2. *Lthread* dan *Rthread* diisi dengan "TRUE" bila pointer Lptr dan Rptr menunjuk ke simpul *parent,* dan diisi dengan "FALSE" bila Lptr dan Rptr menunjuk ke pencabangannya (anaknya).

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 Pohon Threaded yang terkatung-katung

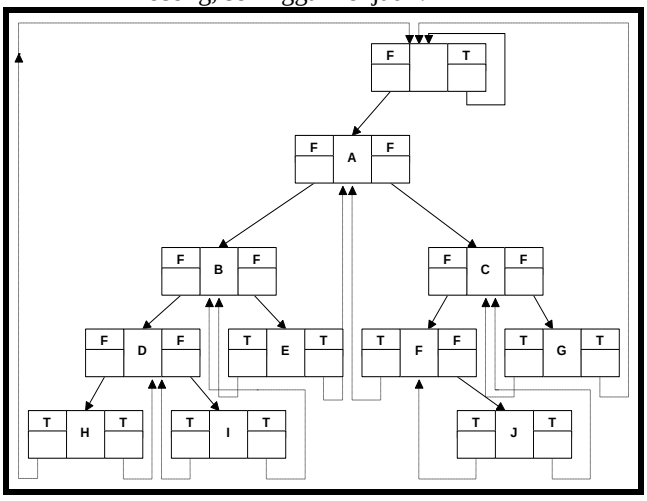
Hal-hal yang perlu diperhatikan dalam penggunaan pohon *Threaded* adalah :

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

<u>Penanganan masalah di atas :</u>

Kasus 1 :

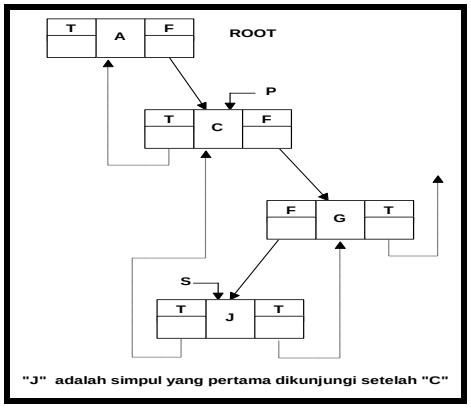
Dibuat simpul HEADER kosong, sehingga menjadi :



Kasus 2 :

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

1. Diketahui simpul p;
2. Jika p ↑.rptr ≠ NIL dan p↑.Rthread = TRUE, maka p↑.rptr adalah penerus simpul p;
3. Jika p↑.rptr ≠ NIL dan p↑.Rthread = FALSE, maka p↑.rptr adalah cabang kanan dari simpul p;
4. 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:




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.