Universitas Esa Unggul Kampus Harapan Indah	**MODUL XIII** **STRUKTUR DATA**

Judul	**POHON BINER DAN POHON PENCARIAN BINER**	
Penyusun	**Distribusi**	**Perkuliahan**
Nixon Erzed	Teknik Informatika Universitas Esa Unggul	Pertemuan – XIII online

Tujuan :

Mahasiswa memahami struktur pohon biner serta pohon pencarian biner, dan dapat membangun pohon pencarian biner serta melakukan operasi-operasi pada pohon pencarian biner

Materi :

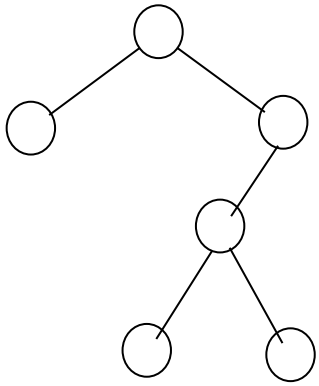
- Binary Tree dan Traverse Binary Tree
- Binary Search Tree
- Operasi-operasi pada Binary Search Tree

Referensi :

1. Struktur Data, Bambang Hariyanto, Infromatika, Bandung, 2002
2. Fundamental of Data Structure, Ellis Horowitz, Pitman International Text, 1978

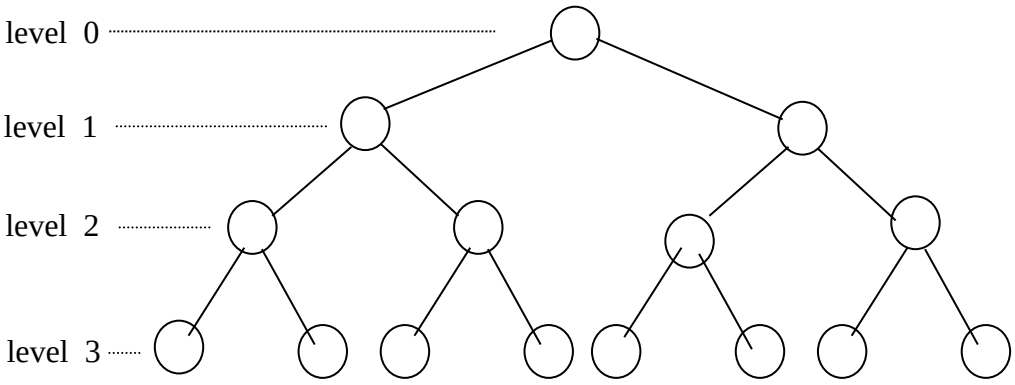
BINARY TREE

Binary Tree, adalah tree dengan node yang memiliki jumlah anak maksimum 2.

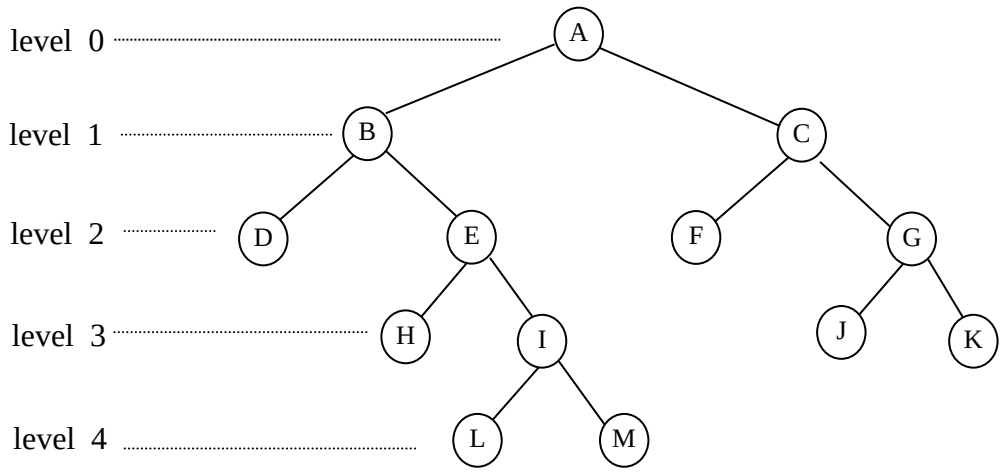


Semua sifat-sifat umum Tree, merupakan sifat Binary Tree. Berikut ini adalah bentuk-bentuk khusus Binary Tree.

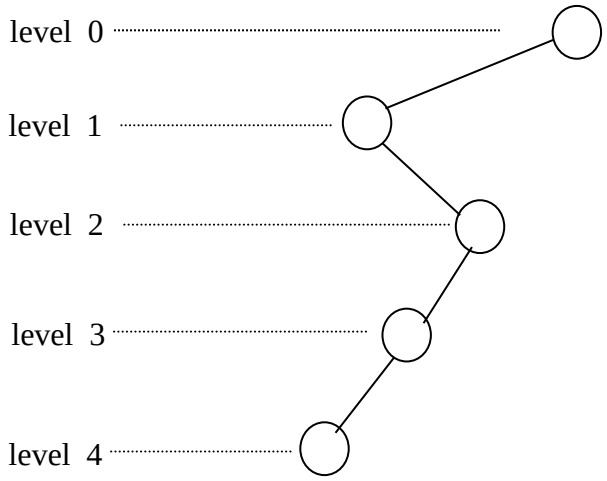
a. **Full Binary Tree** : Binary Tree yang tiap nodenya (kecuali leaf) memiliki dua child/anak dan setiap sub tree harus mempunyai panjang path yang sama (leaf berada pada level yang sama)



b. **Complete Binary Tree** : Binary Tree yang tiap nodenya (kecuali leaf) memiliki dua child/anak, tetapi leaf dapat berada pada level yang berbeda



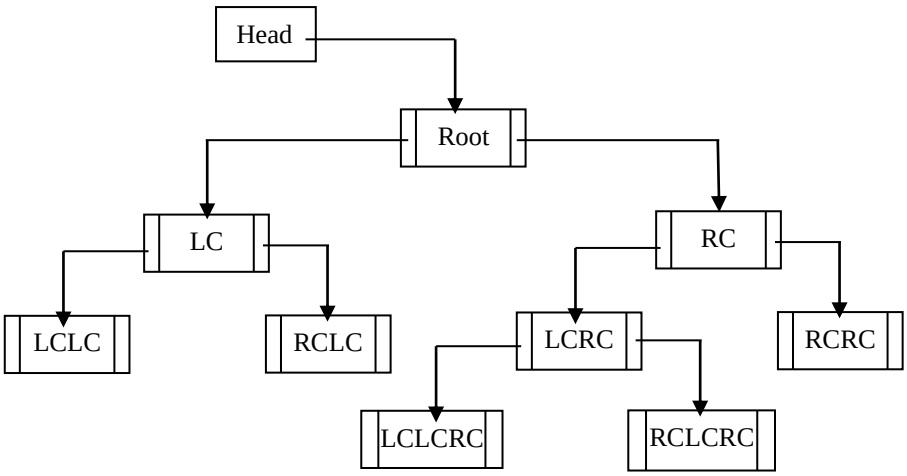
c. **Skewed Binary Tree** : Binary Tree yang tiap nodenya (kecuali leaf) hanya memiliki satu child/anak.



Implementasi Binary Tree dalam program umumnya menggunakan double linked list. Berikut ini adalah contoh deklarasi Linked List untuk merepresentasikan Binary Tree:

```
type
     BTree   = ^simpul
     Node    = record
               Data    : typedata;
               Kiri, Kanan :  Btree;
          end;
var
     head   :   Btree
```

Ilustrasi



LC : Left child (anak kiri)

RC : Right child (anak kanan)

TRAVERSE

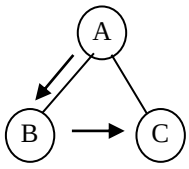
Traverse binary tree, adalah mengunjungi seluruh node-node pada tree, masing-masing satu kali. Hasilnya adalah urutan informasi secara linear yang tersimpan dalam tree. Terdapat tiga cara traverse :

- PreOrder
- InOrder
- PostOrder

Langkah-langkah traverse :

PreOrder

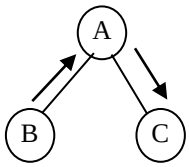
Visit isi node current, kunjungi LeftChild, kunjungi RightChild



Untuk contoh Binary Tree yang diberikan pada Complete Binary Tree, hasil traverse secara Pre Order adalah : A B D E H I L M C F G J K

InOrder

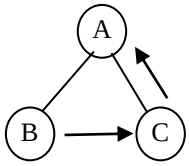
Kunjungi LeftChild, visit isi node current, kunjungi RightChild



Untuk contoh Binary Tree yang diberikan pada Complete Binary Tree, hasil traverse secara In Order adalah : D B H E L I M A F C J G K

Post Order

Kunjungi LeftChild, kunjungi RightChild, visit isi node current



Untuk contoh Binary Tree yang diberikan pada Complete Binary Tree, hasil traverse secara Post Order adalah : D H L M I E B F J K G C A

Operasi-operasi pada binary Tree :

1. Create

   Procedure Create berguna untuk menciptakan BTree baru dan Kosong.
2. Empty

   Function Empty adalah untuk memeriksa apakah Btree masih kosong.
3. Insert

   Adalah prosedur untuk menyisipkan node baru kedalam Tree. Terdapat 3 pilihan yang mungkin : insert sebagai root (jika dan hanya jika Btree Empty), sebagai Left Child/Anak Kiri, dan sebagai Right Child / Anak Kanan
4. Find

   Mencari data tertentu pada Tree, merupakan pemanfaatan Traverse
5. Update

   Mengubah isi dari node yang ditunjuk oleh Current (Current adalah variabel pointer yang dideklarasikan untuk digunakan dalam penelusuran pohon)
6. Retrieve

   Mengetahui isi dari node yang ditunjuk oleh current

7. DeleteSub

   Menghapus sebuah subtree yang ditunjuk oleh current (node beserta seluruh descentdantnya)

8. Clear

   Procedure Clear berguna untuk mengosongkan Btree yang sudah ada. Operasi pengosongan dilakukan dengan cara menghapus seluruh node yang dimulai dari node leaf. Untuk menghindarkan kerumitan logika, maka dilakukan pemanggilan rekursif terhadap Procedure DeleteSub.

BINARY SEARCH TREE

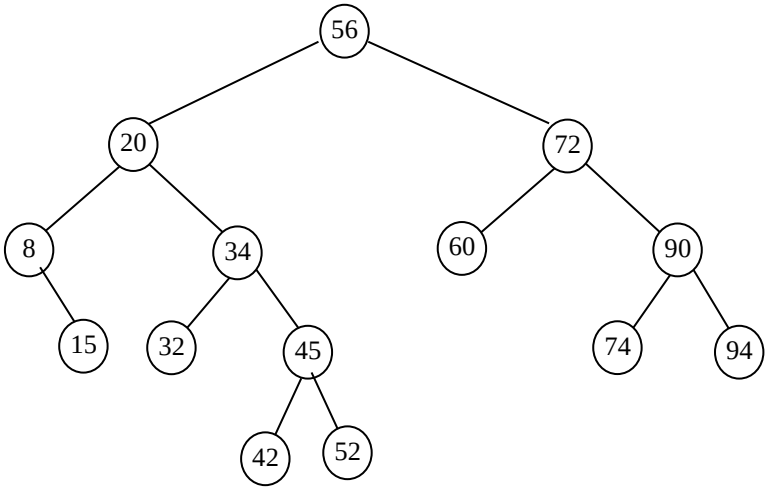
Binary Search Tree (Pohon Pencarian Biner / BST) adalah Binary Tree dengan sifat khusus, yaitu semua Left Child selalu lebih kecil dari Parent dan semua Right Child selalu lebih besar dari Parent, atau LC < Parent < RC.

Binary search tree/BST dibuat untuk mengatasi kelemahan pada binary tree biasa, yaitu kesulitan dalam searching /pencarian node terntentu dalam suatu binary tree.

Jika dimiliki data sebagai berikut :

56 20 72 34 90 74 32 45 42 94 8 52 15 60

dapat dibentuk BST sesuai dengan urutan kedatangan data sebagai berikut :



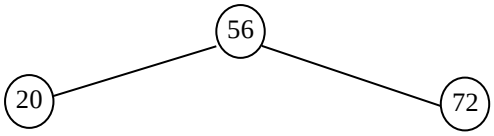
Untuk membangun BST, secara bertahap dilakukan operasi insert sesuai dengan kedatangan data dengan memperhatikan sifat khas BST yaitu : LC < Parent < RC.

Langkah-langkah pembangunan BST untuk data pada contoh diatas adalah sebagai berikut :

Insert (56) sebagai data pertama --› root

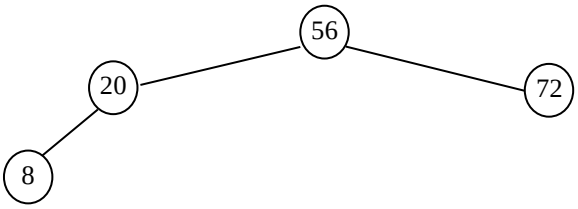
Insert (20) : 20 < 56 --› 20 menjadi LC dari 56

Insert (72) : 72 > 56 --› 72 menjadi RC dari 56



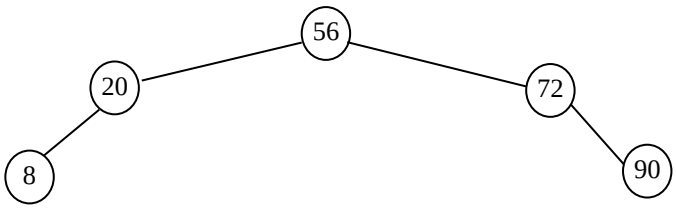
Insert (34) : 34 < 56 --› 34 menjadi LC dari 56

LC sudah berisi 20 : 34 > 20 --› 34 menjadi RC dari 20



Insert (90) : 90 > 56 --› 90 menjadi RC dari 56

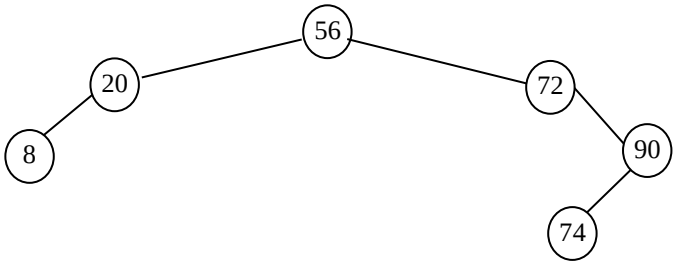
RC sudah berisi 72 : 90 > 72 --› 90 menjadi RC dari 72



Insert (74) : 74 > 56 --› 74 menjadi RC dari 56

RC sudah berisi 72 : 74 > 72 --› 74 menjadi RC dari 72

RC sudah berisi 90 : 74 < 90 --› 74 menjadi LC dari 90

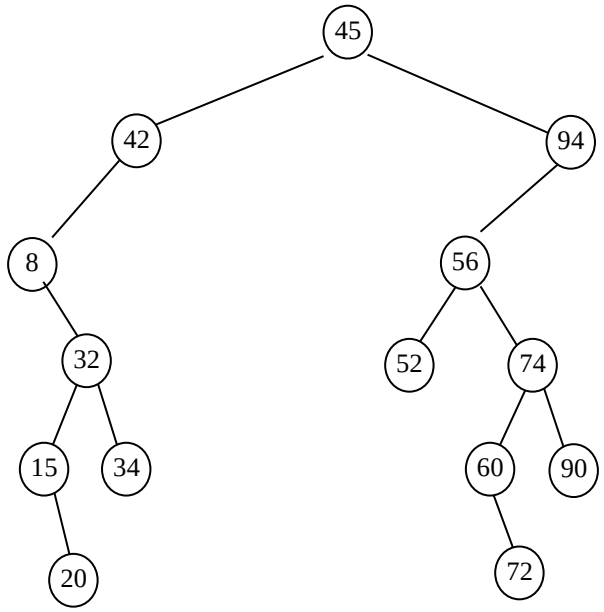


Demikian seterusnya secara berturut-turut akan diinsert 32, 45, 42, 94, 8, 52, 15, 60 sehingga dihasilkan BST.

Bagaimana jika urutan kedatangan data-data tersebut berbeda (datanya sama) ? misalnya sebagai berikut :

45 94 42 8 56 74 32 52 15 34 90 60 20 72

Akan dihasilkan BST sebagai berikut :



Dari contoh tersebut, diketahui bahwa BST yang dihasilkan juga dipengaruhi oleh urutan kedatangan data.

Deklarasi Linked List untuk implementasi BST

```
type
     BTree  =  ^simpul
     Node   =  record
               Data   :  typedata;
               Kiri, Kanan :  Btree;
          end;
var
     root, current  :  Btree
```

Pencarian data dalam BST.

Operasi pencarian data pada BST mengikuti pola/aturan penempatan data pada tree tersebut.

Misalkan ingin dicari apakah pada BST terdapat data X?

Baca Node(root)

If X = Node(root) then data ditemukan

else If X < Node(root) then cari ke Sub BST Kiri

else cari ke Sub BST Kanan

Untuk menghindari kerumitan algoritma, implementasi Pencarian/Searching pada BST menggunakan algoritma rekursif.

```
Procedure Search (var Scurrent : Btree;  cari : typedata );
begin
    if cari < Scurrent^.Data
    then
      if Scurrent^.kiri <> nil
        then   Search ( Scurrent^.kiri , cari )
        else   write ('data tidak ditemukan')
      { end-kiri }
    else
      if cari > Scurrent^.Data
      then
          if Scurrent^.kanan <> nil
            then   Search ( Scurrent^.kanan , cari )
            else   write ('data tidak ditemukan')
        { end-kanan }
      else
          write ('Data ditemukan')
    {end}
  {end}
end;
```

Insert, Update dan Delete pada BST :

BST memiliki sifat khas yaitu : LC < Parent < RC. Sehingga dalam melakukan operasi Insert, Update, dan Delete harus diperhatikan agar hasil akhir operasi tetap merupakan sebuah BST.

Pada operasi Insert, maka mesti ditemukan terlebih dahulu posisi yang sesuai bagi data baru. Sedangkan operasi Update dan Delete, akan memepengaruhi struktur pohon. Karena dapat menyebabkan pohon tersebut bukan lagi BST, sehingga diperlukan perubahan struktur pohon.

Insert

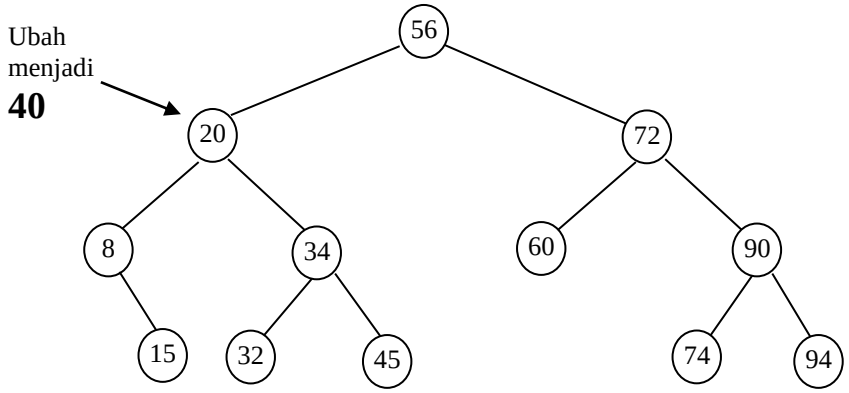
Seperti sudah digambarkan pada langkah-langkah pembangunan BST, penyisipan sebuah node baru, didahului dengan operasi pencarian posisi yang sesuai. Dalam hal ini node baru tersebut akan menjadi daun/leaf.

```
Procedure  Insert (databaru : typedata );
 begin
    new(BTree)'
    B Tree^.data    :=  databaru;
    B Tree^.kiri    :=  nil
    B Tree^.kanan   :=  nil

    If  root  =  nil
     then
          root :=  BTree
     else
       begin
          Current  := root;
          Cariposisi (Current, databaru)
          If  databaru  <=  current^.data
            then
                current^.kiri  :=   BTree
           else
                current^.kanan  :=  BTree
      end;
 end;

Procedure  Cariposisi (var  Pcurrent : BTree ;   dtBaru : tipedata );
 begin
     if   dtBaru  <=  Pcurrent^.Data
     then
       if  Pcurrent^.kiri <>  nil
           then    Cariposisi (Pcurrent^.kiri , dtBaru )
      { end-kiri }
     else
       if   dtBaru  > Pcurrent^.Data
        then
            if  Pcurrent^.kanan <>  nil   then  Cariposisi (Pcurrent^.kanan , dtBaru )
       { end-kanan }
   {end}
end;
```

Update

Operasi update sebuah node dapat menyebabkan tree bukan lagi sebuah BST.

Perhatikan BST berikut ini :



Misalnya anak kiri dari Root (yaitu node dengan nilai 20), diubah harganya menjadi 40, maka tree yang dihasilkan bukan lagi sebuah Binary Search Tree (BST) karena node (40) > anak kanan (34).

Bila akibat operasi update tree tsb. Bukan lagi BST, maka harus dilakukan perubahan (restrukturisasi) pada Tree sehingga tetap menjadi BST.

Aturannya :

- Jika update menyebabkan data menjadi lebih kecil dari anak kiri maka lakukan pertukaran dengan anak kiri, dan periksa kembali apakah pertukaran sudah membentuk BST, jika belum ulangi langkah restrukturisasi.
- Jika update menyebabkan data menjadi lebih besar dari anak kanan maka lakukan pertukaran dengan anak kanan, dan periksa kembali apakah pertukaran sudah membentuk BST, jika belum ulangi langkah restrukturisasi.

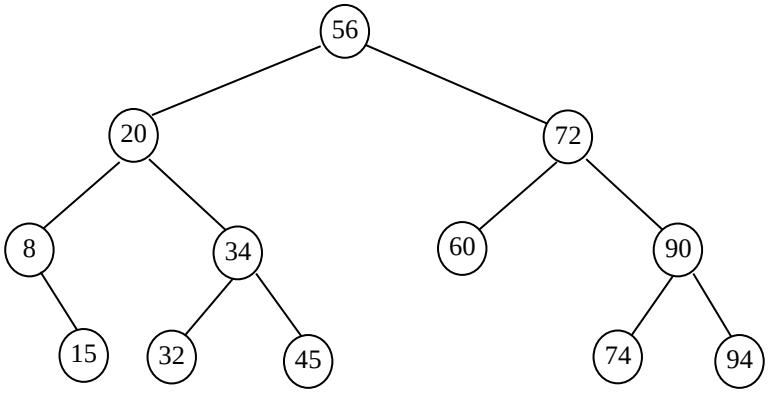
Untuk contoh pengubahan pohon diatas, update menyebabkan data menjadi lebih besar dari anak kanan, sehingga node dipertukarkan. Selanjutnya diperiksa, dan ternyata tree merupakan sebuah BST.

Andaikan perubahan adalah menjadi 50, maka hasil pertukaran dengan anak kanan belum menghasilkan sebuah BST (sebab 50 > 45), dan ulangi restrukturisasi dengan menukarkan node 50 dengan anak kanannya (node 45)

Delete

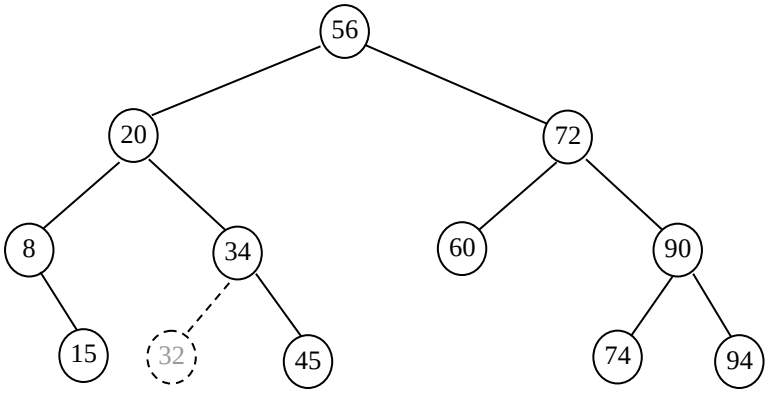
Operasi delete memiliki 3 kemungkinan :

- Delete terhadap node tanpa anak/child (leaf/daun) : node dapat langsung dihapus
- Delete terhadap node dengan satu anak/child : maka node anak akan menggantikan posisinya.
- Delete terhadap node dengan dua anak/child : maka node akan digantikan oleh node paling kiri dari Sub Tree Kanan atau dapat juga digantikan oleh anak paling kanan dari Sub Tree Kiri.

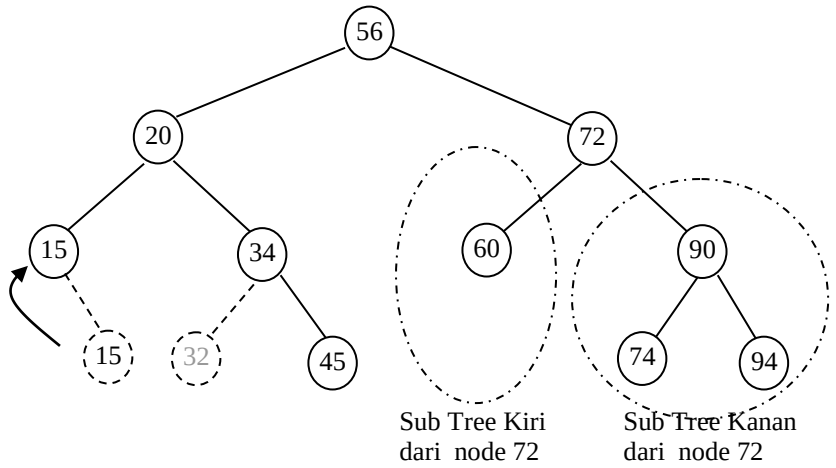


Misalnya ingin dihapus

1. Node (32) : dapat langsung dihapus sehingga akan dihasilkan tree sbb.



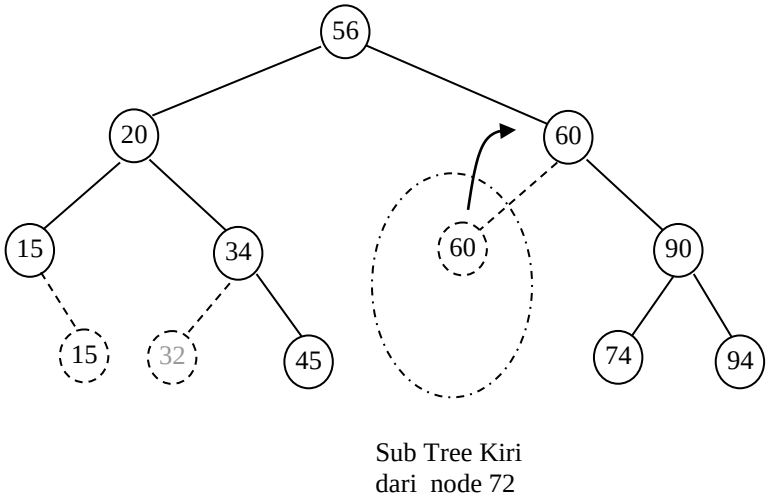
2. Node (8) → node dengan satu child :



3. Node (72) → node dengan 2 child

   Node akan digantikan oleh anak paling kanan dari Sub Tree Kiri → node 60

   Atau anak paling kiri dari Sub Tree Kanan → node 74



Atau