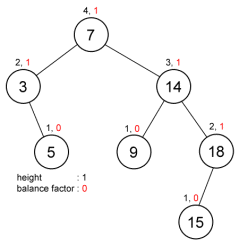
**Struktur Data (AVL, 2-3, RBT, Heap, Deap, Leftist)**

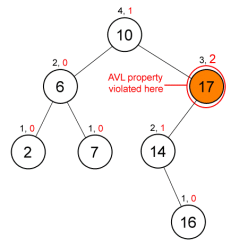
**AVL Tree:**

AVL adalah balanced binary search tree dimana ia memiliki perbedaan jumlah node pada subtree kiri dan subtree kanannya maksimal 1 (atau dapat dikatakan antara tingginya sama atau selisih satu).

Berikut gambarannya :



AVL Tree, karena factor tertingginya 1



Not AVL Tree, karena  balance factor tertingginya 2, sedangkan syarat AVL adalah selisihnya maksimal 1  
Cara menentukan Height dan Balance Factor :Note :

Height :  
– Jika node (root) tidak memiliki subtree heightnya = 0  
– Jika node adalah leaf, height =  1  
– Jika internal node, maka height =  height tertinggi dari anak + 1

Balance Factor :  
-selisih height antara anak kiri dan kanan, jika tidak memiliki anak, dianggap 0.

**AVL Tree Operations : Insertion**

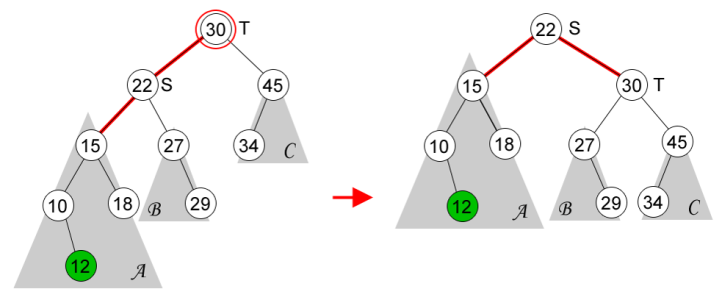
Insert suatu node pada AVL sama halnya pada insert node pada binary search tree, dimana node baru diposisikan sebagai leaf. Setelah memasukkan node baru, maka harus dilakukan penyeimbangan kembali pada path dari node yang baru di insert atau path terdalam. Namun biasanya, path terdalam adalah path dari node yang baru saja di insert.

Ada 4 kasus yang biasanya terjadi saat operasi insert dilakukan, yaitu :  
anggap T adalah node yang harus diseimbangkan kembali  
– Kasus 1 : node terdalam terletak pada subtree kiri dari anak kiri T (left-left)  
– Kasus 2 : node terdalam terletak pada subtree kanan dari anak kanan T (right-right)  
– Kasus 3 : node terdalam terletak pada subtree kanan dari anak kiri T (right-left)  
– Kasus 4 : node terdalam terletak pada subtree kiri dari anak kanan T (left-right)

Ke-4 kasus tersebut dapat diselesaikan dengan melakukan rotasi  
– Kasus 1 dan 2 dengan single rotation  
– Kasus 3 dan 4 dengan double rotation

**Single Rotation**

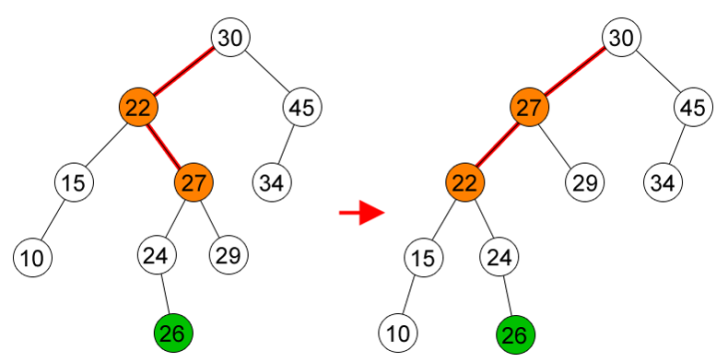
Single rotasi (rotasi 1x) dilakukan apabila searah, left-left atau right-right  
Gambaran Single Rotation :



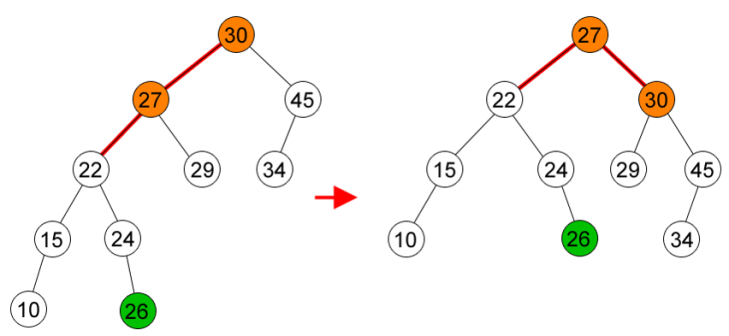
Single Rotation  
pada contoh diatas, left-left karena dari 30 ke 22 ke kiri, dan dari 22 ke 15 ke kiri juga

**Double Rotation**

Double rotasi (rotasi 2x) dilakukan apabila searah, left-right atau right-left  
Gambaran Double Rotation :



Step 1 (Rotasi pertama)  
kasus diatas adalah left-right

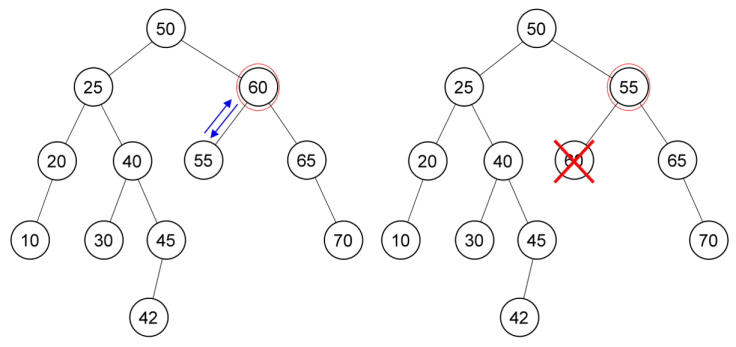


Step 2 (Rotasi kedua)

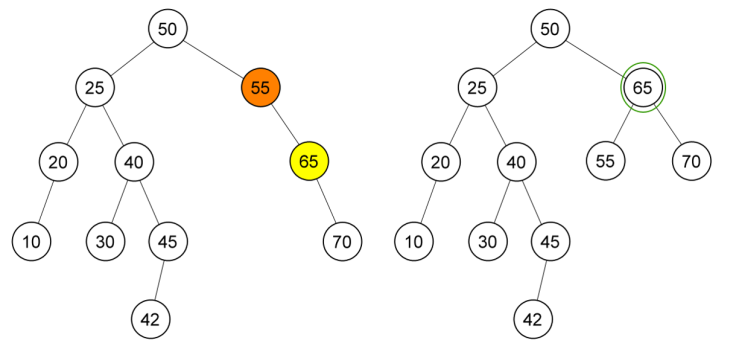
kasus diatas, left-left

**AVL Tree Operations : Deletion**

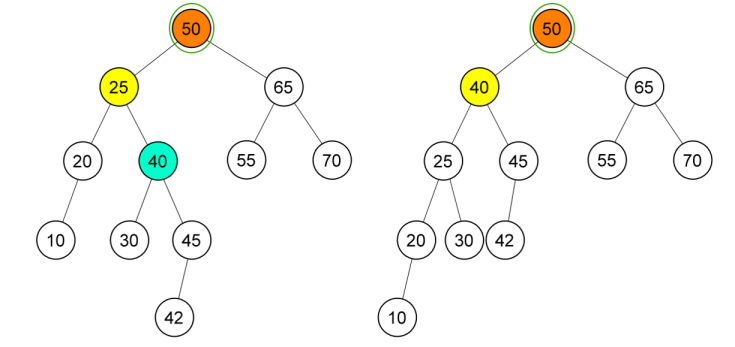
Operasi penghapusan node sama seperti pada Binary Search Tree, yaitu node yang dihapus digantikan oleh node terbesar pada subtree kiri atau node terkecil pada subtree kanan. Jika yang dihapus adalah leaf, maka langsung hapus saja. Namun jika node yang dihapus memiliki child maka childnya yang menggantikannya. Namun setelah operasi penghapusan dilakukan, cek kembali apakah tree sudah seimbang atau belum, jika belum maka harus diseimbangkan kembali. Cara menyeimbangkannya pun sama seperti insertion.



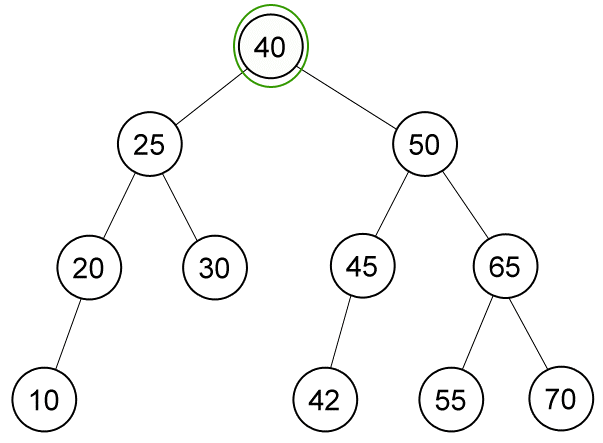
delete node 60



node 55 tidak seimbang, karena anak kiri 0 dan anak kanan 2, selisih 2.  
diseimbangkan dengan single rotation (left-left) karena 55 ke 65 kiri dan 65 ke 70 kiri  
akan tetapi, node 50 menjadi tidak seimbang, di subtree kiri 4 dan subtree  kanan 2



diseimbangkan dengan double rotation (left-right) karena dari 50 ke 25 kiri dan 25 ke 40 kanan

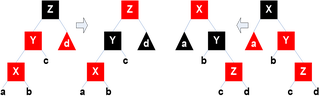


AVL yang sudah balance.

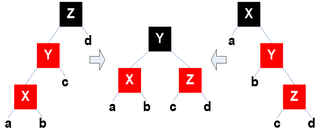
**RBT Tree:**

**Red Black Tree merupakan Binary Search Tree yang terdiri dari 2 warna. Sebuah BST dikatakan RBT apabila memenuhi kondisi berikut :**

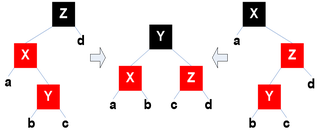
1. setiap node memiliki warna, baik merah atau hitam  
2. root adalah hitam secara default / otomatis  
3. semua external node adalah hitam  
4. jika sebuah node berwarna merah, maka kedua anaknya berwarna hitam. itu berarti tidak ada child merah yang memiliki parent merah  
5. setiap node ke external node keturunan memiliki jumlah node hitam yang samaTerdapat 2 operasi pada RBT yaitu Insert dan Delete– Insert  
insert pada RBT sama seperti pada konsep BST dan setiap node baru berwarna merah. jika parent berwarna hitam maka tidak ada violation dan tinggal insert saja node tersebut, tetapi jika parent berwarna merah maka violation terjadi (sesuai aturan nomor 4).  
untuk memperbaiki violation tersebut maka:  
– misalkan:  
Q = node baru  
P = parentnya  
S = sibling P (parent Q)  
jika parent tidak memiliki sibling, S  = hitam (external node)  
jika S = merah, ganti P dan S menjadi hitam dan parent P menjadi merah  
jika S hitam maka lakukan rotasi seperti rotasi pada AVL, lalu ubah pivot rotasi terakhir menjadi hitam dan anaknya menjadi merah.contoh 1 (node baru X): recolor

[](https://sikisik.files.wordpress.com/2015/06/8e995-rbbt2.png)

contoh 2 (node baru X): single rotate

[](https://sikisik.files.wordpress.com/2015/06/1909f-rbt3.png)

contoh 3 (node baru Y): double rotate

[](https://sikisik.files.wordpress.com/2015/06/41e8b-rbt4.png)

**– Delete**

delete pada RBT juga sama seperti BST, hanya pada RBT ada beberapa kondisi mengenai warna, yaitu:

   – misalkan:

M = target delete

C = pengganti

jika M = merah, C = merah langsung ganti

jika M = merah, C = hitam langsung ganti

jika M = hitam, C = merah langsung ganti dan recolor C = hitam

jika M = hitam, C = hitam akan menciptakan token double black di posisi C

C = N

S = sibling

Sl = sibling left

Sr = sibling right

P = parent

kondisi jika ada token double black:

– kondisi 1 jika S = merah, SL & SR = hitam

maka reverse color P dan S dan rotate di P , token black tetap ada di N. cek kondisi lagi

– kondisi 2 jika s = hitam, SL & SR = hitam

maka recolor S = red , token black pindah ke P

jika warna P = merah recolor jadi hitam, jika P = hitam maka terjadi double black lagi

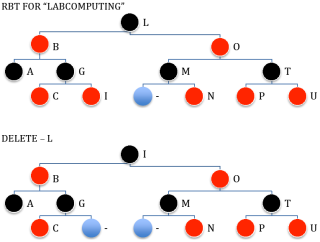
– kondisi 3 jika S = hitam , SL / SR = merah

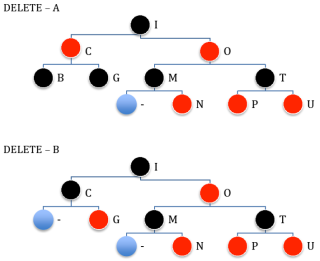
maka rotate di parent (single/double), parent baru mengikuti warna parent lama dan kedua anaknya jadi hitam , lalu token black hilang

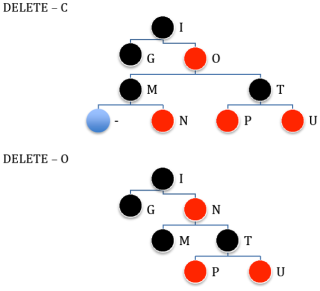
nb : jika token double black berada pada node merah, maka node tsb menjadi hitam dan

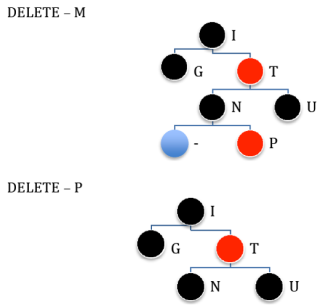
token double black hilang.

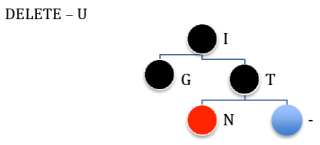
contoh delete:

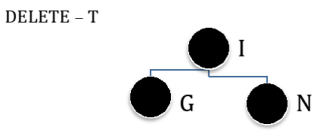
[](https://sikisik.files.wordpress.com/2015/06/c956a-delete1.png)

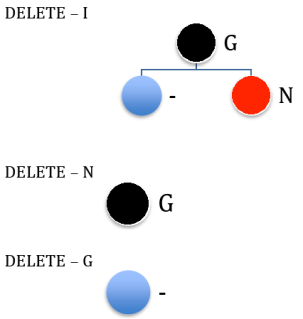
[](https://sikisik.files.wordpress.com/2015/06/1c49b-delete2.png)

[](https://sikisik.files.wordpress.com/2015/06/23249-delete3.png)

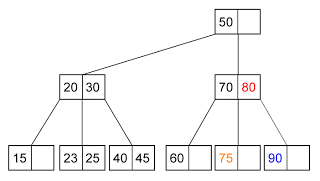
[](https://sikisik.files.wordpress.com/2015/06/f3421-delete4.png)

[](https://sikisik.files.wordpress.com/2015/06/d163f-delete5.png)

[](https://sikisik.files.wordpress.com/2015/06/c7372-delete6.png)

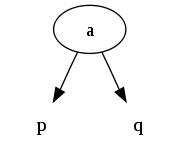
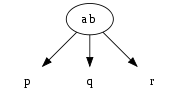
[](https://sikisik.files.wordpress.com/2015/06/6f128-delete7.png)

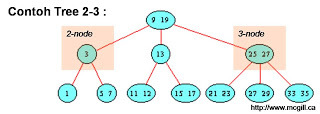
**2-3 Tree**



**Definisi:**

dalam bidang *computer science*, 2-3 Tree adalah salah satu tipe struktur data, dimana setiap node dengan anaknya, memiliki 2 children dan 1 elemen data (2 node) atau 3 children dan 2 elemen data (3 node). perlu diketahui, bahwa 2-3 Tree bukan Binary Tree.

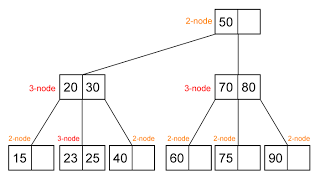
 



**Sifat-sifat 2-3 Tree:**

* Setiapnon-leaf terdapat 2-node atau 3- Sebuah 2-node berisi satu item data dan memiliki dua anak/children (anak kiri dan anak tengah). Sebuah 3-node berisi dua item data dan memiliki 3anak/children (anak kiri, tengah, dan kanan).
* Semua leaf berada di level yang sama (level bawah/bottom level).
* Semua data yang disimpan akan diurutkan:
  1. Misalkan A adalah data yang tersimpan di 2-node. Subtree kiri harus berisi data yang nilainya lebih kecil dari A. Subtree tengah harus berisi data yang nilainya lebih besar dari A.
  2. Misalkan A dan B adalah data yang tersimpan di 3-node. Subtree kiri harus berisi data yang nilainya kurang dari A. Subtree tengah berisi nilai antara A dan B, dan subtree kanan berisi nilai yang lebih dari B.

contoh 2-3 Tree :

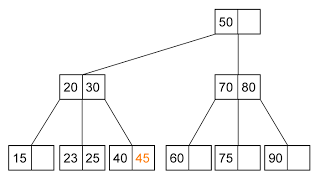


**Insertion pada 2-3 Tree:**

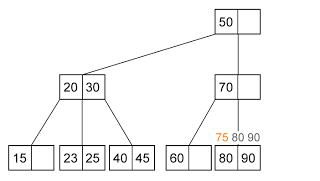
**ketika kita ingin memasukkan data baru ke 2-3 Tree, maka kita harus melakukan beberapa tahapan :**

1. Misalkan kita memasukkan data baru.
2. Pertama-tama, kita harus mencari dimana data tersebut harus diletakkan di 2-3 Tree menggunakan algoritma pencarian, maka akan terdapat 1 leaf yang tepat untuk diletakkan.
3. Jika leaf yang dituju adalah 2-node, maka kita cukup letakkan data baru tersebut, sehingga menjadi 3-node).
4. Sedangkan, jika leaf yang dituju sudah 3-node. Maka simulasikan dengan memasukkan data baru diantara data A dan B (A dan B adalah dua data yang terdapat di 3-node tersebut). nilai data yang berada di tengah-tengah antara nilai A, B, dan data baru akan naik ke parent dan membagi 2 sisanya menjadi 2-node. secara rekursif memperbaiki pada parent-nya.

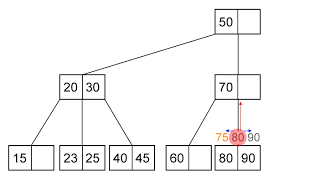
Contoh Insertion pada 2-3 Tree :



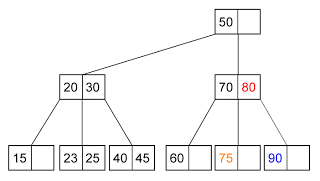
ketika insert(45) dilakukan. leaf yang dituju adalah 2-node. sehingga, letakkan saja nilai 45 tersebut.  
contoh lainnya yaitu memasukkan di 3-node leaf.



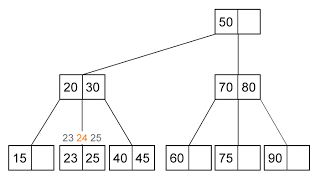
ketika kita insert(75), maka kita akan membandingkan nilai 75-80-90. Nilai yang berada di tengah antara ketiga nilai tersebut akan didorong naik menuju parentnya. seperti gambar di bawah ini



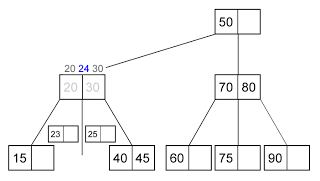
sehingga akan menjadi seperti gambar tree di bawah ini



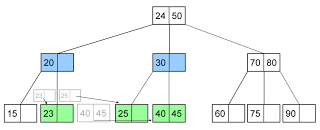
2-3 Tree setelah memasukkan nilai 75.  
Insert(24)



nilai tengah (24) akan didorong ke parent diatasnya.



ternyata parent juga 3-node. dorong nilai tengah diantara 3 nilai tersebut menuju parent di atasnya lagi.



sehigga 2-3 Tree setelah di insert(24), maka akan tampak seperti gambar di atas.

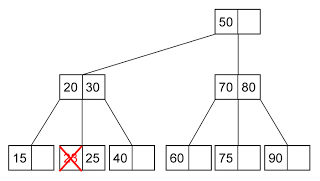
**Deletion pada 2-3 Tree:**

beberapa tahapan deletion pada 2-3 Tree :

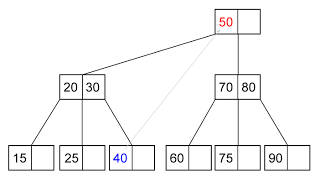
1. Kita masukkan kunci untuk mencari data yang kita ingin hapus dari 2-3 Tree.
2. Seperti di Binary Search Tree (BST), pertama-tama kita harus mencari leaf yang terdapat data yang kita cari.
3. Jika leaf-nya sudah 3-node, maka kita tinggal menghapus saja data tersebut dan menjadikannya 2-node.
4. Jika leaf-nya adalah 2-node :

* Jika parent-nya 3-node, maka ambil 1 nilai dari 3-node tersebut. Jika Sibling/Saudaranya adalah 3-node, maka push nilai tersebut dari sibling ke parent untuk membuat parent menjadi 3-node lagi. Jika sibling adalah 2-node, buat parent menjadi 2-node dan gabungkan node dengan sibling-nya.
* Jika parent-nya 2-node, Jika sibling-nya adalah 3-node, maka ambil satu nilai dari parent dan push satu nilai dari sibling ke parent-nya sendiri. Hal lainnya yaitu mengabunggkan parent dengan sibling.

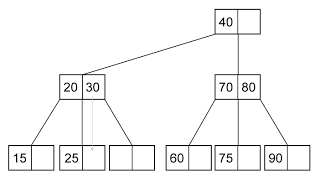
Contoh Deletion pada 2-3 Tree :



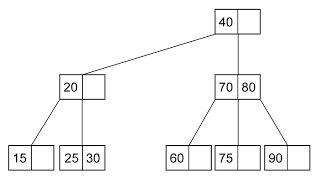
Delete(23), angka 23 terdapat pada leaf, dan leaf tersebut adalah 3-node. Jadi, kita tinggal menghapus saja angka 23.  
Contoh Deletion lainnya :



ketika delete(50), maka ambil nilai 40 pada leaf paling bawah , dan gantikan dengan 50 pada parent atas, lalu delete/hapus nilai 40 pada leaf paling bawah. sehingga seperti gambar di bawah ini.



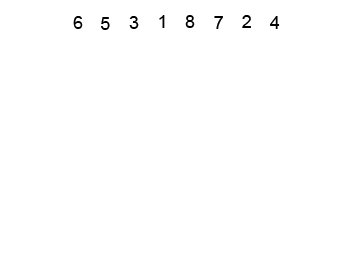
parent-nya adalah 3-node, sedangkan sibling-nya merupakan 2-node. Jadi, gabungkan (25) dan (30). Sehingga 2-3 Tree nya adalah pada gambar di bawah ini.



2-3 Tree setelah delete(50).

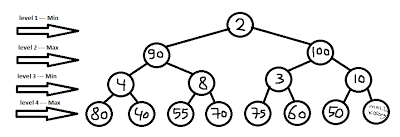
**Heap:**

Kali ini kita akan membahas mengenai struktur data heap, dimana heapsort memiliki struktur data sendiri untuk merepresentasikan tree. Struktur data ini biasa diberi nama *heapify.* Ok dari pada berlama lama anda dapat langsung perhatikan gambar dibawah ini :



**Min-Max Heap**

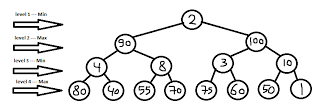
Min-Max Heap adalah kombinasi dari min heap dan max heap di mana masing-masing level akan berganti-ganti antara min heap dan max heap. Heap jenis ini berguna untuk langsung menemukan nilai min dan max dalam 1 heap saja.  
Ini contoh min-max heap:



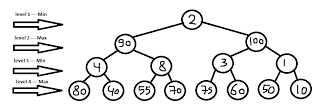
**Insertion:**

Saat sebuah data baru dimasukkan, proses yang akan dilakukan dipengaruhi oleh lokasi data baru tersebut.  
Jika data baru berada di level min, maka bandingkan dengan parentnya. Jika parent < newnode maka swap dan upheapmax dari parentnya. Jika tidak, upheapmin dari posisi newnode.  
Insertion di level max berlaku hal yang serupa, namun berbeda tanda dan berbeda jenis upheap. Selebihnya, sama.  
Upheap dalam min-max heap berbeda dengan upheap biasa. Upheap dalam kasus ini membandingkan dengan grandparent; bukan dengan parent.

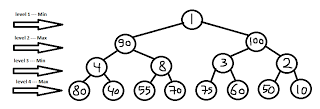
Berikut contoh penerapannya:



angka 1 dimasukkan sebagai newnode. Karena berada di level max, maka jika 1<10, maka swap.

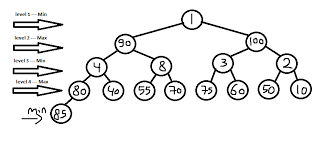


1 dan 10 diswap. Kemudian, dari posisi 1, di-upheapmin. karena 1<2, maka 1 dan 2 diswap.

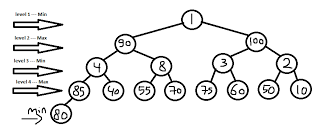


hasil akhirnya pun menjadi begini.

Berikut contoh insertion bila dalam level min:



newnode 85 masuk di level min. Maka, 85 dibandingkan dengan parentnya (80).



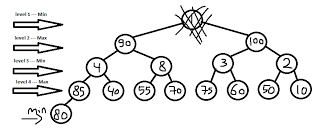
ternyata 85>80, maka swap. Kemudian, 85 di-upheapmax. Tapi, karena 85<90, maka 85 tetap di posisinya.

**Deletion:**

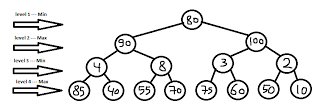
Deletion dalam min-max heap ada 2: delete min dan delete max. Jika delete min, maka root langsung dihapus. Jika delete max, maka angka max dari children root akan dihapus.

Berikut penerapannya.

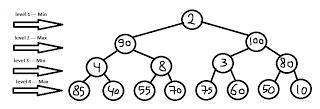
1. Delete min



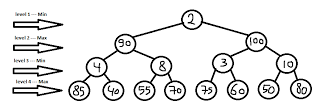
Wolverine datang dan mencakar node 1. 1 pun hilang entah ke mana. Untuk semua children dan grandchildren nya dia meninggalkan pesan agar yang terkecil akan menggantikan dia.



80 (node terakhir), sebagai pengacara naik menggantikan 1 untuk sementara. Dia meneruskan wasiat si “1” untuk mencari anak atau cucu terkecil dan lebih kecil dari dirinya (80).



2 pun terpilih untuk menggantikan 1. Maka, 2 swap dengan 80. 80 pun mengecek kembali bilamana ada children/grandchildren dari 2 yang paling kecil dan lebih kecil dari dirinya.



10 terpilih menggantikan 80. Dan demikianlah berakhir proses downheap dari delete min.

**2. delete max**

mirip dengan delete min, namun semua dimulai dari level 2. Karena malas edit di paint lagi, maka sesi ini sengaja tanpa gambar. Bayangkanlah sendiri.

saat wolverine datang dan membunuh 1, ternyata cyclops datang dan ingin membunuh anak 1 yang paling besar. 100 pun dibunuhnya. Ternyata, 100 sudah meninggalkan wasiat agar anak dan cucu nya yang terbesar naik untuk menggantikan dia.

Kembali, 80 sebagai pengacara (kebetulan saja dia node paling akhir lagi), naik menggantikan 100 untuk sementara. Dia meneruskan wasiat dari almarhum 100. Anak/cucu nya yang paling besar adalah 75. Namun, karena 80 merasa 75 lebih kecil dari dia, maka 75 tidak pantas untuk menggantikan posisinya. Maka, tidak ada swap yang terjadi.

Demikianlah akhirnya 80 menjadi pewaris tahta yang ditinggalkan 100. Anak dan cucunya dibiarkan melarat.

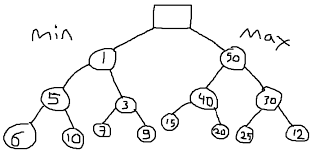
**Deap**

Deap seakan-akan mirip dengan heap. Namun, pada kenyataannya, Deap lebih mirip dengan Min-Max Heap. Deap adalah Min-Max heap yang lebih sederhana. Bila min-max heap, min dan max nya selang-seling atas-bawah, maka dalam deap, min dan max nya ada di kiri dan di kanan.

Beberapa aturan utama mengenai Deap adalah:  
1. root selalu NULL  
2. left subtree dari root adalah min-heap  
3. right subtree dari root adalah max-heap  
4. hanya itu saja

sesuatu yang unik dari deap adalah sesuatu yang bernama “partner”. Partner adalah pasangan dari sebuah node dari deap.  Di bawah akan dijelaskan dengan lebih detail mengenai partner.

Berikut adalah contoh dari Deap:



**Partner:**

Partner adalah node yang berada di posisi yang sama bila subtree min-heap dan max-heap ditumpuk menjadi 1. misalkan, node 1 partnernya 50, node 9 partnernya 12, dst.  
rumus untuk mencari lokasi partnernya adalah:

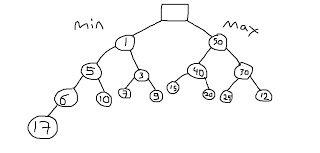
min-partner = x-2^((log x) -1)  
max-partner = x+2^((log x)-1)

bila lokasi partner ternyata kosong, maka partnernya menjadi parent dari posisi partner yang seharusnya.  
Supaya tidak ada di awang-awang, berikut contoh penggunaannya. Kita langsung masuk ke insertion.

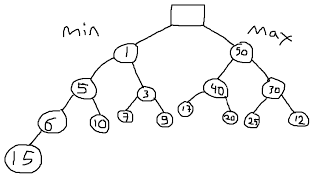
INSERTION:

Saat sebuah data baru dimasukkan, maka akan langsung dicek dengan partnernya. Bila posisinya tertukar, maka swap, kemudian di-upheap. Jika tidak, maka langsung saja di-upheap.

berikut contohnya:



Node 17 masuk. posisinya berada di dalam min heap. Maka, nodenya langsung dicek dengan partnernya. Partner 17 harusnya berada di index ke:  
16+((log 16)-1) = 24.  
Tapi, karena index 24 masih kosong, maka partner 17 adalah parent dari index 24, yaitu 15.

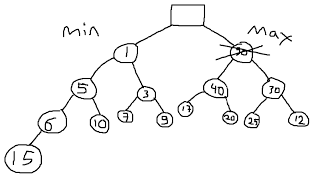


Karena 17>15, maka swap. Kemudian, 17 di-upheap. Karena 17<40, maka tidak ada swap yang terjadi.

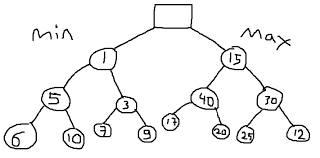
DELETION

Deletion dalam deap ada 2: delete min dan delete max. Jika delete min, maka yang dihapus adalah root dari min-heap. Bila delete max, maka yang dihapus adalah root dari max-heap.  
Setelah dihapus, seperti biasa, elemen terakhir dari heap menggantikan posisi dari yang terhapus, kemudian di-downheap. Bedanya dengan heap biasa adalah setelah di-downheap, maka akan dilakukan pengecekan terhadap partner.

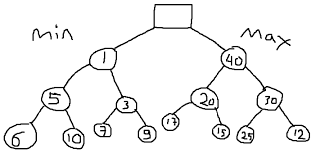
Berikut contohnya:  
1. delete max  
bosen aja delete min yang duluan. Jadi, yang kali ini max yang duluan.



saat 50 dihapus, maka 15 menggantikan posisi 50.



kemudian, 15 di-downheap biasa.



setelah di-downheap, 15 dicek dengan partnernya: 10. Bila 15<10, maka swap. Ternyata tidak. Jadi, selesai sudah proses deletion.

2. delete min  
berhubung malas untuk menggambar lagi, dan sejujurnya, delete min konsepnya sama dengan delete max, maka saya hanya bisa berkata bahwa delete min dengan delete max hanya berbeda tanda saja. Tidak lebih, tidak kurang.

**Leftist Tree**

Leftist Tree adalah varian dari heap yang dapat menemukan elemen terkecil atau terbesar pada tree.  
Leftist Tree memudahkan dalam penggabungan dua tree dan mudah diimplementasikan dalam penggunaan linked list.  
Binary tree dikatakan leftist tree jika pada setiap simpul internal nilai s anak kiri lebih besar dari atau sama dengan nilai s dari anak kanan.

* Dalam min leftist tree setiap node lebih kecil dari anaknya.
* Dalam max leftist tree setiap node lebih besar dari anaknya.

COMBINE :  
Operasi utama Leftist Tree adalah menggabungkan dua pohon kiri(x dan y) menjadi satu pohonkiri yang berisi semua elemen dalam x dan y.

Menyisipkan dan menghapus data dapat dilakukan dengan menggunakan penggabungan operasi.

INSERT & DELETE :

Insert:

* Membuat leftist tree, lalu b dengan node tunggal, node dimasukkan.
* Gabungkan utama leftist tree dengan b.

Delete:

* Hapus root.
* Kombinasikan kiri–anak sub–pohon akar dengan anak sub–pohon akar dan mendapatkan leftist tree baru.

TRIES

Tries (prefix tree) adalah struktur data tree memerintahkan yang digunakan untuk menyimpanarray asosiatif (biasanya string)

The TRIE Istilah berasal dari kata RETRIEVAL, karena itu Tries bisa di sebut juga sebagai Retrieval.

