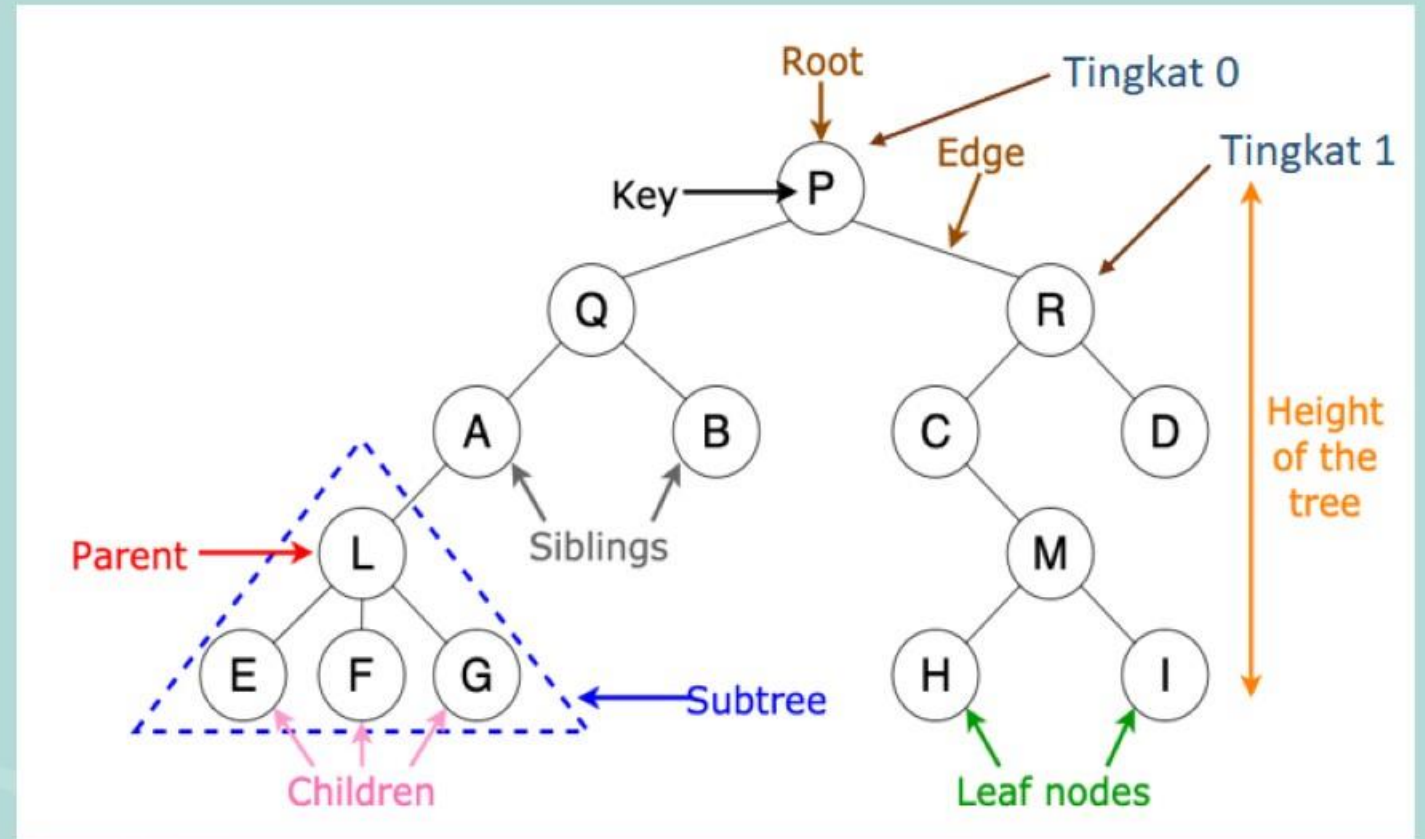




Struktur Data Pohon

Pendahuluan

- Kumpulan *node* dengan nilai tertentu, yang memiliki nol atau lebih *child*.
- Berbentuk menyerupai akar pohon dengan susunan hierarki.
- Terdiri dari satu simpul akar (*root node*) dan *subtree-subtree*.
- *Subtree* merupakan bagian dari keseluruhan *tree* yang ada.
- *Height* menunjukkan tingkatan dalam suatu *tree*.

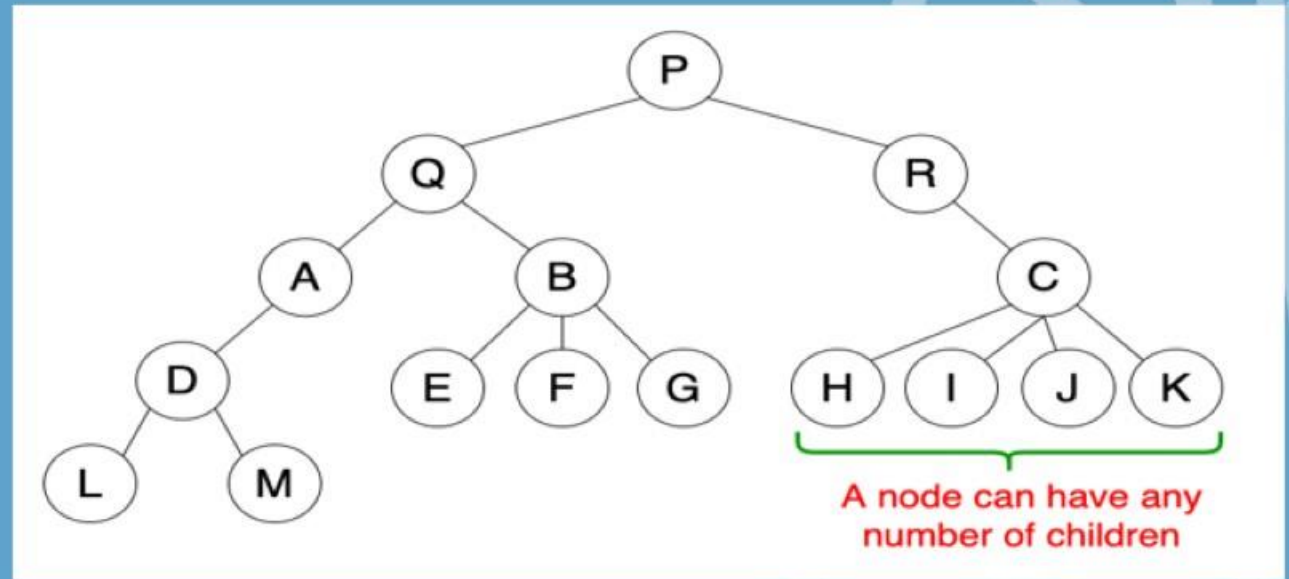


Terminologi Pohon

Degree	Banyaknya <i>child</i> dalam <i>node</i> tertentu
Node Internal	<i>Node</i> yang memiliki <i>child</i>
Node External	<i>Node</i> yang tidak memiliki <i>child</i>
Sibling	<i>Node</i> yang memiliki <i>parent</i> yang sama dan dapat membentuk struktur spesifik bernama <i>subtree</i> .
Size	Banyaknya tingkatan dalam suatu <i>tree</i>
Root	<i>Node</i> yang tidak memiliki <i>parent</i>
Leaf	<i>Node</i> dalam <i>tree</i> yang tidak memiliki <i>child</i>
Edge	Garis yang menggambarkan hubungan, dapat memiliki nilai yang menandakan beban antara 2 <i>node</i> .

General Tree

- Tidak ada batasan pada *degree* suatu *node*.
- *Subtree* tidak berurutan karena *node* tidak diurutkan menurut kriteria tertentu.
- Setiap *node* memiliki *in-degree* (jumlah *node parent*) satu, dan *out-degree* maksimum (banyaknya simpul anak) sebanyak n .
- Untuk menyimpan data hierarki seperti struktur folder.



Binary Tree

Struktur data pohon yang memiliki maksimal 2 anak, yaitu *left child* dan *right child*. Metode penempatan *record* dalam *database*, terutama *random access memory* (RAM).

Maksimum *node* pada setiap tingkat = 2^n

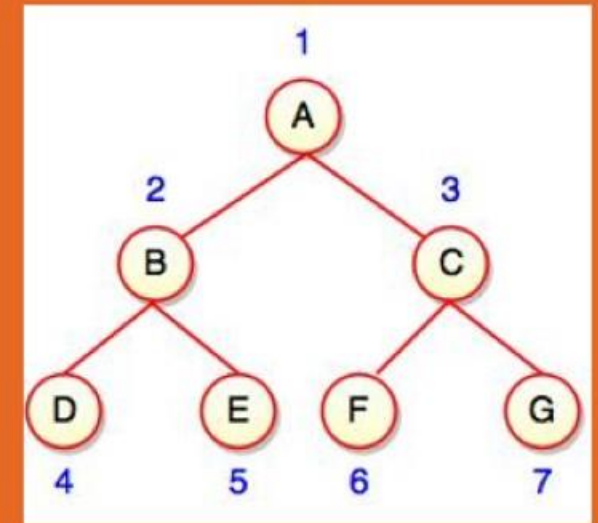
Total *node* pada *binary tree* = $2^n - 1$

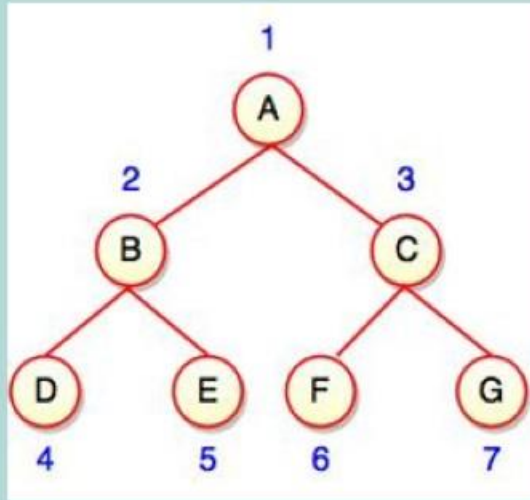
- **Representasi *Binary Tree* menggunakan *Array***

Array mewakili simpul yang diberi nomor secara berurutan tingkat demi tingkat dari kiri ke kanan.

0	1	2	3	4	5	6	7
7	A	B	C	D	E	F	G

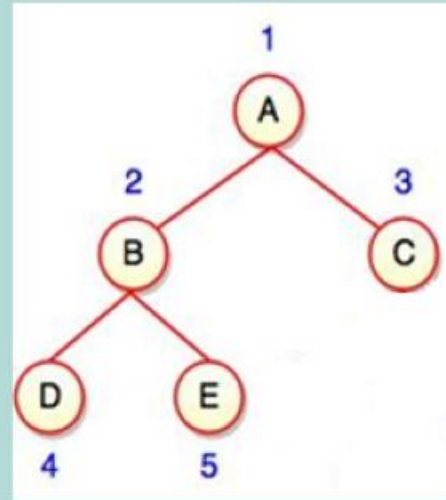
Indeks pertama array yaitu '0' menyimpan jumlah total *node* dalam sebuah tree. Setiap *node* yang memiliki indeks *i* dimasukkan ke dalam array sebagai elemen ke-*i*.





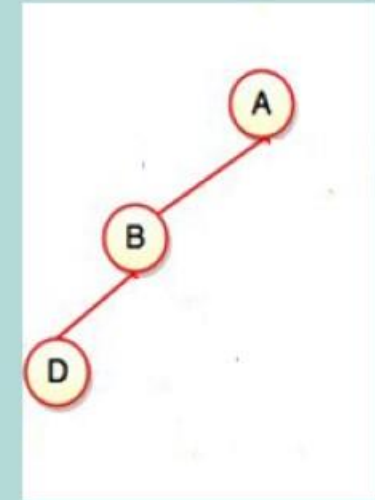
Full Binary Tree

- Setiap *node* (kecuali *leaf*) pasti memiliki 2 *child*
- Setiap subpohon memiliki panjang *path* yang sama



Complete Binary Tree

- Setiap *node* (kecuali *leaf*) pasti memiliki 2 *child*
- Setiap subpohon boleh memiliki panjang *path* yang berbeda

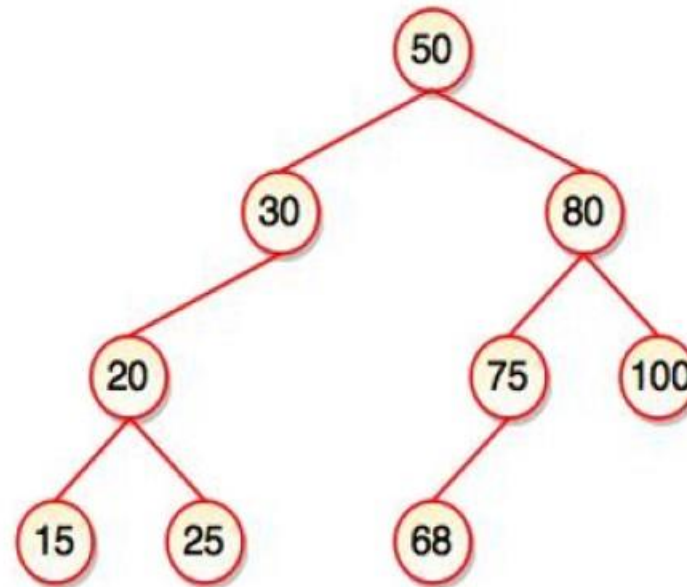


Skewed Binary Tree

- Setiap *node* (kecuali *leaf*) hanya memiliki 1 *child*

Binary Search Tree

- Pohon biner yang subpohon kiri dari setiap *node* memiliki nilai yang lebih kecil dari *parent node*-nya dan subpohon kanan berisi nilai yang lebih besar.
- *Binary search tree* (BST) digunakan untuk meningkatkan kinerja pohon biner dalam melakukan operasi pencarian.

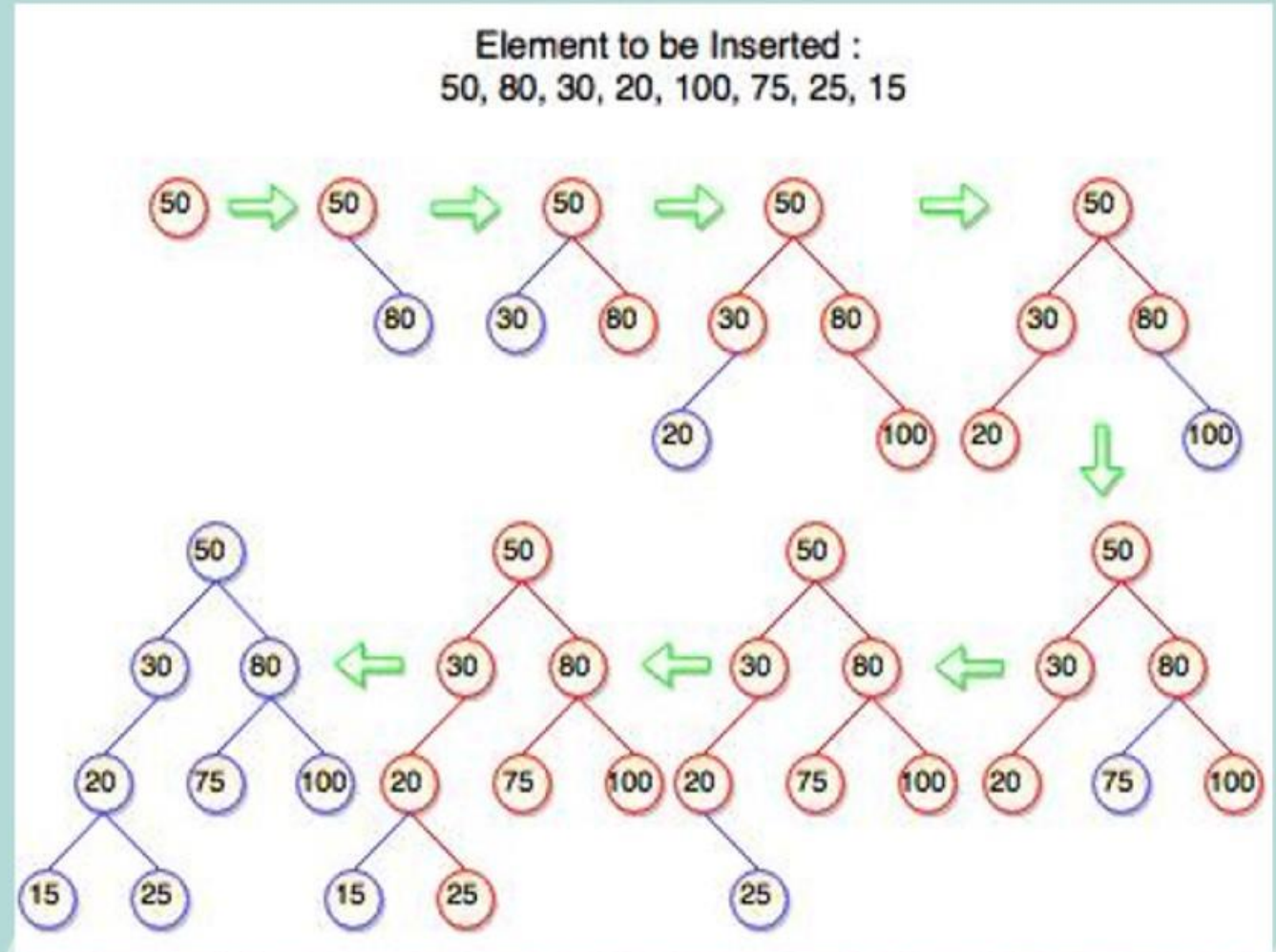


Penyisipan Binary Search Tree

Operasi penyisipan (*insert*) dilakukan dengan kompleksitas waktu $O(\log n)$ dalam BST.

Langkah algoritma operasi penyisipan:

- Buat node baru dengan sebuah nilai.
- Sisipkan sebuah node baru, dan cek apakah node baru tersebut lebih kecil atau lebih besar dari parent node.
- Jika node baru lebih kecil atau sama, maka letakkan di sebelah kiri, jika node baru lebih besar, maka letakkan di sebelah kanan.



Pencarian Binary Search Tree

Operasi pencarian (*search*) pada *binary search tree* dilakukan dengan kompleksitas waktu $O(\log n)$.

Langkah algoritma operasi pencarian:

- Bandingkan nilai yang dicari oleh pengguna dengan *root node*.
- Jika nilai yang dicari lebih kecil, maka lanjutkan pencarian ke subpohon kiri.
- Jika nilai yang dicari lebih besar, maka lanjutkan pencarian ke subpohon kanan.
- Ulangi proses hingga *node* tujuan ditemukan.
- Jika pencarian sudah mengunjungi semua *leaf node* dan data tetap tidak ditemukan, maka tampilkan "data tidak ditemukan".

Binary Tree Traversal

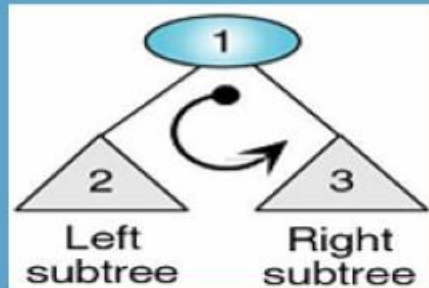
Proses mengunjungi setiap *node* pohon tepat satu kali. *Binary tree traversal* didefinisikan secara rekursif.

Preorder Traversal

- Cetak *root node* yang dikunjungi
- Kunjungi subpohon kiri
- Kunjungi subpohon kanan

Fungsi:

- Membuat salinan pohon baru
- Mendapat ekspresi *prefix*

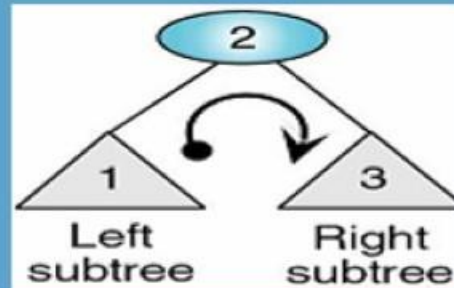


Inorder Traversal

- Kunjungi subpohon kiri
- Cetak *root node* yang dikunjungi
- Kunjungi subpohon kanan

Fungsi:

- Memproyeksikan ke bentuk satu dimensi kembali

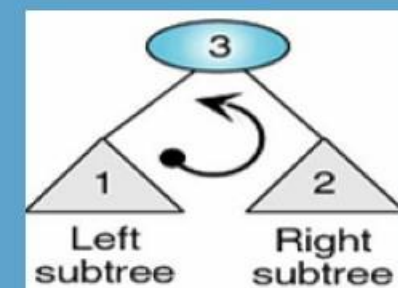


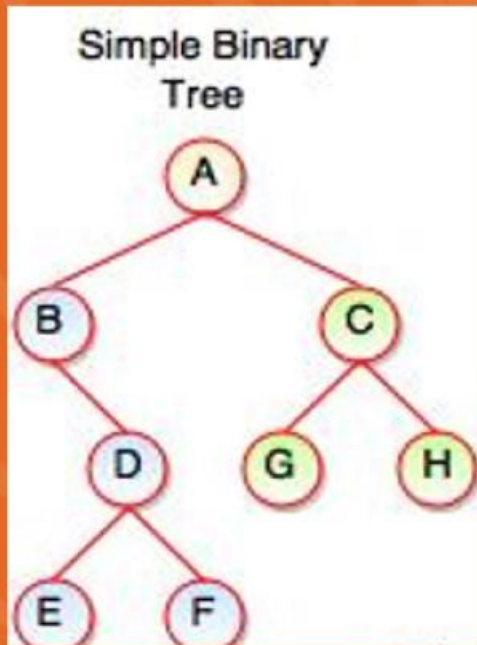
Postorder Traversal

- Kunjungi subpohon kiri
- Kunjungi subpohon kanan
- Cetak *root node* yang dikunjungi

Fungsi:

- Menghapus *node* mulai dari *leaf node* hingga *root node*





Preorder Traversal: ABDEFCGH

- Langkah 1: A + B (B + *Preorder* D (D + *Preorder* E dan F)) + C (C + *Preorder* G dan H)
- Langkah 2: A + B + D (E + F) + C (G + H)
- Langkah 3: A + B + D + E + F + C + G + H

Inorder Traversal: BEDFAGCH

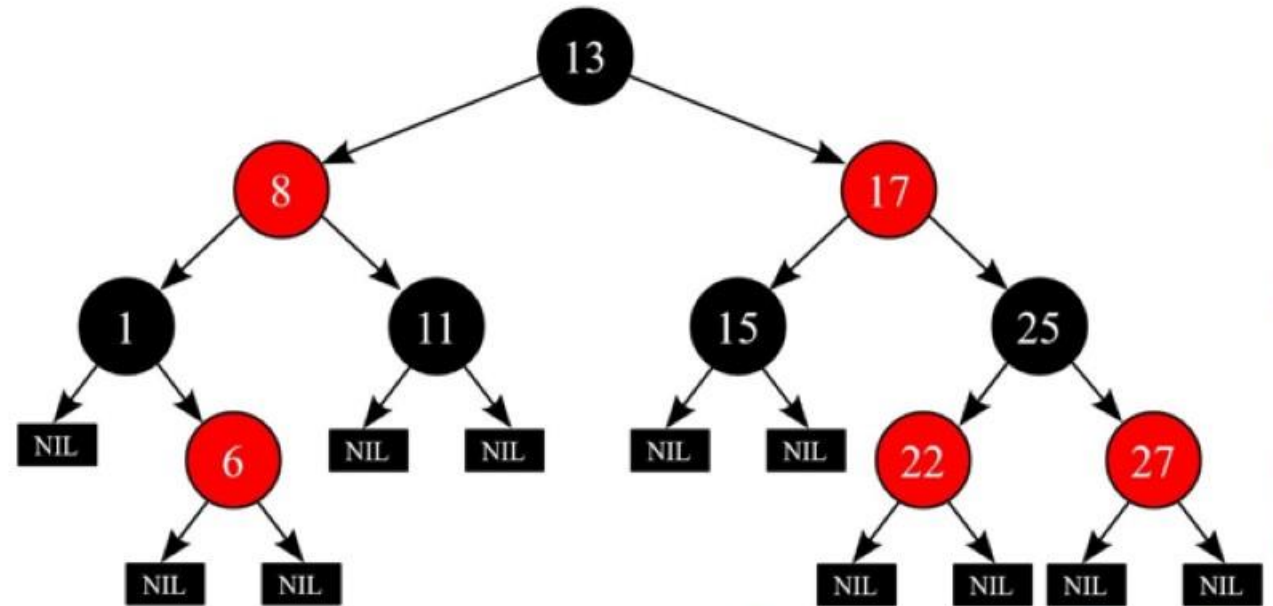
- Langkah 1: B + (*Inorder* E) + D + (*Inorder* F) + (Root A) + (*Inorder* G) + C (*Inorder* H)
- Langkah 2: B + (E) + D + (F) + A + G + C + H
- Langkah 3: B + E + D + F + A + G + C + H

Postorder Traversal: EFD BGHCA

- Langkah 1: (*leaf node* terakhir) => ((*Postorder* E + *Postorder* F) + D + B)) + ((*Postorder* G + *Postorder* H) + C) + (Root A)
- Langkah 2: (E + F) + D + B + (G + H) + C + A
- Langkah 3: E + F + D + B + G + H + C + A

Red Black Tree

- Setiap node memiliki tambahan 1 bit penyimpanan untuk kode warna merah atau hitam.
- Dapat menyeimbangkan diri dengan memastikan tidak ada jalur yang 2 kali lebih panjang dari jalur lain.
- Akar pohon (*root node*) selalu hitam.
- Tidak ada dua *node* merah yang berdekatan (sebagai *parent* atau *child*).
- Semua *leaf node* (yang dilambangkan sebagai NULL) berwarna hitam.
- Setiap jalur dari sebuah *node* (termasuk *root*) ke *leaf node* NULL memiliki jumlah *node* hitam yang sama.



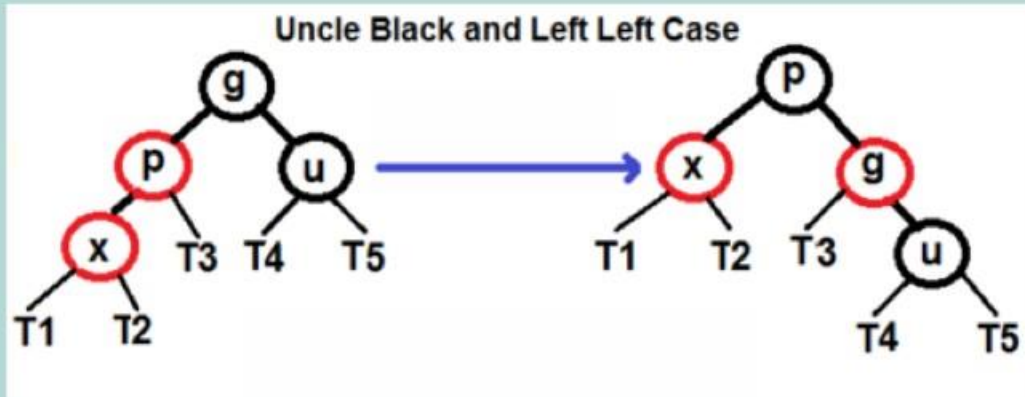
Fungsi:

- Sebagai dasar struktur data yang digunakan dalam C++ STL (*map*, *multimap*, *multiset*).
- Digunakan dalam '*Completely Fair Scheduler*' yang digunakan pada kernel Linux.
- Digunakan dalam implementasi sistem pemanggilan *epoll* dari kernel Linux.

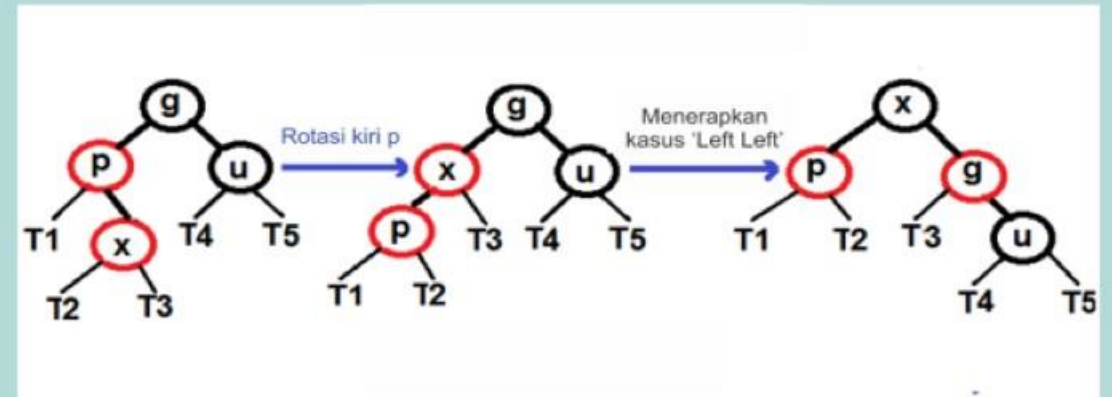
Mengapa *Red Black Tree*?

Sebagian besar operasi BST membutuhkan waktu $O(h)$ dengan h adalah tinggi BST. Jika tinggi pohon *red-black* tetap $\log(n)$, dengan n adalah jumlah *node*, setelah operasi penyisipan dan penghapusan, maka batas atas kompleksitas $O(\log(n))$ untuk semua operasi lainnya.

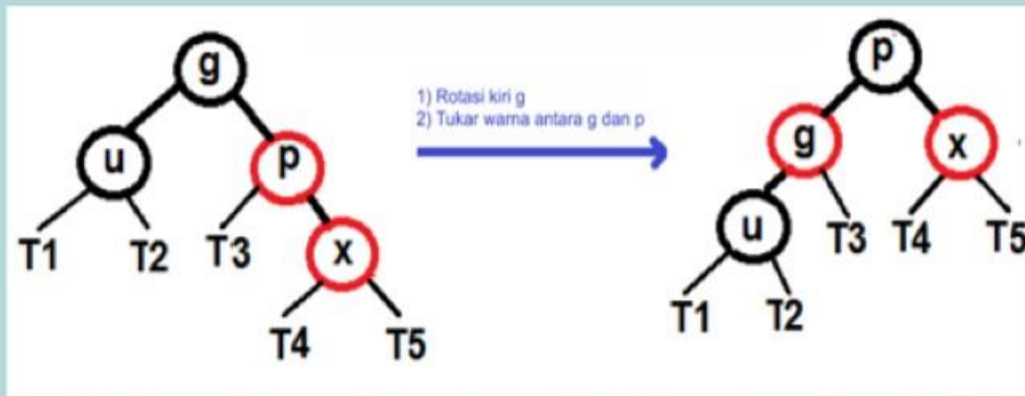
Operasi Saat Uncle Node Berwarna Hitam



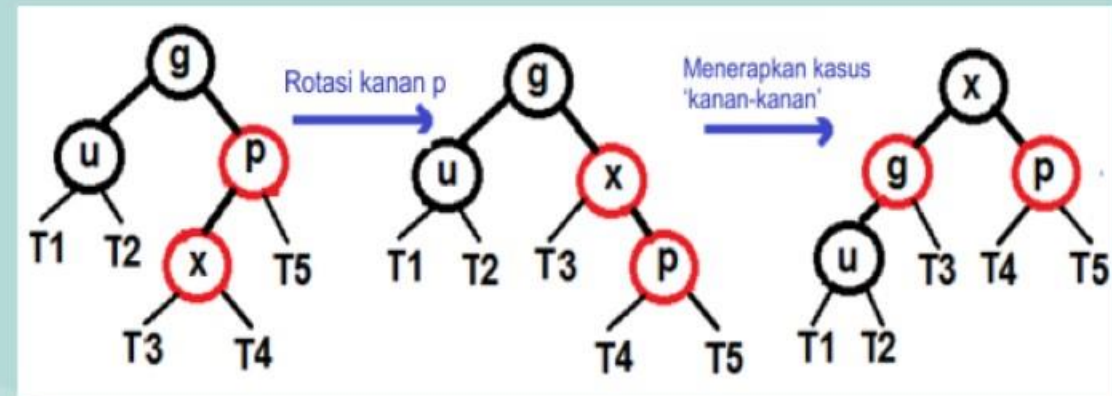
Metode Kiri Kiri: saat p adalah *left child* dari node g, dan node x adalah *left child* dari p



Metode Kiri Kanan: saat p adalah *left child* dari node g, dan node x adalah *right child* dari p



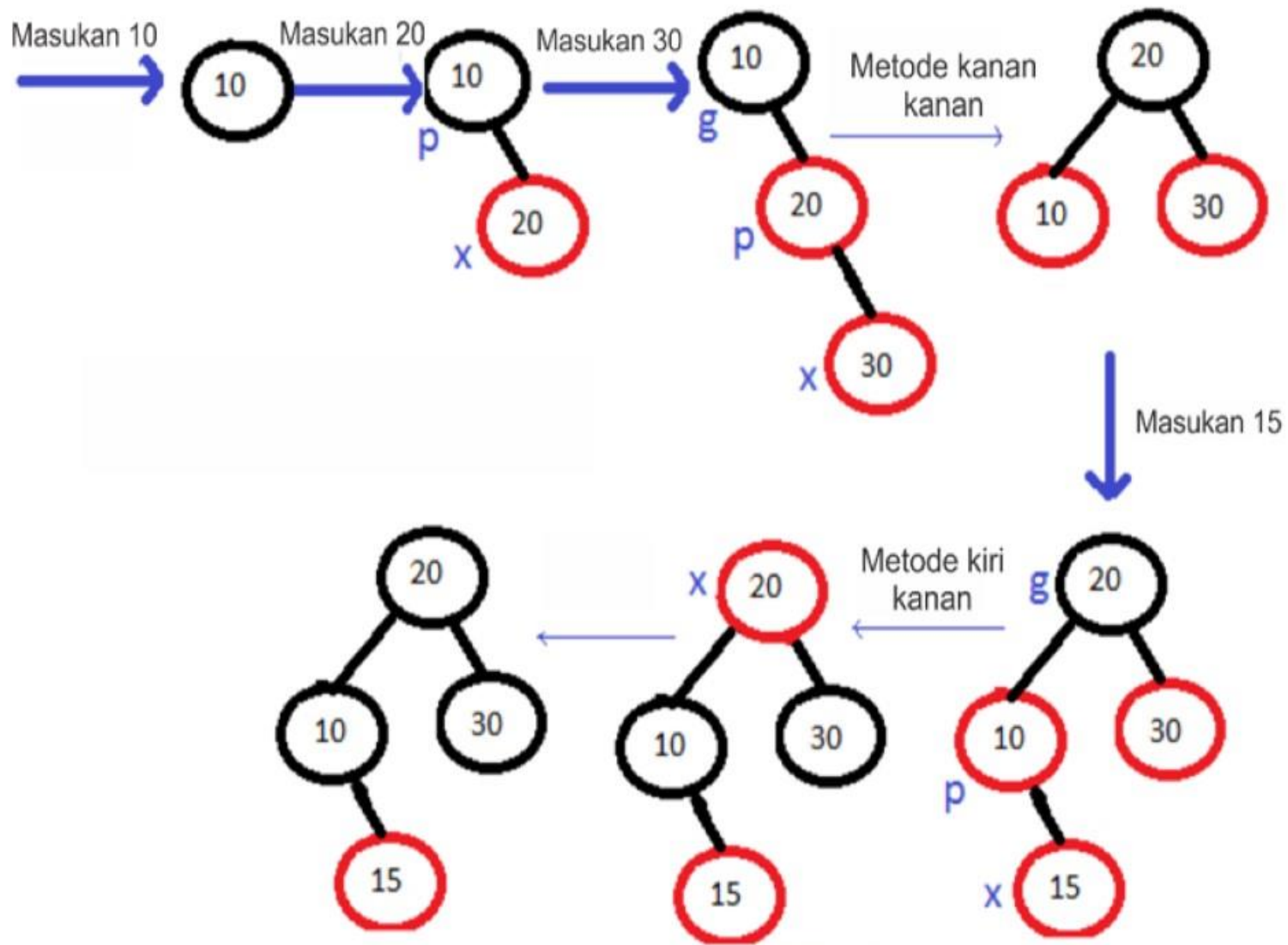
Metode Kanan Kanan: cermin dari metode kiri kiri



Metode Kanan Kiri: cermin dari metode kiri kanan

Penyisipan Red Black Tree

Masukan nilai 10, 20, 30, dan 15 ke dalam pohon kosong



AVL Tree

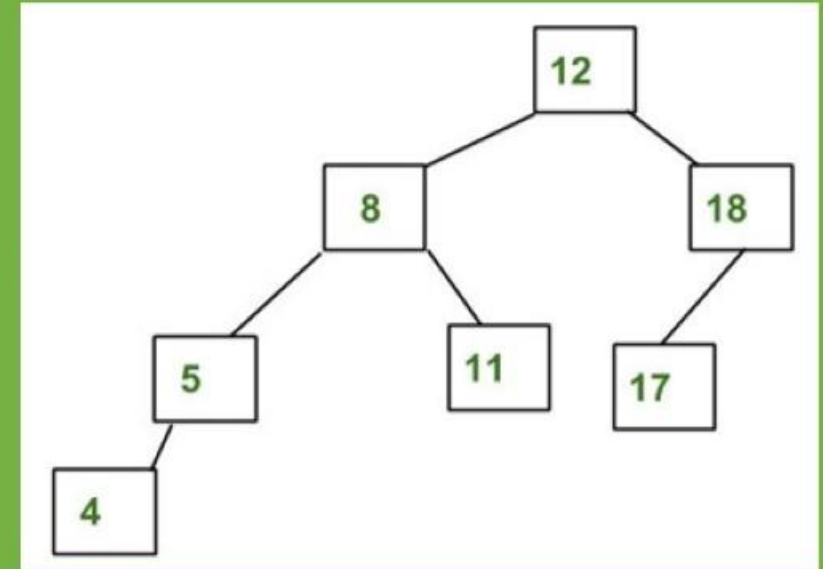
AVL tree memiliki perbedaan tinggi atau level antara subpohon kiri dan subpohon kanan maksimal 1.

Fungsi:

- Waktu pencarian dan bentuk tree dapat dipersingkat dan disederhanakan.
- Menyeimbangkan *Binary Search Tree*.

Mengapa *AVL Tree*?

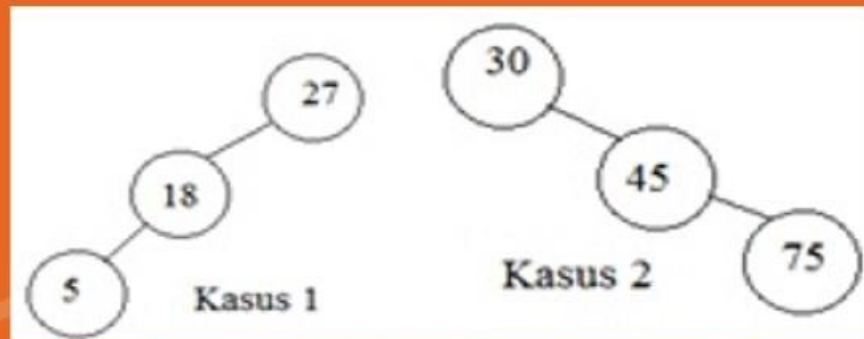
Sebagian besar operasi BST membutuhkan waktu $O(h)$ dengan h adalah tinggi BST. Jika tinggi pohon tetap $O(\log(n))$ setelah penyisipan dan penghapusan, maka batas atas waktu adalah $O(\log(n))$ untuk setiap operasi. Ketinggian untuk *AVL tree* selalu $O(\log(n))$ dengan n adalah jumlah *node* di pohon.



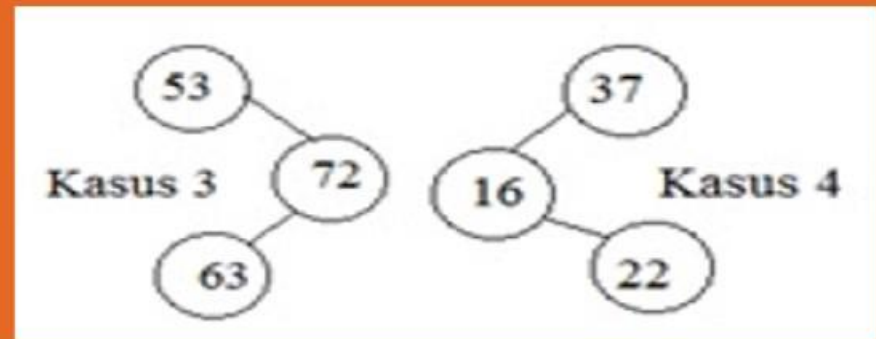
Penyisipan AVL Tree

Penyelesaian kasus yang pada umumnya terjadi saat penyisipan (T: node yang harus diseimbangkan kembali):

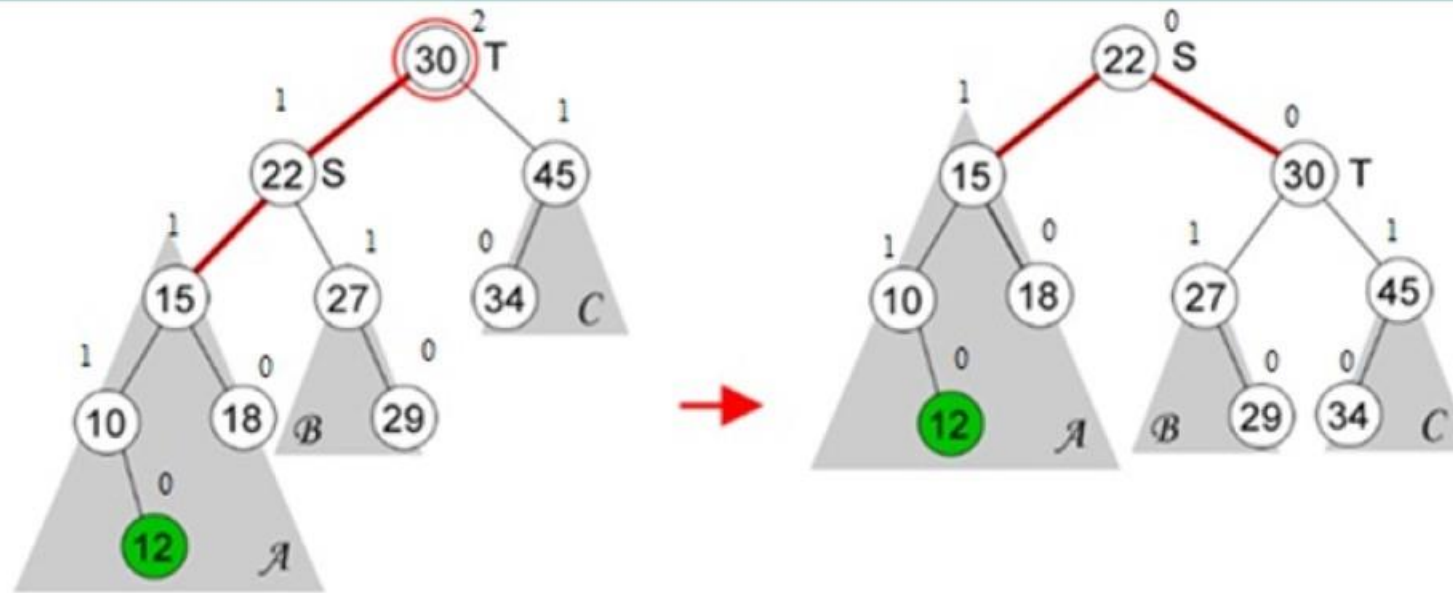
- **Single rotation:** node terdalam terletak pada *subtree* kiri dari anak kiri T (*left-left*), atau *subtree* kanan dari anak kanan T (*right-right*).
- **Double rotation:** node terdalam terletak pada *subtree* kanan dari anak kiri T (*right-left*), atau pada *subtree* kiri dari anak kanan T (*left-right*).



Single Rotation

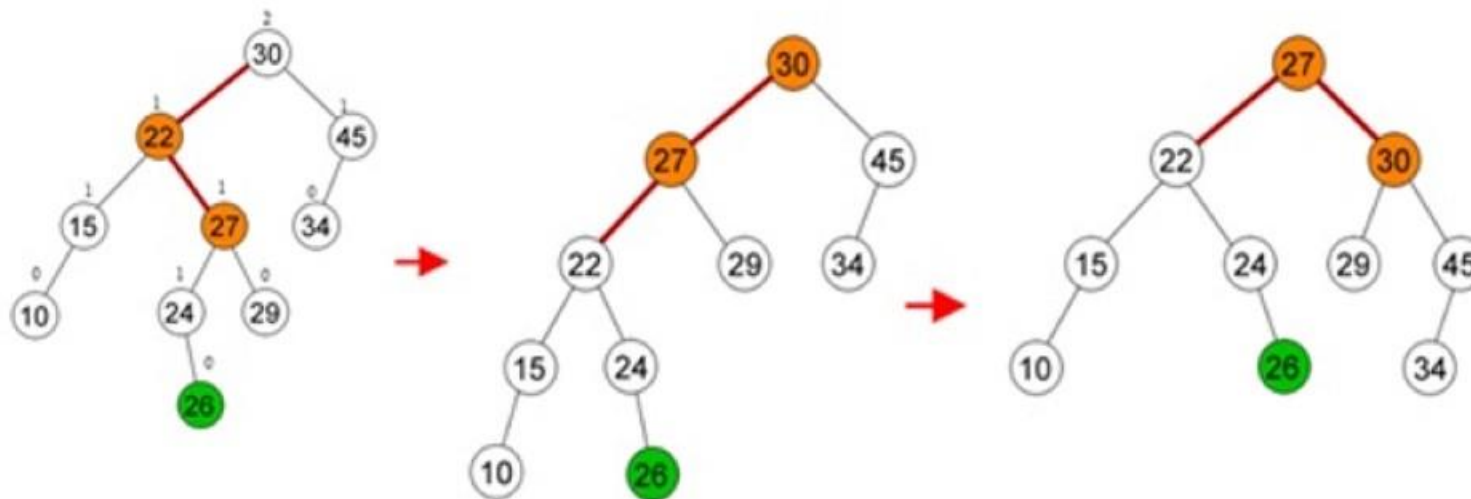


Double Rotation



Single Rotation

Jika di-insert *node* baru dengan nilai 12, maka akan terjadi ketidakseimbangan pada posisi root.



Double Rotation

Jika di-insert *node* bernilai 26, maka akan terjadi ketidakseimbangan.

B-Tree

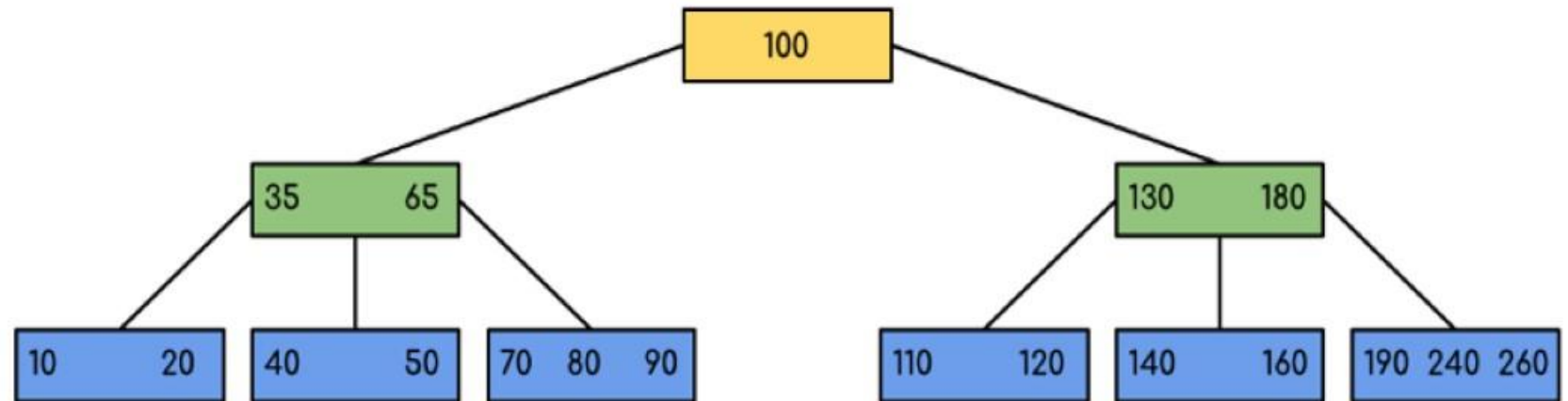
Struktur data pohon yang membuat data diurutkan dan memungkinkan pencarian, penyisipan, dan penghapusan dalam penambahan waktu logaritmik.

Fungsi:

- Mengurangi jumlah akses *disk*. Sering digunakan dalam *database* dan sistem *file*.
- Menyeimbangkan diri sendiri.

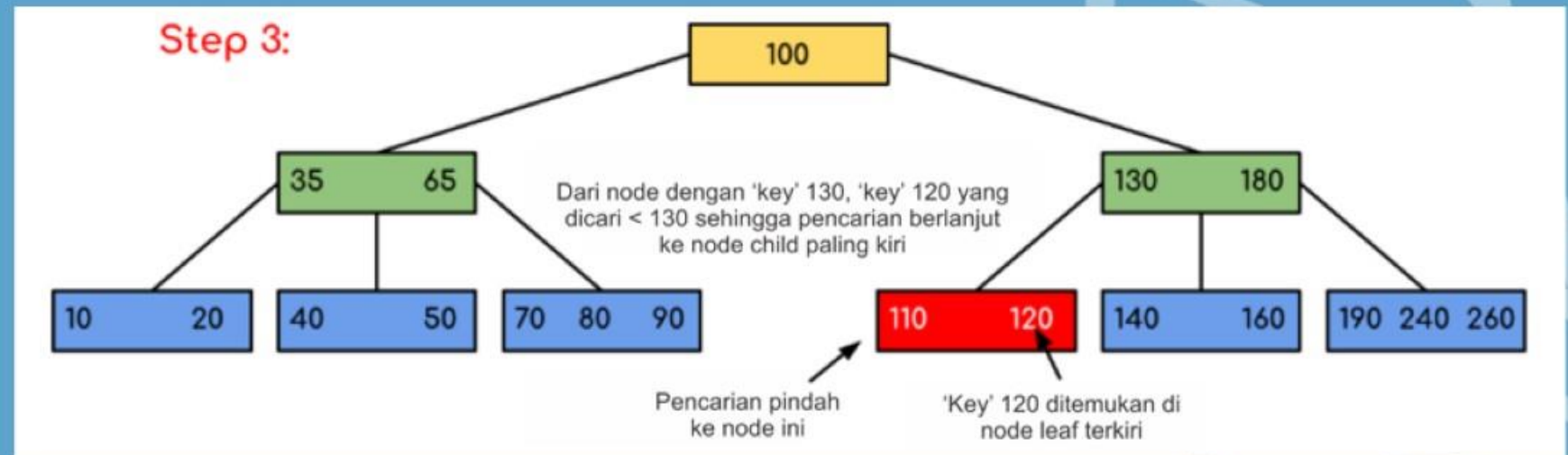
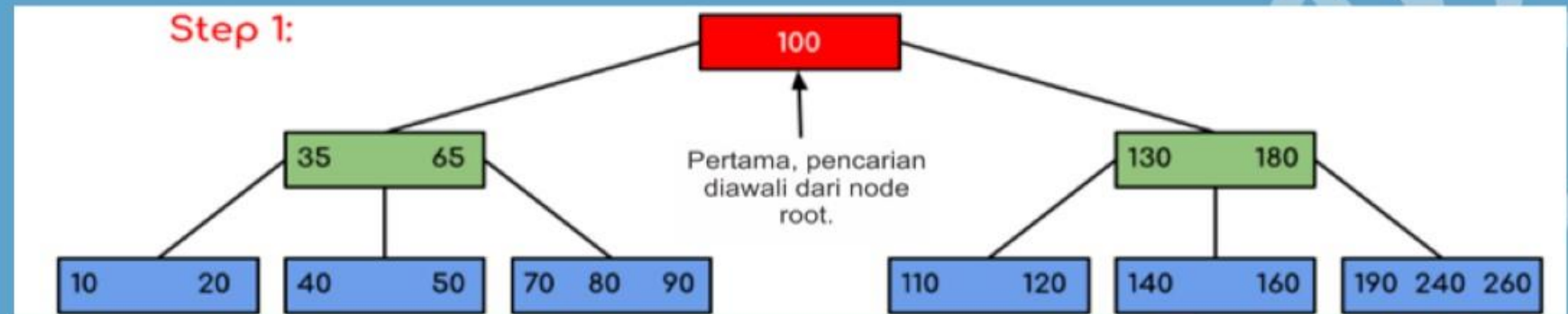
Mengapa *B-Tree*?

Sebagian besar operasi pohon memerlukan akses *disk* dengan kompleksitas $O(h)$ dengan h adalah tinggi pohon. *B-tree* dioptimalkan untuk pembacaan dan penulisan blok data yang besar.



Pencarian B-Tree

Mencari *item* 120



Penyisipan B-Tree

Struktur data *B-tree* yang awalnya kosong dengan derajat minimum t sebagai 3, dan urutan bilangan bulat 10, 20, 30, 40, 50, 60, 70, 80 dan 90.

Insert 10



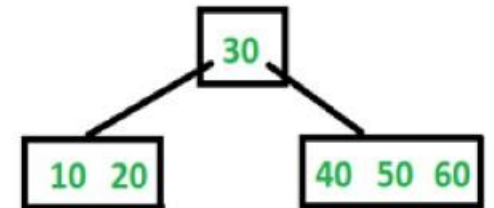
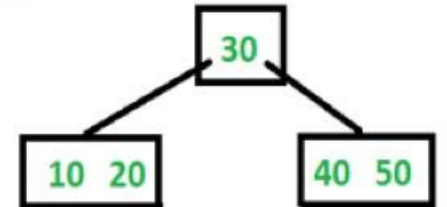
Awalnya root adalah null.

Insert 20, 30, 40 and 50



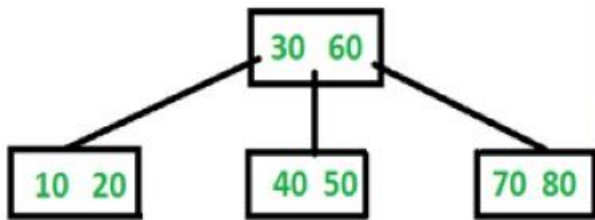
Semua masukkan ke *root* karena jumlah *key* maksimum yang dapat ditampung sebuah *node*: $2 * t - 1$ yaitu 5.

Insert 60



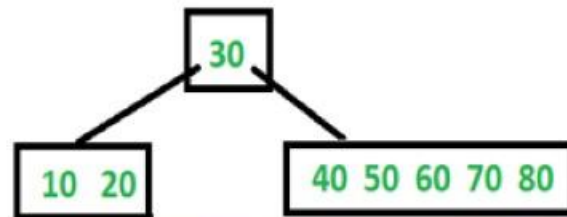
Karena *root node* sudah penuh, bagi menjadi dua, masukkan 60 ke *node child* yang sesuai.

Insert 90



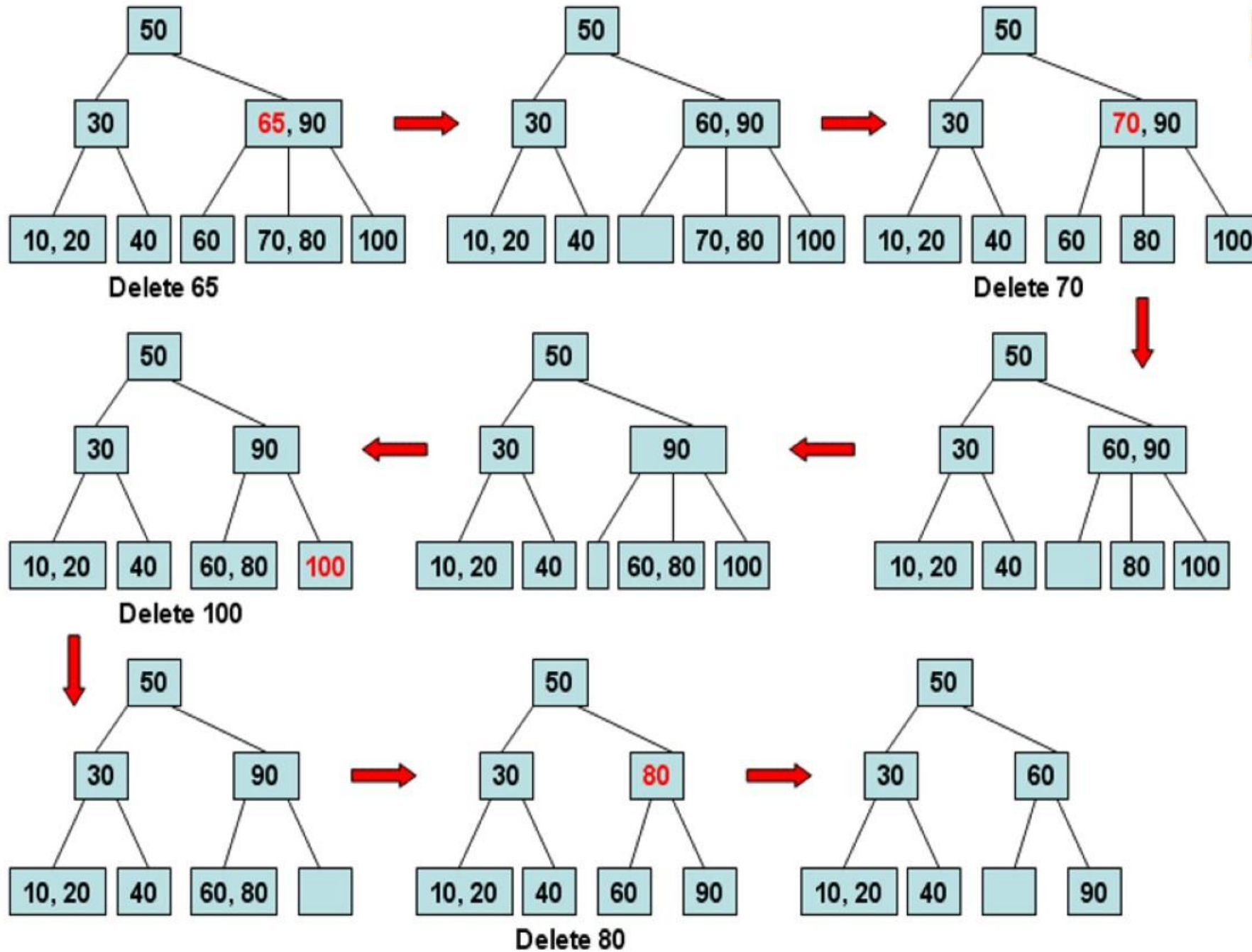
Penyisipan ini melibatkan operasi *splitting B-tree*. *Key* tengah kemudian akan naik ke induknya.

Insert 70 and 80



Dua *key* baru ini akan disisipkan ke *node leaf* yang sesuai tanpa operasi *splitting B-tree*.


Penghapusan B-Tree

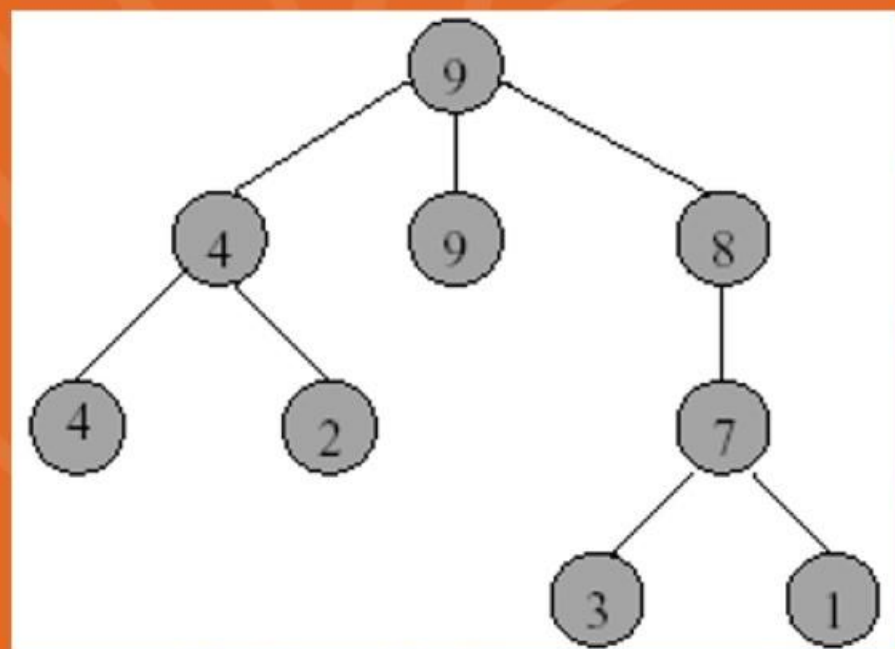


Antrian Berprioritas

- Antrian berprioritas adalah kumpulan dari kosong atau banyak elemen, setiap elemen mempunyai prioritas atau nilai.
- Urutan penghapusan dari antrian berprioritas ditentukan berdasar prioritas elemen, bukan berdasar urutan masuk elemen dalam antrian.

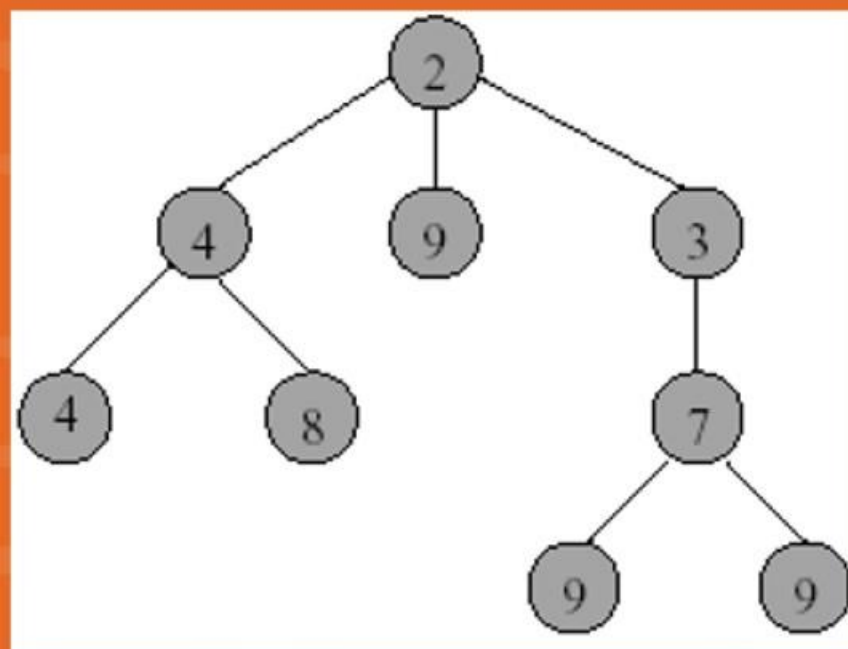
Implementasi:

- *Heap* adalah *complete binary tree* yang disimpan dengan efisien menggunakan bentuk *array*.
 - *Leftist tree* adalah struktur data *linked* yang disesuaikan untuk penggunaan antrian berprioritas.
- 



Max Tree

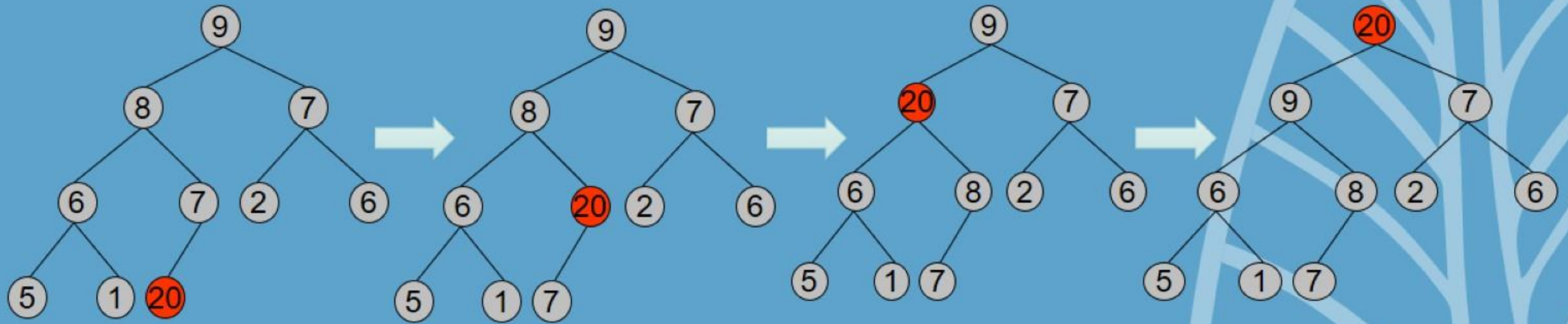
- Pohon yang nilai setiap *node*-nya lebih besar atau sama dengan nilai *child* (jika punya).
- *Root node* memiliki nilai terbesar.
- **Max heap** adalah *max tree* yang juga sebuah *complete binary tree*



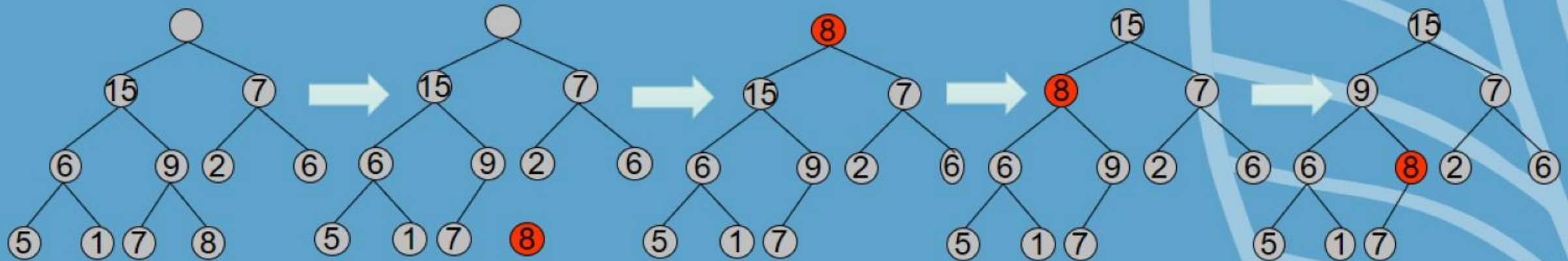
Min Tree

- Pohon yang nilai setiap *node*-nya lebih kecil atau sama dengan nilai *child* (jika punya).
- *Root node* memiliki nilai terkecil.
- **Min heap** adalah *min tree* yang juga sebuah *complete binary tree*

Penyisipan Max Heap

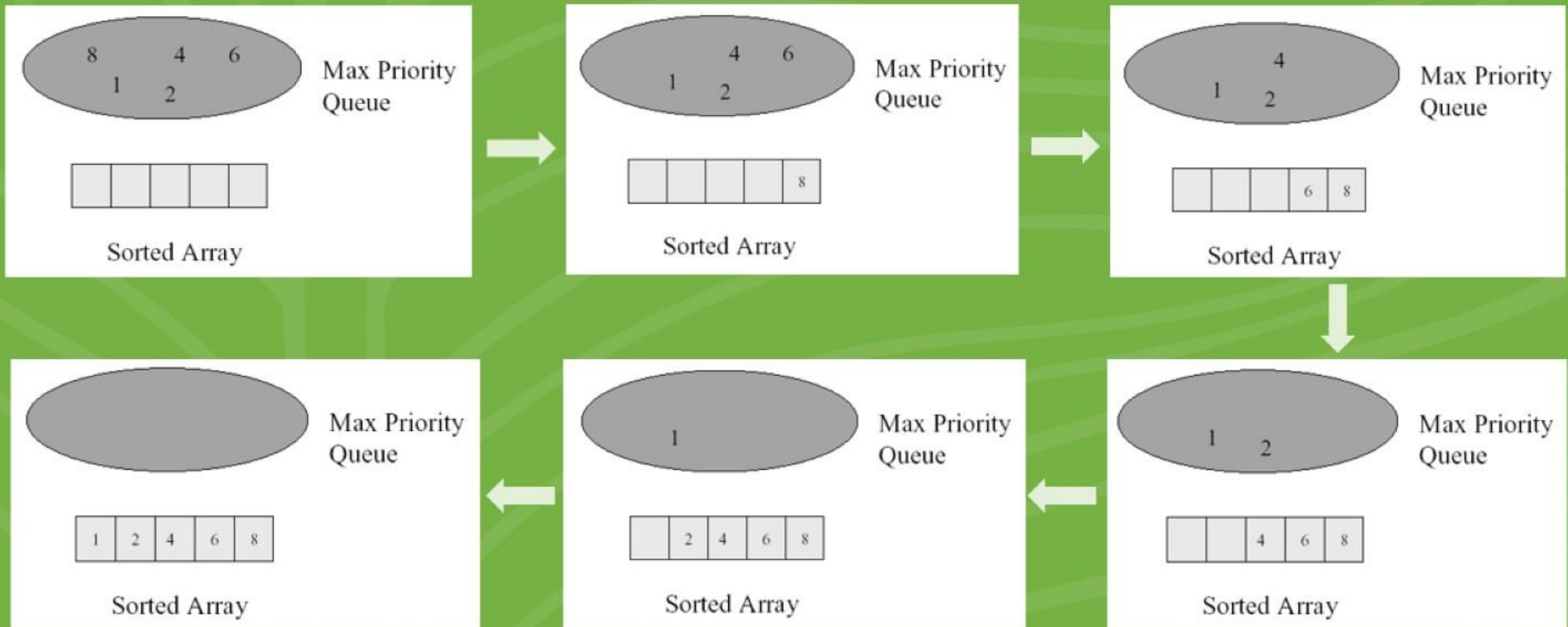


Penghapusan Max Heap



Pengurutan Heap

- Letakkan elemen yang akan diurutkan dalam antrian berprioritas.
- Jika *min priority queue* yang digunakan, elemen diekstrak dengan urutan prioritas naik.
- Jika *max priority queue* yang digunakan, elemen diekstrak dengan urutan prioritas menurun.





Referensi

Persada, Anugerah Galang; Hernanda, Henoch; dan Haifa, Juz'an Nafi. 2020. Modul Pembelajaran Mandiri Algoritma dan Struktur Data: Struktur Data Pohon.

adeab.staff.ipb.ac.id/files/2011/12/struktur-data-pohon.ppt

lecturer.ukdw.ac.id/anton/download/Tlstrukdat10.ppt