



ALGORITME DAN STRUKTUR DATA

Struktur Data Pohon



A. Pengenalan Struktur Data Pohon

Struktur data pohon merupakan kumpulan *node* dengan nilai tertentu, yang memiliki nol atau lebih *child* dengan urutan tertentu. Struktur data pohon menyimpan informasi berbentuk menyerupai akar pohon dengan susunan hierarki (dari atas ke bawah), juga merupakan struktur data non linear jika dibandingkan dengan *array*, *linked list*, *stack*, dan *queue*. Struktur data pohon terdiri dari satu simpul akar (*root node*) dan *subtree-subtree*. Setiap *node* pohon memiliki paling banyak satu *node* induk.

Kelebihan dari penggunaan struktur data pohon:

- Menggambarkan relasi struktural dari data secara hirarkis
- Masukan data dan operasi pencarian akan lebih efisien
- Fleksibel, urutan *node* dapat diubah-ubah dengan mudah secara logis

B. Terminologi Pohon

Predecesor	Node yang berada diatas node tertentu.
Successor	Node yang berada dibawah node tertentu.
Ancestor	Seluruh node yang terletak sebelum node tertentu dan terletak pada jalur yang sama
Descendant	Seluruh node yang terletak setelah node tertentu dan terletak pada jalur yang sama
Parent	Predecessor satu level di atas suatu node.
Child	Successor satu level di bawah suatu node.
Sibling	Node-node yang memiliki parent yang sama
Subtree	Suatu node beserta descendantnya.
Size	Banyaknya node dalam suatu tree
Height	Banyaknya tingkatan dalam suatu tree
Root	Node khusus yang tidak memiliki predecessor.
Leaf	Node-node dalam tree yang tidak memiliki successor.
Degree	Banyaknya child dalam suatu node

Tabel 1. Terminologi Pohon

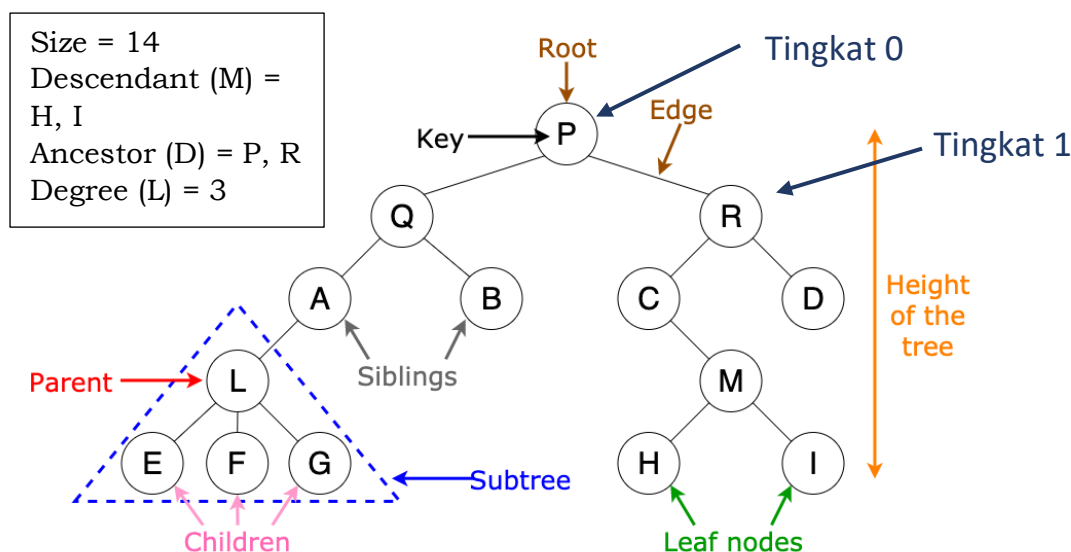
Tabel 1 menunjukkan istilah-istilah yang sering digunakan dalam struktur data pohon. Selain itu, terdapat sebutan lain yang memiliki arti sama dengan *leaf*, yaitu *node external*. Sedangkan *node* yang memiliki *child* disebut *node internal*. Banyaknya *child* dalam sebuah *node* disebut *degree*.

C. Ilustrasi Struktur Data Pohon

Gambar 1 dibawah ini menunjukkan salah satu ilustrasi dari sebuah sturktur data pohon. Sebuah struktur data pohon akan memiliki satu root atau akar. Sebuah root adalah sebuah *node*, dengan definisi *node* adalah satu unit pada struktur data pohon tersebut. Pada Gambar

1, sebuah *node* dapat berupa *Node P*, yang merupakan *root*. *Node* yang lainnya adalah *node Q* dan *R*, yang keduanya merupakan anak (*child*) dari *Node P*. Pada kondisi ini, *P* disebut memiliki 2 *child* (anak). Sedangkan *Q* dan *R* mempunyai *parent* / orang tua yang sama, yaitu *Node P*. Pada tingkatan selanjutnya, sebuah *tree* akan dapat memiliki nol atau lebih *child* / anak. Anak dari setiap *node* dapat mencerminkan jenis dari struktur data *tree* tersebut, sebagaimana yang akan dibawah pada bagian lain dari modul ini.

Setiap dua *node* dihubungkan oleh sebuah *edge*. *Edge* dapat memiliki nilai tertentu yang menandakan beban di antara dua *node*. Sebuah *node* akan memiliki sebuah kunci penanda. Kunci ini secara umum akan bernilai unik, sehingga mampu membedakan antara *node* satu dengan *node* lainnya.



Gambar 1. Contoh Struktur Data Pohon dan Istilahnya

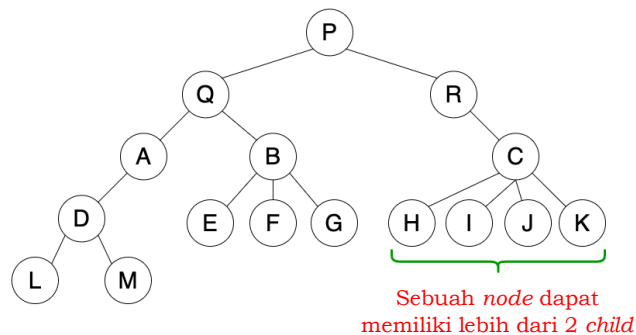
Setiap *node* yang memiliki saudara lain pada jalur yang berbeda (namun masih pada *height of tree* yang setara) disebut mempunyai sibling. Misalkan, pada Gambar 1 tersebut, terdapat sibling antara huruf A dan huruf B. Mengacu pada ilustrasi yang masih sama pula, setiap sibling dapat membentuk suatu struktur yang berbeda dan lebih spesifik ruang lingkungannya. Struktur ini diberi nama *subtree*.

D. Jenis-Jenis Struktur Data Pohon

1. General tree

General tree adalah struktur data yang setiap *node* dapat memiliki nol atau banyak *node* turunan, tidak ada batasan pada derajat (*degree*) suatu *node*. Subpohon (*Subtree*)

dari *general tree* tidak berurutan karena *node* tidak dapat diurutkan menurut kriteria tertentu. Setiap *node* memiliki *in-degree* (jumlah *node parent*) satu dan *out-degree* maksimum (banyaknya simpul anak) sebanyak n .



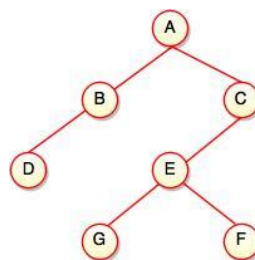
Gambar 2. General Tree

Fungsi

- Digunakan untuk menyimpan data hierarki seperti struktur folder.

a. Binary tree

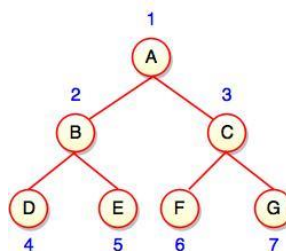
Pohon biner (*binary tree*) adalah tipe khusus dari struktur data. Dalam pohon biner, setiap *node* dapat memiliki maksimal dua anak, yaitu *left child* dan *right child*. Hal tersebut merupakan metode penempatan *record* dalam *database*, terutama dalam *random access memory* (RAM).



Gambar 3. Binary Tree

Representasi *Binary Tree* menggunakan *Array*

Array mewakili simpul yang diberi nomor secara berurutan tingkat demi tingkat dari kiri ke kanan, termasuk *node* kosong.



Gambar 4. *Binary tree* Representasi Menggunakan *Array*

Indeks *array* adalah nilai dalam *node* pohon dan nilai *array* yang diberikan ke simpul induk (*node parent*) dari indeks atau *node* tersebut. Nilai indeks *root node* (simpul akar) selalu -1 karena tidak ada induk untuk akar. Ketika *item data* dari pohon diurutkan dalam larik, angka yang muncul pada *node* akan berfungsi sebagai indeks *node* dalam *array*.

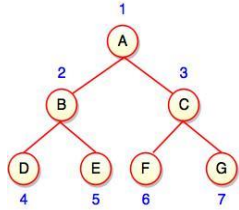
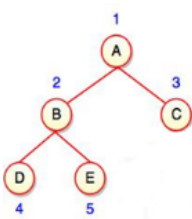
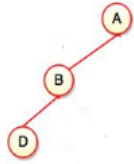


Gambar 5. Posisi Indeks dalam *Array* dari *Tree*

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*.

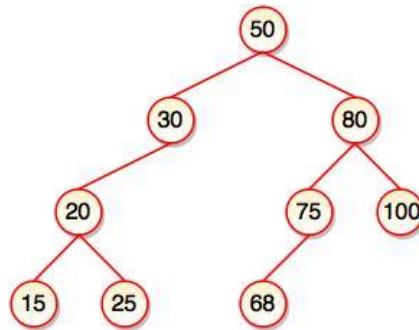
Gambar 5 di atas menunjukkan pohon biner yang direpresentasikan sebagai array. Nilai '7' adalah jumlah total *node*. Jika ada *node* yang tidak memiliki turunan, nilai null disimpan pada indeks yang sesuai dari *array*.

Jenis *Binary Tree*

<i>Full Binary Tree</i>	<i>Complete Binary Tree</i>	<i>Skewed Binary Tree</i>
Setiap <i>node</i> (kecuali <i>leaf</i>) pasti memiliki 2 <i>child</i> dan setiap subpohon memiliki panjang <i>path</i> yang sama	Setiap <i>node</i> (kecuali <i>leaf</i>) pasti memiliki 2 <i>child</i> dan setiap subpohon boleh memiliki panjang <i>path</i> yang berbeda	Setiap <i>node</i> (kecuali <i>leaf</i>) hanya memiliki 1 <i>child</i>
 <p>Gambar 6. <i>Full Binary Tree</i></p>	 <p>Gambar 7. <i>Complete Binary Tree</i></p>	 <p>Gambar 8. <i>Skewed Binary Tree</i></p>

b. *Binary Search Tree*

Binary Search Tree adalah 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.



Gambar 9. *Binary Search Tree*

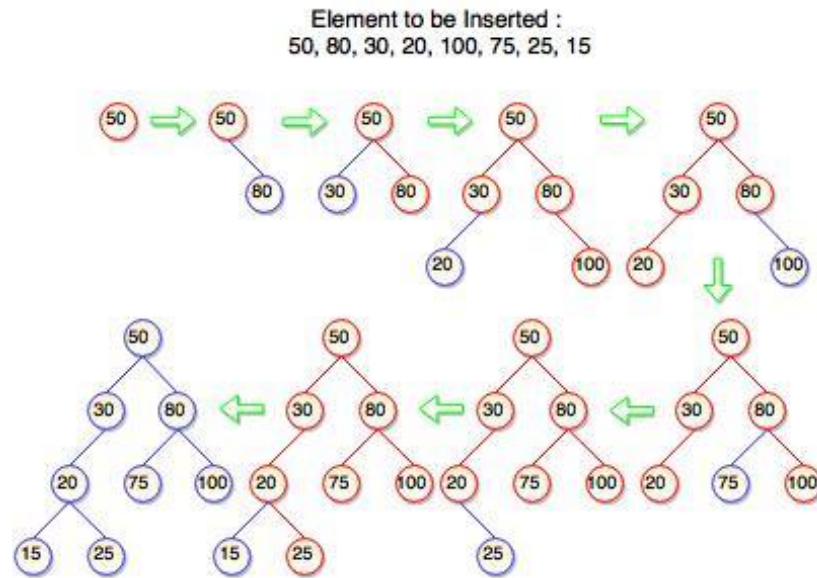
Operasi Pohon Pencarian Biner

- Operasi Penyisipan

Operasi penyisipan (*insert*) dilakukan dengan kompleksitas waktu $O(\log n)$ dalam BST. Operasi penyisipan dimulai dari simpul akar (*root node*) setiap kali elemen akan dimasukkan.

Langkah algoritma untuk operasi penyisipan (*insert*) pada BST:

- ❖ 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.



Gambar 10. Operasi Penyisipan (*Insert*)

- Operasi Pencarian

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

Langkah algoritma operasi pencarian pada BST:

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

c. Binary Tree Traversal

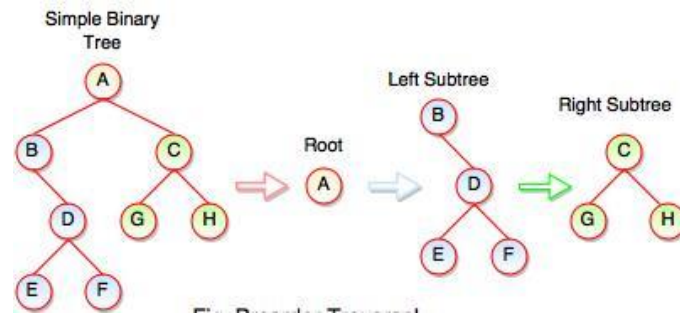
Traverse pohon biner (*binary tree*) adalah proses mengunjungi setiap *node* tepat satu kali. *Traversal* pohon biner juga didefinisikan secara rekursif.

Terdapat tiga teknik traversal, yaitu:

1) *Preorder Traversal*

Langkah-langkah *preorder traversal*:

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



Gambar 11. *Preorder Traversal*

Penggunaan *preorder*:

Preorder traversal digunakan untuk membuat salinan pohon baru. *Preorder traversal* juga digunakan untuk mendapatkan ekspresi awalan (*prefix expression*) pada pohon ekspresi notasi (*polish notation*). Sebagai contoh, jika ingin membuat replika pohon, letakkan *node* dalam array dengan *traversal preorder*. Kemudian lakukan operasi sisipan (*insert*) pada pohon baru untuk setiap nilai dalam larik.

Langkah-langkah *flow* dari *preorder traversal*:

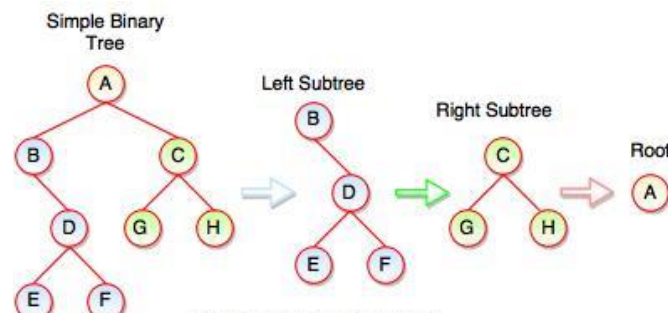
- ❖ $A + B (B + \text{Preorder } D (D + \text{Preorder } E \text{ dan } F)) + C (C + \text{Preorder } G \text{ dan } H)$
- ❖ $A + B + D (E + F) + C (G + H)$
- ❖ $A + B + D + E + F + C + G + H$

Preorder traversal: A B D E F C G H

2) *Postorder Traversal*

Langkah-langkah *postorder traversal*:

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



Gambar 12. *Postorder Traversal*

Penggunaan *postorder*:

Traversal postorder digunakan untuk menghapus satu-persatu *node* pohon dari *leaf node* ke *root node*.

Langkah-langkah *flow* dari *postorder traversal*:

❖ (leaf *node* terakhir) => ((Postorder E + Postorder F) + D + B)) + ((Postorder G + Postorder H) + C) + (Root A)

❖ (E + F) + D + B + (G + H) + C + A

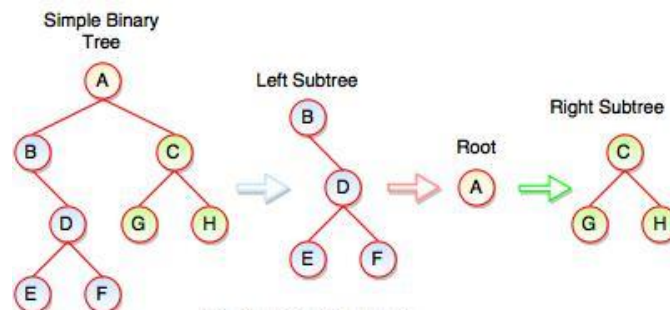
❖ E + F + D + B + G + H + C + A

Postorder traversal: E F D B G H C A

3) *Inorder Traversal*

Langkah-langkah *inorder traversal*:

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



Gambar 13. *Inorder Traversal*

Penggunaan *inorder*:

Dalam kasus *binary search tree* (BST), *inorder traversal* digunakan untuk mendapatkan nilai pada *node* dalam urutan tidak menurun dalam BST. Jika ingin memproyeksikan pohon kembali ke bentuk satu dimensi (*linked list*, *array*, dll.), maka digunakan *inorder traversal*. Pohon akan diproyeksikan dengan cara yang sama seperti saat dibuat.

Langkah-langkah *flow* dari *inorder traversal*:

❖ B + (Inorder E) + D + (Inorder F) + (Root A) + (Inorder G) + C (Inorder H)

❖ B + (E) + D + (F) + A + G + C + H

❖ B + E + D + F + A + G + C + H

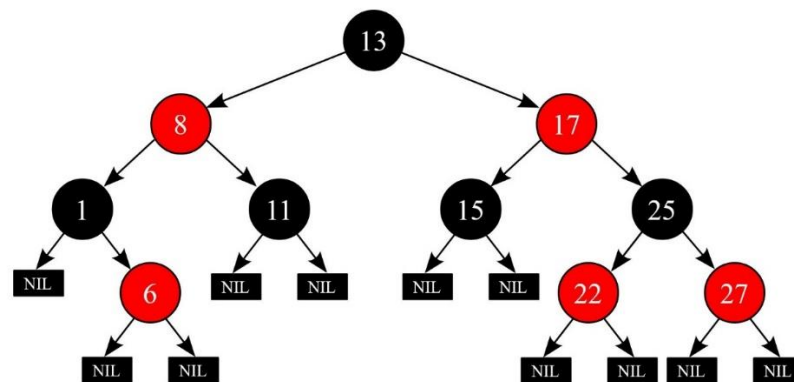
Inorder traversal: B E D F A G C H

2. Red-Black Tree

Red-black tree adalah BST dengan tambahan satu bit penyimpanan pada setiap node untuk menyimpan kode warna (merah atau hitam). Dengan memberikan warna simpul (*node*) pada setiap jalur dari akar (*root node*) sampai ke daun (*leaf node*), *red-black tree* memastikan bahwa tidak ada jalur yang dua kali lebih panjang dari jalur lainnya, sehingga akan selalu seimbang.

Red-black tree adalah *binary search tree* (BST) mengikuti aturan sebagai berikut.

- 1) Setiap *node* memiliki warna merah atau hitam.
- 2) Akar pohon (*root node*) selalu hitam.
- 3) Tidak ada dua *node* merah yang berdekatan (Sebagai *parent* atau *child*).
- 4) Semua *leaf node* (yang dilambangkan sebagai NULL) berwarna hitam.
- 5) Setiap jalur dari sebuah *node* (termasuk *root*) ke *leaf node* NULL memiliki jumlah *node* hitam yang sama.



Gambar 14. *Red-Black Tree*

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.

Kegunaan *red-black tree*:

- 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.

Cara untuk melakukan penyeimbangan dalam *red-black tree*:

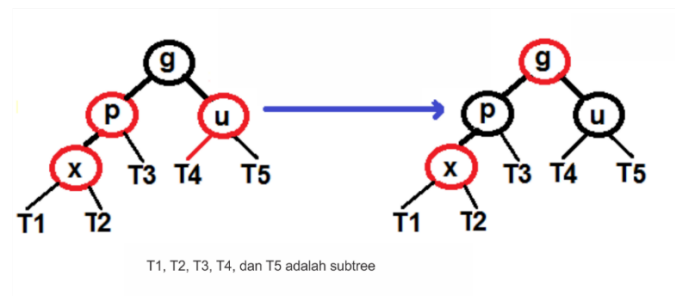
1) Pewarnaan ulang

2) Rotasi

Penyeimbangan dilakukan dengan cara pewarnaan ulang terlebih dahulu, jika tidak berhasil, baru dilakukan metode rotasi. Algoritme memiliki dua jenis kasus, tergantung pada warna *node* paman (sebelah dari *node parent*) dari tempat *node* baru akan diletakkan. Metode pewarnaan ulang untuk *node* paman berwarna merah, sedangkan untuk *node* paman berwarna hitam, dilakukan metode rotasi dan/atau pewarnaan ulang.

Misalkan x menjadi simpul (*node*) baru yang disisipkan.

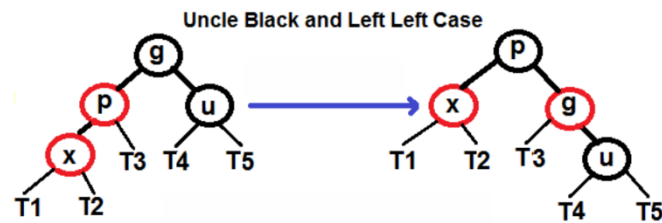
- Lakukan langkah penyisipan *binary search tree* (BST) seperti biasa dan buat warna *node* yang baru disisipkan menjadi merah.
- Jika x adalah *root*, ubah warna *node* x menjadi hitam (tinggi hitam pada pohon bertambah 1).
- Lakukan langkah-langkah dibawah ini jika warna induk x bukan hitam dan bukan simpul akar (*root node*).
 - a. Jika paman dari *node* x merah (*grandparent* dari *node* x pasti berwarna hitam berdasarkan aturan *red-black tree* nomor 5), lakukan langkah berikut:
 - i) Mengubah warna *node parent* dan *node uncle* menjadi hitam.
 - ii) Warna *grandparent* merah.
 - iii) Ubah $x = \text{grandparent}$ dari x , ulangi langkah 2 dan 3 untuk x baru lainnya.



Gambar 15. Penyisipan *Node* pada Pohon Merah-Hitam

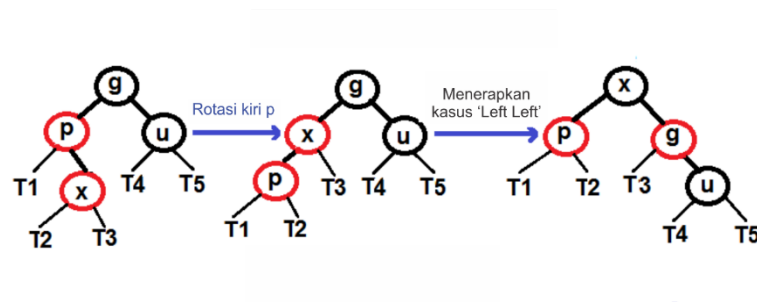
- b. Jika *uncle node* dari *node* x hitam, maka akan ada empat kemungkinan untuk x , parent *node* x (p) dan *grandparent node* x (g), lakukan langkah berikut:

- i) Kasus kiri-kiri (p adalah *left child* dari g , dan $node x$ adalah *left child* dari p)



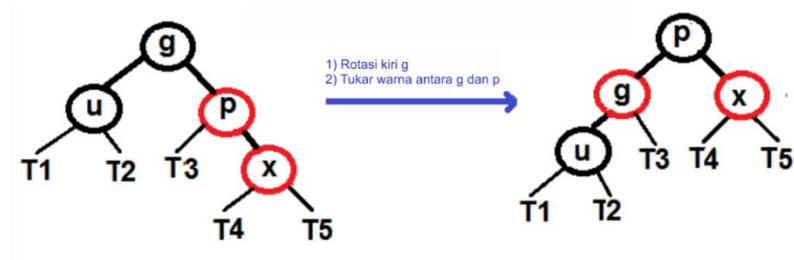
Gambar 16. Metode Kiri Kiri

- ii) Kasus kiri-kanan (p adalah *left child* dari g , dan $node x$ adalah *right child* dari p)



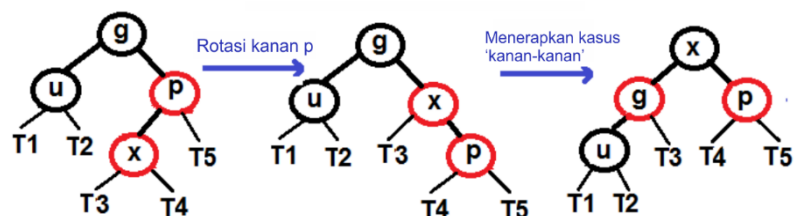
Gambar 17. Metode Kiri Kanan

- iii) Kasus kanan-kanan (cermin kasus i)



Gambar 18. Metode Kanan Kanan

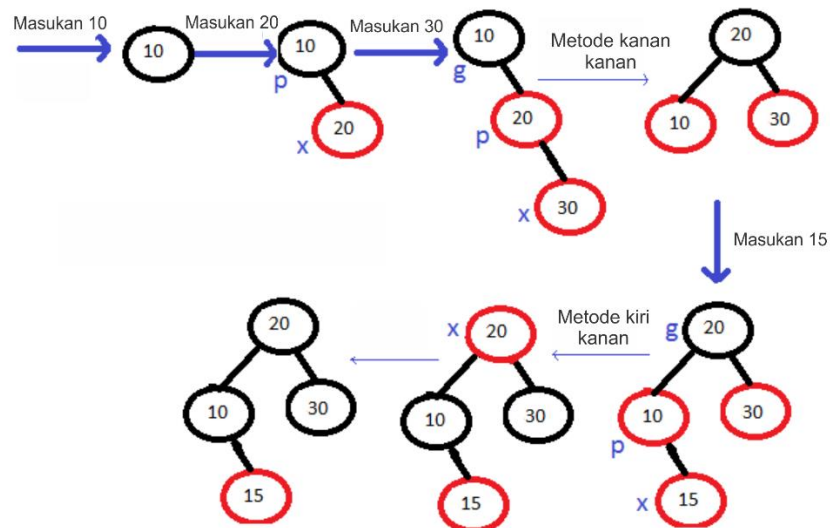
- iv) Kasus kiri-kanan (cermin kasus ii)



Gambar 19. Metode Kanan Kiri

Contoh Penyisipan pada *Red-black Tree*

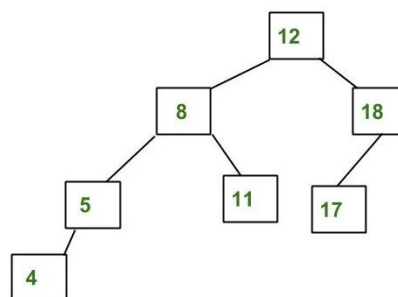
Masukan nilai 10, 20, 30, dan 15 ke dalam pohon kosong



Gambar 20. Contoh Penyisipan *Node* Baru pada Pohon Merah-Hitam

3. AVL Tree

AVL tree adalah *Binary Search Tree* yang memiliki perbedaan tinggi atau level maksimal 1 antara *subtree* kiri dan *subtree* kanan.



Gambar 21. AVL 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.

Kegunaan *AFL tree*:

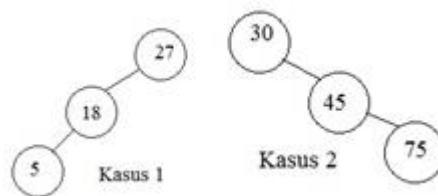
- Waktu pencarian dan bentuk *tree* dapat dipersingkat dan disederhanakan.
- Menyeimbangkan *Binary Search Tree*.

Komponen utama *AFL tree*:

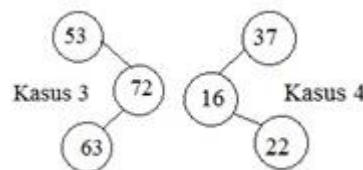
- *Key*: variabel peunik pada model data atau yang umumnya kita implementasikan menjadi *struct node*, berguna sebagai pembeda terhadap *node* lain di *data structure* yang sama.
- *Height/Level*: kalkulasi posisi lokal *node* pada *tree* berdasarkan *base node tree*-nya yang terjauh (*leaf*).
- *Balance Factor*: selisih *height current/local root* (*node* dapat dikatakan *root* karena setiap *node* di *tree* merupakan *subtree*, sehingga keseluruhan *tree* adalah kumpulan *subtree* yang terkoneksi oleh lengannya).

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*).

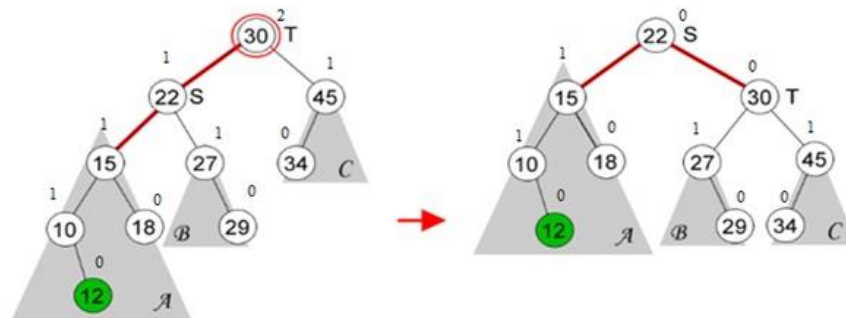


Gambar 22. Contoh Kasus *Single Rotation*



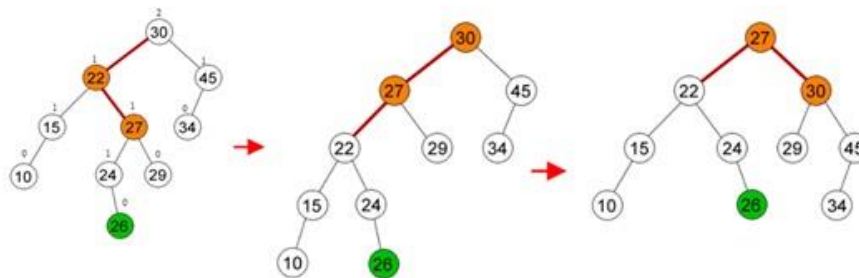
Gambar 23. Contoh Kasus *Double Rotation*

Single Rotation: Jika di-insert node baru dengan nilai 12, maka akan terjadi ketidakseimbangan posisi root.



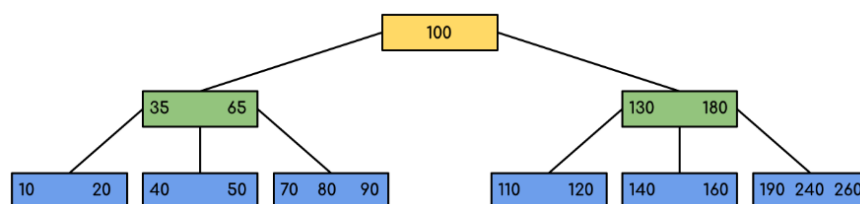
Gambar 24. *Single Rotation*

Double Rotation: Jika di-insert node 26, akan terjadi ketidak seimbangan, sehingga dapat diselesaikan dengan kasus 4.



Gambar 25. *Double Rotation*

4. B-Tree



Gambar 26. B-tree

B-tree merupakan struktur data pohon yang membuat data diurutkan dan memungkinkan pencarian, penyisipan, dan penghapusan dalam penambahan waktu logaritmik.

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.

Kegunaan *B-Tree*:

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

Kompleksitas waktu dari *B-Tree*:

<i>Algorithm</i>	<i>Time Complexity</i>
<i>Search</i>	$O(\log n)$
<i>Insert</i>	$O(\log n)$
<i>Delete</i>	$O(\log n)$

Tabel 2. Kompleksitas Waktu *B-Tree*

Fakta Menarik:

Tinggi minimum *B-tree* yang bisa ada dengan n adalah banyaknya *node* dan m adalah banyaknya maksimum *node child* dari sebuah *node* yang bisa dimiliki adalah:

$$h_{min} = \lceil \log_m(n + 1) \rceil - 1$$

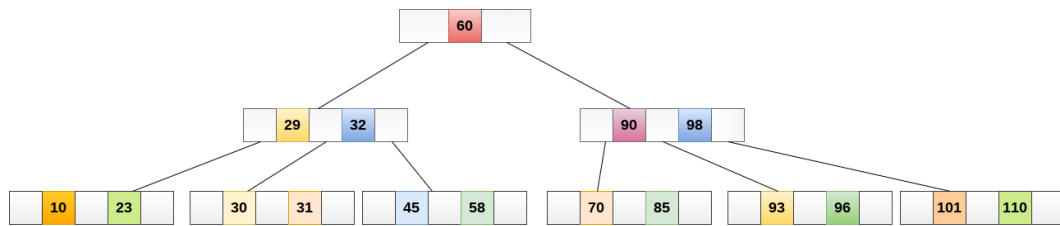
Tinggi maksimum *B-tree* yang dapat ada dengan n adalah banyaknya *node* dan t adalah banyaknya minimum *node child* yang dapat dimiliki oleh *node non-root* adalah:

$$h_{max} = \left\lceil \log_t \frac{n+1}{2} \right\rceil \text{ dan } t = \left\lceil \frac{m}{2} \right\rceil$$

Misalkan struktur data *B-tree* memiliki orde m , sehingga ciri-ciri dari *B-tree* tersebut adalah sebagai berikut.

- Setiap *node* dalam *B-tree* berisi paling banyak m *node child*.
- Setiap *node* dalam *B-Tree*, kecuali *node root* dan *node leaf* berisi setidaknya $m/2$ anak. (Tidak perlu semua *node* berisi jumlah *node child* yang sama)
- *B-tree* didefinisikan dengan istilah derajat minimum t .
- Setiap *node* kecuali *root* harus berisi setidaknya $t - 1$ *key*. *Node root* berisi minimal 1 *key*.
- Semua *node* (termasuk *root*) dapat berisi paling banyak $2t - 1$ *key*.
- Banyaknya *node child* dari sebuah *node* sama dengan banyaknya *key* di dalamnya ditambah 1.
- *Node root* harus memiliki minimal 2 *node*.
- Semua *node leaf* pada sebuah *B-tree* harus berada pada level yang sama.

B-tree berorde 4 ditunjukkan pada gambar berikut.



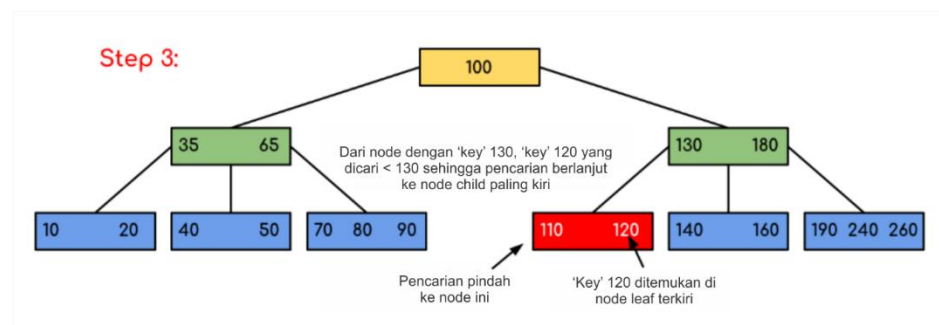
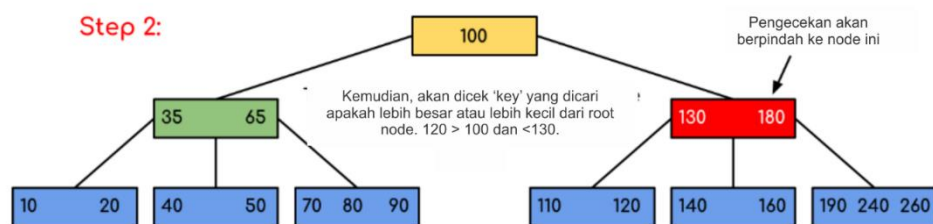
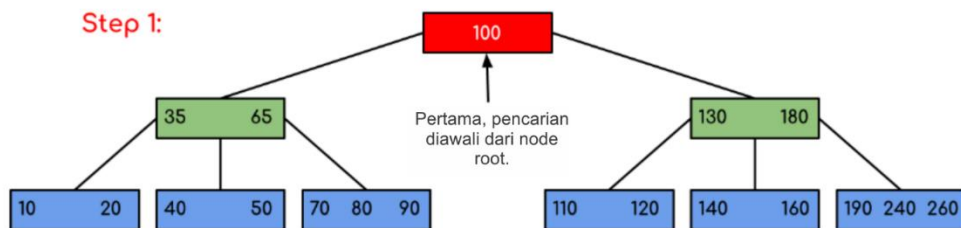
Gambar 14. B-tree berorde empat

Operasi

- Search

Pencarian di B-tree mirip dengan pencarian pada *binary search tree* (BST).

Misalnya, jika kita mencari item 120 di B-tree berikut.



Gambar 15-17. Langkah Pencarian pada B-Tree

- *Insert*

- a. Inisialisasi x sebagai *root*.
- b. Jika x bukan *node leaf*, lakukan langkah di bawah berikut:
 - i. Temukan *node child* dari x yang akan di-*traverse*. Ubah *node child* menjadi y .
 - ii. Jika y belum penuh, ubah x untuk terhubung ke y .
 - iii. Jika y penuh, pisahkan dan ubah x untuk terhubung ke salah satu dari dua bagian y . Jika k lebih kecil dari *mid-key* pada y , maka atur x sebagai bagian pertama dari y . Jika k tidak lebih kecil, maka x bagian kedua dari y . Saat kita membagi y , kita memindahkan kunci dari y ke induknya x .
- c. Perulangan pada langkah 2 berhenti jika x adalah *node leaf*. x harus memiliki ruang untuk satu *key* tambahan karena telah dipisahkan semua *node* sebelumnya. Sehingga cukup memasukkan k ke x .

Contoh 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 di B-tree.

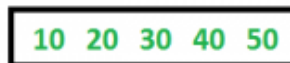
- 1) Awalnya *root* adalah *null*. Masukkan 10.

Insert 10



- 2) Semuanya akan dimasukkan ke dalam *root* karena banyaknya *key* maksimum

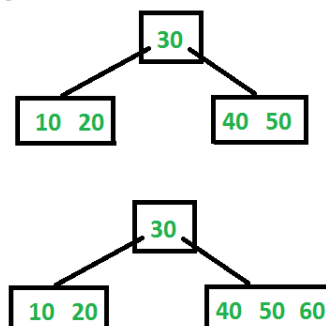
Insert 20, 30, 40 and 50



yang dapat ditampung oleh sebuah *node* adalah $2 * t - 1$ yaitu 5.

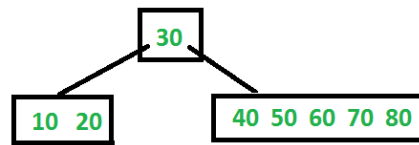
- 3) Karena *root node* sudah penuh, pertama-tama akan dibagi menjadi dua, kemudian 60 akan dimasukkan ke dalam *node child* yang sesuai.

Insert 60



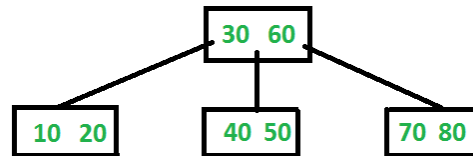
- 4) Dua *key* baru ini akan disisipkan ke *node leaf* yang sesuai tanpa operasi *splitting* B-tree.

Insert 70 and 80



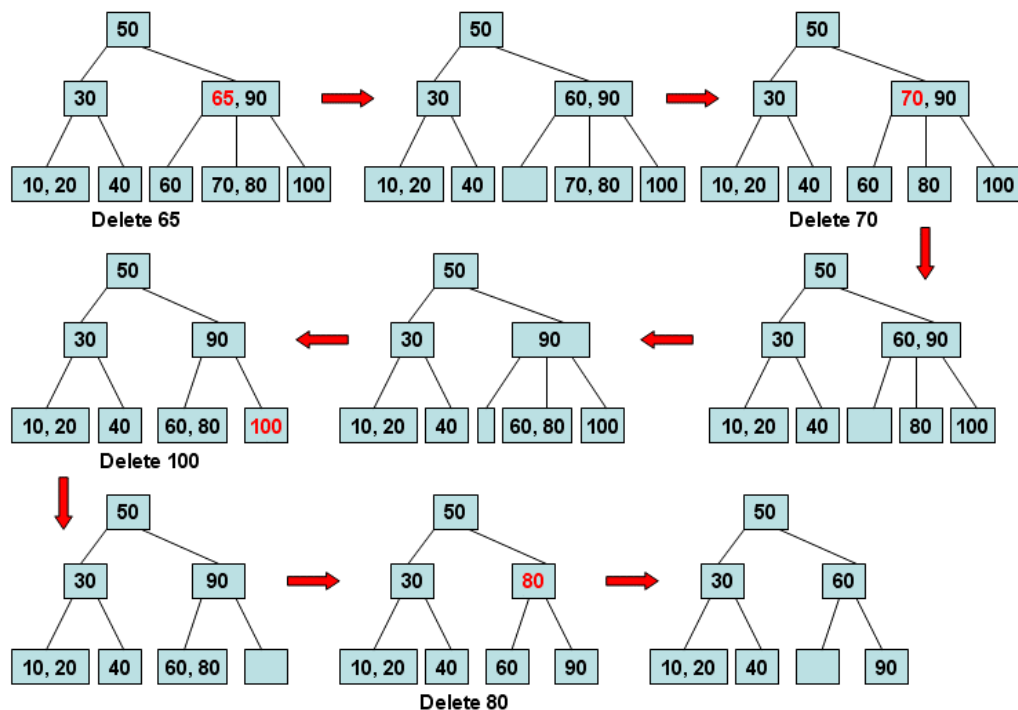
- 5) Penyisipan ini melibatkan operasi *splitting* B-tree. *Key* tengah kemudian akan naik ke induknya.

Insert 90



Gambar 18-22. Langkah Penyisipan pada B-Tree

- Delete



Gambar 23. Langkah Penghapusan pada B-tree

5. Antrian Berprioritas

Antrian berprioritas adalah kumpulan dari kosong atau banyak elemen, setiap elemen mempunyai prioritas atau nilai. Urutan penghapusan dari antrian berprioritas

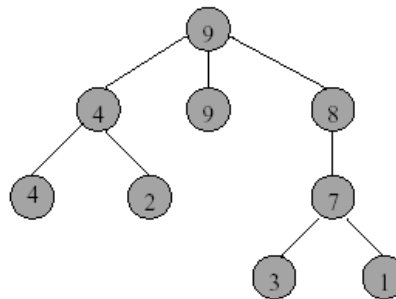
ditentukan berdasar prioritas elemen, bukan berdasar urutan masuk elemen dalam antrian.

Implementasi antrian berprioritas:

- *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

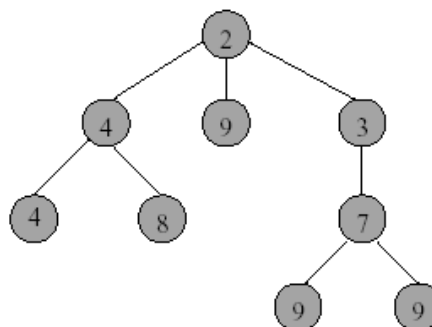
Max tree adalah pohon yang nilai setiap *node*-nya lebih besar atau sama dengan nilai *child* (jika punya), sehingga *root node* memiliki nilai terbesar. *Max heap* adalah *max tree* yang juga sebuah *complete binary tree*.



Gambar 24. Max Heap

Min Tree

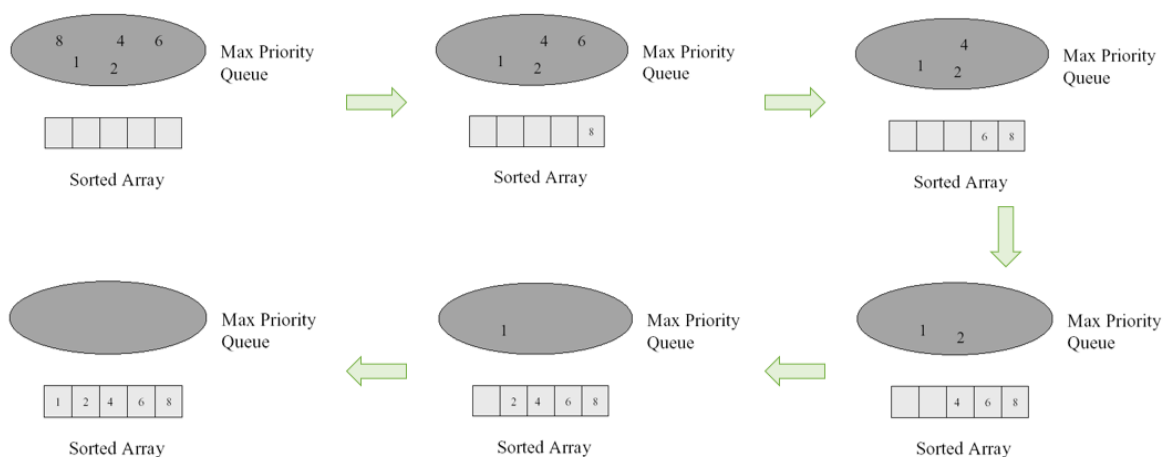
Min tree adalah pohon yang nilai setiap *node*-nya lebih kecil atau sama dengan nilai *child* (jika punya), sehingga *root node* memiliki nilai terkecil. *Min heap* adalah *min tree* yang juga sebuah *complete binary tree*.



Gambar 25. Min Heap

- a. Penyisipan dalam *max heap*:
 - 1) Sisipkan elemen baru ke *leaf node* terakhir.
 - 2) Apabila *parent node* bernilai lebih kecil dari elemen baru, tukar posisi elemen baru dengan *parent node* tersebut.
 - 3) Terus bandingkan nilai elemen baru dengan *parent node* hingga *root node*, apabila elemen baru bernilai lebih besar, tukar posisinya.
- b. Penghapusan elemen *max root node* dalam *max heap*:
 - 1) Hapus elemen *max root node*.
 - 2) Sisipkan elemen *leaf node* terakhir ke elemen *max root node*.
 - 3) Terus bandingkan nilai elemen *max root node* dengan *child node*, apabila nilai *child node* lebih besar, tukar posisinya.
- c. Pengurutan dalam *heap*:
 - 1) Letakkan elemen yang akan diurutkan dalam antrian berprioritas.
 - 2) Jika *min priority queue* yang digunakan, elemen diekstrak dengan urutan prioritas naik.
 - 3) Jika *max priority queue* yang digunakan, elemen diekstrak dengan urutan prioritas menurun.

Berikut merupakan contoh *max priority queue*.



Gambar 26. *Max Priority Queue*

REFERENSI

Persada, Anugerah Galang; Hernanda, Henoeh; 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/TIstrukdat10.ppt