Jurnal TIMES , Vol. IV No 2 : 61-65 , 2015
ISSN : 2337 - 3601

METODE AVL TREE UNTUK PENYEIMBANGAN TINGGI BINARY TREE

Suwanty[1] Octara Pribadi[2]
Program Studi Teknik Informatika[1,2]
STMIK TIME[1,2]
Jalan Merbabu No. 32 AA-BB Medan[1,2]
e-mail : dharma_suwanty@gmail.com[1] octarapribadi@gmail.com[2]

Abstrak

Binary tree (pohon biner) merupakan struktur data yang dimanfaatkan untuk melakukan operasi pencarian data dalam waktu konstan O(1), namun jika data yang dimasukkan kedalam pohon tidak seimbang (*imbalance*) maka akan meningkatkan kompleksitas waktu dalam proses pencarian data. AVL tree adalah metode yang digunakan untuk otomatisasi penyeimbangan (*self-balance*) tinggi pohon sehingga dapat menjamin waktu yang dibutuhkan untuk mencari data, bernilai konstan atau O(1).
Kata Kunci : AVL Tree, Binary Search Tree, Data Structure

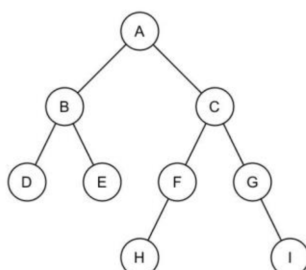
1. Pendahuluan

Dalam ilmu komputer, pohon biner merupakan salah satu struktur data pohon (*tree*) yang memiliki paling banyak 2 anak (*child*) [1]. Pohon biner memiliki fungsi-fungsi dinamis seperti *search, insert, delete, maximum, minimum, successor,* dan *predecessor*. Oleh sebab itu pohon biner dapat dimanfaatkan sebagai struktur data untuk melakukan pencarian atau biasa disebut *binary search tree*. Dengan *complete binary tree* dengan jumlah node n, data dapat dicari dalam waktu O(lg n) untuk *worst case*. Namun jika pohon biner tersusun *linear* maka butuh waktu O(n) untuk *worst case*. Sehingga dalam melakukan *insert* data kedalam pohon biner harus dilakukan sedemikian rupa sehingga pohon dapat berada dalam keadaan seimbang (*balance*).

AVL *tree* atau pohon AVL merupakan *self-balancing binary search tree* yang dapat melakukan pengulangan penyeimbangan pohon biner (*rebalancing*), sehingga ketinggian (*height*) antara dua anak semua simpul memiliki perbedaan paling banyak sebesar satu. Penelitian ini bertujuan untuk membuktikan bahwa AVL *tree* dapat digunakan untuk memastikan pohon biner dalam keadaan seimbang.

2. Landasan Teori

Pohon (Tree) adalah graf terhubung yang tidak mengandung sirkuit. Karena merupakan graf terhubung maka pada pohon selalu terdapat path atau jalur yang menghubungkan kedua simpul di dalam pohon. Pohon dilengkapi dengan *Root (akar)*. Contoh Pohon berakar T



Gambar 1. Pohon Biner T

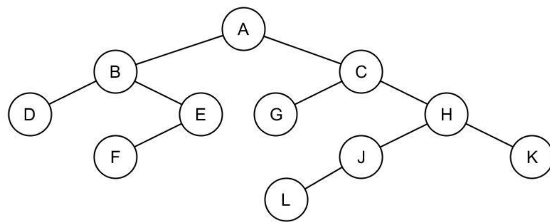
Sifat utama pohon berakar :

1. Jika pohon mempunyai simpul (*node*) sebanyak n, maka banyaknya ruas (*edge*) adalah (n-1). Pada contoh : banyak simpul adalah maka banyaknya ruas adalah 8.
2. Mempunyai simpul khusus yang disebut Root (Akar), jika simpul tersebut memiliki derajat keluar ≥ 0 dan derajat masuk = 0. Simpul A merupakan root.
3. Mempunyai simpul yang disebut Leaf (Daun), jika simpul tersebut memiliki derajat keluar = 0 dan derajat masuk = 1. Simpul D, E, H, I merupakan daun pada pohon T.
4. Setiap simpul mempunyai tingkatan (level), dimulai dari root dengan level 0 sampai dengan level n pada daun yang paling bawah.
   Pada contoh :
   A mempunyai level 0
   B, C mempunyai level 1
   D, E, F, G mempunyai level 2
   H, I mempunyai level 3
   Simpul yang mempunyai level yang sama disebut bersaudara (*brother* atau *sibling*)
5. Pohon mempunyai ketinggian (kedalaman / height) yaitu level tertinggi +1. Ketinggian pohon T adalah 3+1 = 4.
   Pohon mempunyai berat (bobot / weight) yaitu banyaknya daun pada pohon. Berat pohon T adalah 4.

Pohon biner (*binary tree*)

Pohon biner adalah himpunan simpul yang terdiri dari 2 subpohon (yang disjoint / saling lepas) yaitu subpohon kiri dan subpohon kanan. Setiap simpul dari pohon biner mempunyai derajat keluar maksimum = 2. Pendefinisian pohon biner bersifat rekursif. Pohon biner sering disajikan dalam bentuk diagram.

Contoh pohon biner A:



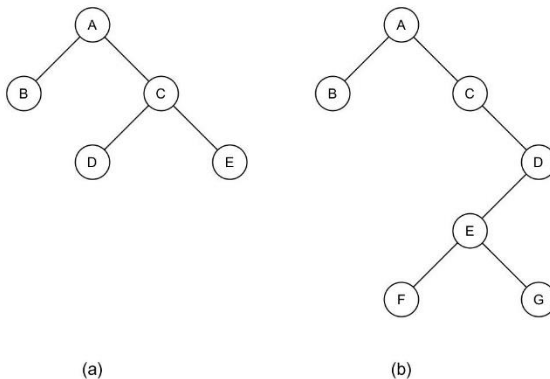
Gambar 2. Pohon Biner A

Untuk menggambarkan suksesor kiri dan suksesor kanan, dibuat garis ke kiri bawah dan ke kanan bawah. B adalah suksesor kiri dari A, sedangkan C adalah suksesor kanan dari A. Subpohon kiri dari A mengandung simpul B, D, E dan F, sedangkan subpohon kanan mengandung simpul C, G, H, J, K dan L. Pada gambar 2, terdiri dari

1. Root adalah A.
2. Simpul yang mempunyai 2 anak adalah simpul A, B, C dan H.
3. Simpul yang mempunyai 1 anak adalah simpul E dan J.
4. Simpul yang tidak mempunyai anak disebut daun (terminal) adalah D, F, G, K dan L.

Pohon Ketinggian Seimbang

Pohon biner yang mempunyai sifat bahwa ketinggian subpohon kiri dan subpohon kanan dari pohon tersebut berbeda paling banyak 1, disebut Pohon Ketinggian Seimbang atau *Height Balanced Tree* (HBT). Contoh :



Gambar 3. (a) Pohon biner seimbang (b) Pohon biner tidak seimbang

Ketinggian Minimum dan Maksimum Pohon Biner

Jika banyaknya simpul = N, maka :

1. Ketinggian Minimum adalah : $H_{min} = INT(^2log\ N) + 1$
2. Ketinggian Maksimum adalah : N

Contoh : untuk N = 8

Ketinggian Minimum adalah :

H_{min} = INT(2log N) + 1
= INT(2log 8) + 1
= INT(2log 23) + 1
= INT(3) + 1
= 3 + 1 = 4

Ketinggian Maksimum adalah : 8

Binary Search Tree

Kelemahan dari Binary Tree adalah tidak efisien dalam pencarian (*searching the target node*), yang harus dilakukan secara sequensial dari mulai *root* sampai ke *Node* yang dikehendaki.

Selain juga penghapusan terhadap Node Binary Tree tidak dapat dilakukan, yang dapat dilakukan penghapusan 1 sub tree. Hal ini disebabkan karena pada Binary Tree node tidak diurut. Binary Search Tree mengeliminir kelemahan tersebut diatas, dengan cara mengurutkan node yang ada.

Ketentuan peletakkan node dalam binary search tree adalah sebagai berikut :

1. Semua Left Child harus lebih kecil dari pada Parent dan Right Child
2. Semua Right Child harus lebih kecil dari pada Parent dan Left Child

Keuntungan dari Binary Search Tree dibandingkan dengan Binary Tree adalah

1. Searching / pencarian target node menjadi lebih efisien dan cepat dibandingkan dengan Binary Tree.
2. Penghapusan dapat dilakukan terhadap target node, bukan terhadap 1 subtree seperti pada Binary Tree.

Semua operasi dalam Binary Tree bisa di implementasikan langsung pada Binary Search Tree, kecuali :

1. Insert
2. Update
3. DeleteKey

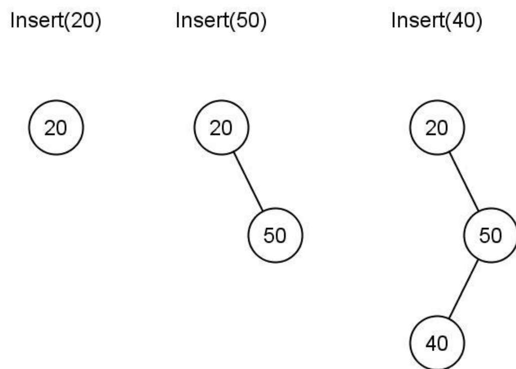
Karena operasi diatas dapat mengakibatkan Binary Search Tree tidak urut. Sehingga harus dilakukan modifikasi terhadap posisi node agar Binary Search Tree tetap terurut.

Operasi Insert

Operasi insert pada Binary Search Tree dilakukan dengan cara mencari lokasi untuk node yang akan di insert. Pencarian selaku dimulai dari root, jika node yang akan diinsert ternyata lebih kecil dari root, maka akan dilakukan insert pada left sub tree sedangkan bila node yang akan diinsert lebih besar dari root, maka akan dilakukan insert pada right sub tree.

Pencarian lokasi untuk node diteruskan sampai memenuhi kondisi Binary Search Tree yaitu semua node yang berada pada left sub tree lebih kecil dari parentnya, sedangkan yang berada pada right sub tree harus lebih besar dari parentnya.

Berikut ini akan dilakukan operasi insert, yang langsung akan digambarkan Binary Search Tree setelah dilakukan operasi insert.



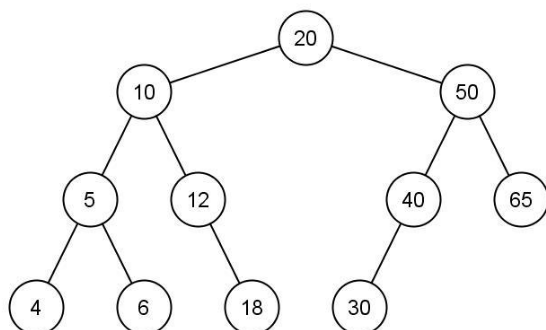
Gambar 4. Operasi Insert Pada Binary Search Tree

Operasi DeleteKey

Berbeda dengan Binary Tree dimana penghapusan harus dilakukan terhadap sebuah sub tree, maka pada Binary Seacrh Tree, penghapusan dapat dilakukan terhadap sebuah node (key) tertentu. Operasi deletekey dapat mengakibatkan Binary Search Tree menjadi tidak urut lagi, sehingga perlu dilakukan rotasi terhadap Binary Search Tree tersebut agar menjadi urut kecuali, beberapa posisi node pada saat dilakukan operasi deletekey adalah :

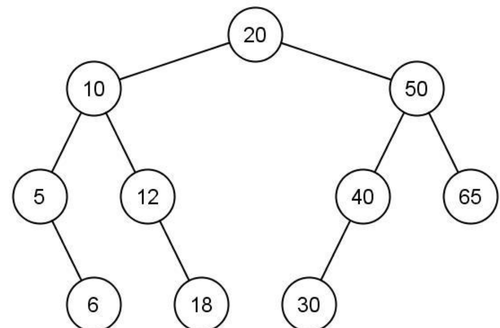
1. Node yang dihapus adalah leaf, sehingga penghapusan akan tetap membuat Binary Search Tree terurut. Bila yang terjadi adalah ini, maka operasi penghapusan dapat langsung dilakukan.
2. Node yang dihapus adalah node yang memiliki node 1 child, sehingga child yang bersangkutan dapat langsung dipindahkan untuk menggantikan posisi node yang dihapus.
3. Node yang akan dihapus memiliki 2 children (2 subtree), maka node yang diambil untuk menggantikan posisi node yang dihapus adalah :
   a. Bisa berasal dari left sub tree, dimana node yang diambil adalah node yang mempunyai nilai paling besar (yang berada pada posisi paling kanan).
   b. Bisa berasal dari Right Sub Tree, dimana node yang diambil adalah node yang mempunyai nilai paling kecil (yang berada pada posisi paling kiri).

Dengan menggunakan gambar Binary Search Tree berikut ini ,akan diilustrasikan operasi deletekey untuk masing-masing kondisi di atas.



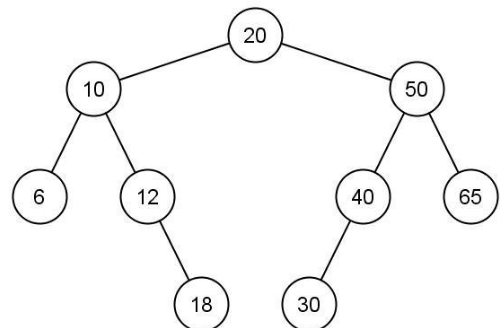
Gambar 5. Binary Search Tree dengan ketinggian 4

Posisi 1, Jika yang didelete adalah LEAF, maka tidak perlu dilakukan modifikasi terhadap lokasi contoh :



Gambar 6. Operasi Delete(4)

Posisi 2, jika yang didelete adalah node yang hanya memilki 1 child, maka child tersebut langsung menggantikan parentnya contoh :



Gambar 7. Operasi Delete (5)

Posisi 3, jika yang didelete adalah node dengan 2 children (2 Subtree), maka node yang diambil untuk menggantikan posisi node yang dihapus adalah :

1. Berasal dari left sub tree, yang diambil adalah node yang paling kanan (yang mempunyai nilai yang terbesar)
2. Atau dari right sub tree, yang diambil adalah node yang paling kiri (yang mempunyai nilai yang terkecil).

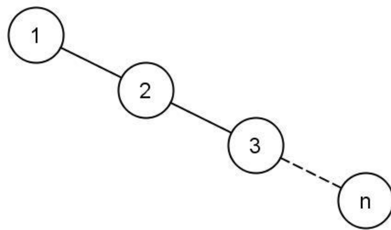
3. Metode Penelitian

Walaupun Binary SearchTree sudah dapat mengatasi kelemahan pada Binary Tree dengan cara mengurutkan / sort node yang di-insert, di-update dan di-delete, tetapi masih ada kendala lain yang dihadapai binary search tree , yaitu masih ada kemungkinan terbentuk skewed binary tree (tree

miring), yang mempunyai perbedaan height (height-balanced) antara subtree kiri dengan subtree kanan.

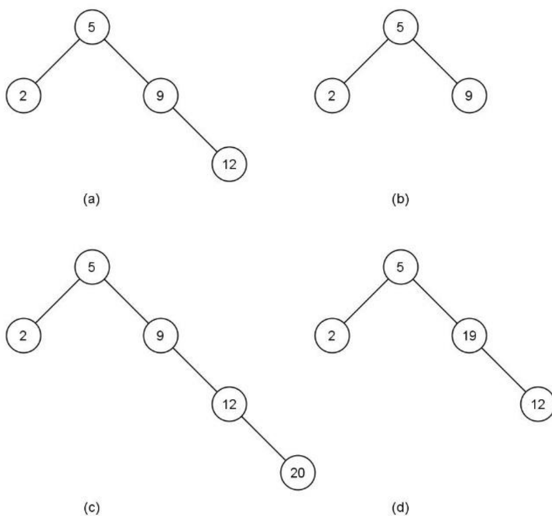
AVL (Adelson, Velskii dan Landis) Tree mengatasi hal ini dengan cara membatasi height-balanced maksimum 1. AVL Tree dapat didefinisikan sebagai Binary Search Tree yang mempunyai ketentuan bahwa “maksimum perbedaan height antara subtree kiri dan subtree kanan adalah 1”.

AVL Tree juga sering disebut dengan height-balanced 1-tree. Gambar dibawah ini memperlihatkan Binary Search Tree setelah dilakukan operasi insert sebagai berikut : +1, +2, +3,.................+n.



Gambar 8. Operasi Insert pada Binary Search Tree

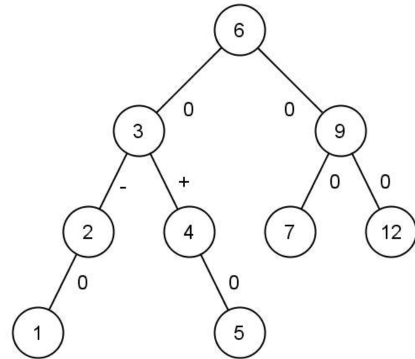
Bila dilakukan pencarian terhadap node n diatas, maka pencarian sama seperti dilakukan pada Binary Search Tree, yaitu pencarian menjadi sekuensi, yang memakan waktu lama, hal ini tidak mungkin terjadi pada AVL Tree karena perbedaan height dibatasi maksimal hanya 1. berikut ini adalah contoh ATL Tree dan yang bukan AVL Tree.



Gambar 9. (a) dan (b) AVL Tree sedangkan (c) bukan AVL tree karena perbedaan tinggi >1 dan (d) bukan Binary Search Tree

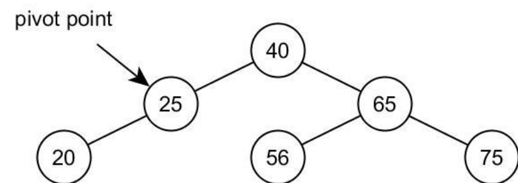
Setiap node dalam AVL Tree diberikan simbol untuk mengetahui tentang statusnya (lihat gambar dibawah ini), yaitu :

1. Node diberi simbol – dan disebut TallLeft bila sub tree kiri lebih panjang dari Sub Tree kanan.
2. Node diberi simbol + dan disebut TallRight bila sub tree kanan lebih panjang dari subtree kiri.
3. Node diberi simbol 0 dan disebut Balance bila subtree kiri dan kanan mempunyai height yang sama.



Gambar 10. Simbol pada AVL Tree

Path pencarian lokasi untuk penentuan lokasi elemen pada operasi INSERT disebut dengan Searching Path. Bila node pada search Path yang balancenya TallLeft (tanda-) tau TallRight (tanda +) dan terletak paling dekat dengan node yang baru maka node tersebut dinamakan Pivot Point. Gambar di bawah ini menunjukan pivot point :

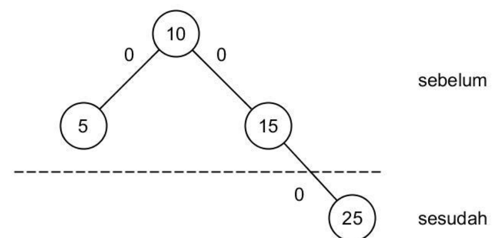


Gambar 11 Pivot Point AVL Tree

Operasi Insert

Agar AVL Tree dapat tetap mempertahankan Height-Balanced 1-Tree maka setiap kali pelaksanaan operasi insert, jika diperlukan maka harus dilakukan rotasi. Operasi insert dalam AVL Tree ada 3 kondisi / kasus , yaitu :

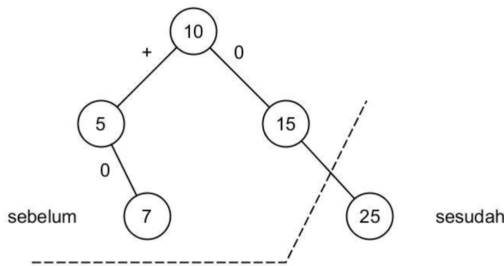
1. Case 1
   Tidak ada Pivot Point dan setiap node adalah balance, maka bisa langsung diinsert sama seperti pada Binary Search Tree (tanpa perlu diregenerate).



Gambar 12. Operasi Insert Case 1

2. Case 2

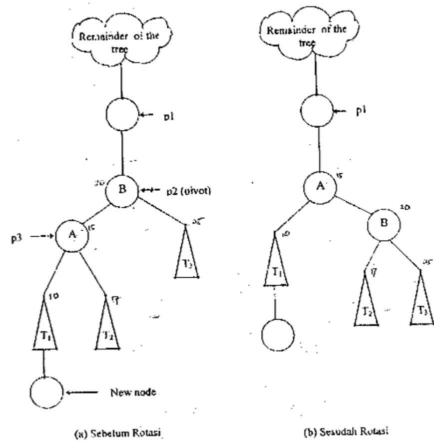
Jika ada Pivot Point tetapi subtree yang akan ditambahkan node baru memiliki height yang lebih kecil, maka bisa langsung diinsert.



Gambar 13. Operasi Insert Case 2

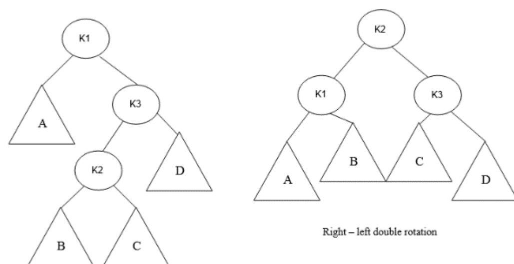
3. Case 3

   Jika ada Pivot point dan subtree yang akan ditambahkan node baru memiliki height yang lebih besar, maka tree harus diregenerate , supaya tetap menghasilkan AVL Tree. Cara melakukan re-generate adalah dengan melakukan :
   a. Single Rotation
   b. Double Rotation



Gambar 14. Operasi Insert Single Rotation

Pada beberapa kasus di AVL tree dapat diselesaikan hanya dengan cara single rotation, seperti pada contoh diatas, tetapi ada pula yang tidak dapat diselesaikan hanya dengan single rotation, itu artinya harus menggunakan teknik double rotation untuk menyelesaikan masalah *imbalance* pada tree.



Gambar 15. Operasi Insert Double Rotation

Operasi Delete

Operasi delete dalam AVL Tree adalah sama dengan operasi Deletekey dalam Binary Search Tree (3 kasus yang bisa terjadi). Yang harus diperhatikan adalah harus diusahakan agar pohon asli Delete tetap berupa AVL-Tree yaitu dengan melakukan rotasi.

4. Kesimpulan

Dari penelitian ini didapatkan kesimpulan bahwa :

1. Dengan menggunakan AVL Tree maka pohon biner dapat melakukan otomatis penyeimbangan.
2. AVL Tree dapat membuat perbedaan ketinggian pohon memiliki selisih tinggi maksimal 1.

5. Daftar Pustaka

[1] Daniel F Stubbs & Neil W. Webre. 1985. Data Structures with Abstract Data Type and Pascal. Brook / Cole Publishing Company

[2] Aaron M. Tanenbaum, Yedidyah Langsam, Moshe J Augenstein. 1990. Data Structures Using C, Prentice Hall.

[3] Maria, Anna. 1998. Struktur Data. Buku Ajar Universitas Bina Nusantara Jakarta.

[4] Weiss, Mark Allen. 2006. Data Structures and Algorithm Analysis in C++. Addison Wesley.