Penerapan Teori Pohon Dalam Kajian Struktur Data

Khoirush Sholih Ridhwaana Akbar
NIM : 13505120
Program Studi Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung
E-mail : if15120@students.if.itb.ac.id

Abstrak

Makalah ini membahas tentang penerapan teori pohon dalam kajian struktur data dimana beberapa penerapannya merupakan salah satu struktur penyimpanan data terbaik.

Salah satu penerapan teori pohon yang paling berguna dan dipakai yaitu konsep *binary search tree* dimana konsep ini memberikan struktur data yang memudahkan operasi pencarian, penambahan, dan penghapusan terhadap data. Operasi tersebut lebih efisien dan jauh lebih baik pada konsep ini dibanding *sequential search* pada senarai berkait dalam waktu eksekusi / *run-time*.

Dari konsep *binary search tree* ini dikembangkan lagi suatu struktur penyimpanan data yang merupakan modifikasi dari *binary search tree* tersebut yaitu *AVL-Tree* dan *Splay Tree* yang masing-masing mempunyai keunggulan pada kasus tertentu yang sekarang ini sering dijumpai.

AVL-Tree merupakan modifikasi *binary search tree* yang tinggi setiap upapohon kiri dan upapohon kanan sama atau setidaknya selisih antara keduanya tidak lebih dari 1. Keunggulan dari *AVL-Tree* antara lain untuk mengoptimasi pencarian data terutama untuk kasus pohon yang condong ke kiri atau ke kanan sehingga pencarian akan jauh lebih mudah apabila pohon tersebut seimbang. Kasus pohon yang condong ke kiri atau kanan itu mungkin saja terjadi terutama apabila penambahan elemen dan penghapusan elemen dilakukan terus-menerus dan tidak dapat diketahui urutannya.

Sedangkan *Splay-Tree* justru kebalikan dari AVL-Tree yang tidak mempermasalahkan kecondongan upapohonnya namun setiap kali data diakses maka simpul dari data yang diakses tersebut akan dinaikkan keatas mendekati akar pohon. Data yang sering diakses / aktif akan berada dekat pada akar pohon sehingga data tersebut mudah diakses (waktu yang diperlukan untuk mengakses data menjadi singkat). Konsep ini sangat berguna untuk struktur data jaringan rumah sakit.

Dengan demikian dapat disimpulkan bahwa penerapan teori pohon sangatlah bermanfaat dalam kajian struktur data.

Kata Kunci : *Binary Search Tree, AVL-Tree, Splay Tree, run-time, balance factor, record.*

1. Pendahuluan

Teori pohon merupakan salah satu teori yang cukup tua karena sudah dikenal sejak tahun 1857, dimana ketika itu matematikawan Inggris Arthur Cayley menggunakan teori pohon ini untuk menghitung jumlah senyawa kimia.[1]

Teori pohon ini sebenarnya adalah suatu mekanisme penyelesaian suatu masalah dengan menganalogikan permasalahan tersebut kedalam struktur pohon untuk memudahkan pencarian solusi masalah tersebut. Teori pohon ini juga merupakan salah satu penerapan konsep graf. Dimana pohon itu dapat didefinisikan sebagai graf tak-berarah terhubung yang tidak mengandung sirkuit. [1]

Kajian struktur data merupakan kajian yang sangat penting dalam bidang informatika. Dan di zaman sekarang ini yang teknologinya semakin berkembang, dibutuhkan struktur data yang efisien yang dapat meningkatkan kinerja program.

Teori pohon ini merupakan teori yang sangat berguna dalam struktur data dimana aplikasi-aplikasi dari teori pohon ini dapat dijadikan

struktur penyimpanan data yang sangat baik dalam kasus tertentu yang mana kasus tersebut sudah umum ditemui sekarang ini.

Oleh karena itu dalam makalah ini akan dijelaskan beberapa aplikasi teori pohon yang dipakai untuk membentuk suatu struktur penyimpanan data yang efisien.

2. Struktur Penyimpanan Data

Terdapat 2 garis besar struktur penyimpanan data yang telah diimplementasikan secara global yaitu penyimpanan secara statik dan dinamik.

2.1 Penyimpanan Data Statik

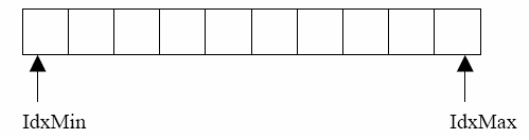
Penyimpanan data secara statik merupakan penyimpanan data dimana memori yang disiapkan untuk dipakai sudah ditentukan dari awal. Implementasi dari penyimpanan data seperti ini lebih kepada struktur data yang menggunakan tabel kontigu. Penyimpanan data dengan cara seperti ini tentunya memiliki kelebihan dan kekurangan.

Kelebihan dari penyimpanan data statik ini adalah data dapat diakses langsung asalkan kita mengetahui diindeks keberapa data disimpan atau dapat juga disebut pengambilan data (*data retrieval*) lebih mudah.

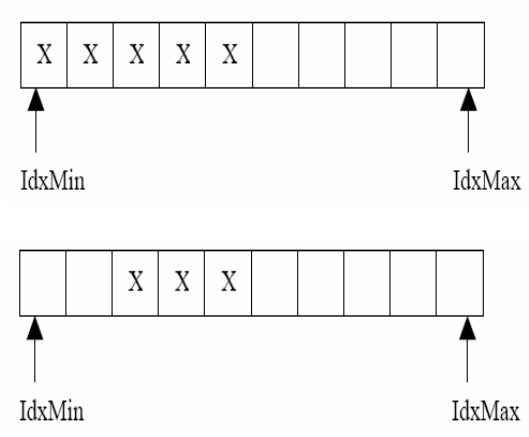
Kekurangan dari penyimpanan data statik ini adalah kita tidak tahu apakah tabel terisi penuh atau tidak dan kita juga tidak tahu batas indeks yang dipakai dan juga elemen tabel hanya bisa sejenis, tidak bisa terdapat jenis elemen yang berbeda pada sebuah tabel kontigu. Dan juga walaupun data masukan tidak ada atau tabel kosong, memori yang dipakai tetap sama dengan tabel penuh, sehingga dapat dikatakan penyimpanan data secara statik ini boros dalam pemakaian memori.

Dengan melihat kekurangan dari penyimpanan data statik ini dapat ditarik kesimpulan bahwa penggunaan penyimpanan data secara statik ini sangat tidak bagus untuk kasus dimana data masukan berbeda tipenya dan tidak diketahui jumlah maksimum data yang akan ditampung.

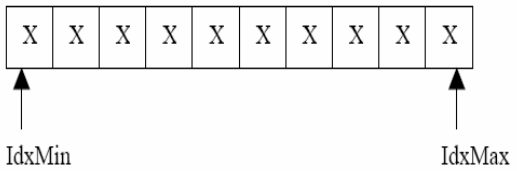
Sehingga kita akan mengalami kesulitan dalam mendeklarasikan tabel kontigu yang dapat menampung data masukan yang masih belum diketahui secara jelas dan bisa bermacam-macam jenisnya.



Gambar 1. Tabel Kontigu Kosong



Gambar 2. Tabel Kontigu Terisi Sebagian



Gambar 3. Tabel Kontigu Terisi Penuh

Tentunya dari penjelasan tadi kita dapat mengetahui kasus apa yang cocok untuk memakai penyimpanan data statik ini.

Ya, penyimpanan data statik ini sangat cocok dipakai untuk data yang statik yang tidak akan diubah-ubah dan hanya digunakan sebagai *resource data*. Misalnya untuk sebuah *database* yang tidak akan diubah nilainya dan hanya untuk dipakai saja pada proses-proses lainnya. [2]

2.2 Penyimpanan Data Dinamik

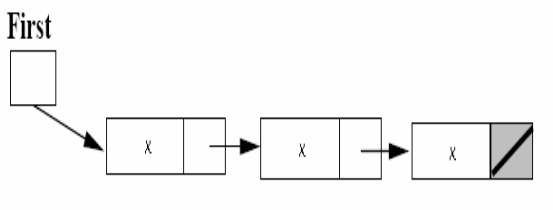
Penyimpanan data secara dinamik lebih banyak dipakai daripada penyimpanan data statik karena kasus yang lebih baik menggunakan penyimpanan data dinamik ini jauh lebih banyak daripada penyimpanan data statik. Hal ini

dikarenakan pada penyimpanan data dinamik mempunyai beberapa keunggulan yang sangat kontras dibanding penyimpanan data statik antara lain :

1. Pada penyimpanan data dinamik pemesanan memori hanya dilakukan saat ada data masukan yang baru atau dengan kata lain program tidak akan meminta memori pada komputer apabila tidak ada data masukan sehingga untuk data kosong tidak akan memakai memori sama sekali
2. Tipe setiap elemen dapat diatur sedemikian sehingga tipe data setiap elemennya dapat berbeda-beda.

Penyimpanan data secara dinamik ini umumnya diimplementasikan dengan senarai berkait (*linked list*). Dimana sebuah list itu mempunyai sebuah penunjuk elemen pertama (*First*) list yang dijadikan sebagai acuan sebuah list tersebut kosong atau berisi. [2]

Berikut ini gambar dari kondisi list yang terisi dan kosong :



Gambar 4. Senarai Berkait dengan 3 elemen

Gambar 5. Senarai Berkait Kosong

Namun implementasi dengan senarai berkait ini memiliki kekurangan dimana untuk operasi pencarian, penambahan, dan penghapusan data kita harus menelusuri elemennya dari elemen pertama. Untuk program yang berskala menengah kebawah mungkin ini bukan menjadi suatu permasalahan yang genting. Namun untuk suatu program atau software yang berskala besar yang membutuhkan kecepatan eksekusi (*run-time*) yang tinggi untuk setiap perintah-perintah yang ada, hal ini tentunya merupakan suatu permasalahan yang harus dipandang serius. Karena untuk program skala besar ini kompleksitas algoritma sangat memegang peran penting dalam kecepatan eksekusi. Untuk itu diperlukan suatu cara baru untuk mengimplementasikan penyimpanan data dinamik dengan kecepatan eksekusi yang tinggi.

Disinilah peran penting penerepan teori pohon pada kajian struktur data sangat terasa efeknya. Penerapan teori pohon yang dimaksud adalah konsep *Binary Search Tree* yang memberikan kecepatan eksekusi yang lebih tinggi dari pada implementasi senarai berkait terutama pada operasi pencarian, penambahan, dan penghapusan data. Namun selain konsep *Binary Search Tree* masih banyak lagi implementasi penerapan teori pohon yang merupakan pilihan struktur penyimpanan data yang merupakan solusi dari masalah-masalah yang dihadapi dii bidang informatika saat ini antara lain *AVL-tree, Splay-tree, Lexicographic Search Tree / tries , External Search Tree / B-tree, dsb.* [3]

3. Implementasi Teori Pohon

Implementasi teori pohon merupakan salah satu pilihan struktur data terbaik yang pernah ada dan masing-masing implementasi tersebut tentunya merupakan pilihan terbaik untuk kasus-kasus tertentu. Berikut beberapa contoh implementasi dari teori pohon :

1. *Binary Search Tree*
2. *AVL-Tree (The Height Balance Tree)*
3. *Splay-Tree (The Self-Adjusting Tree)*

3.1 Binary Search Tree

3.1.1 Gambaran Permasalahan

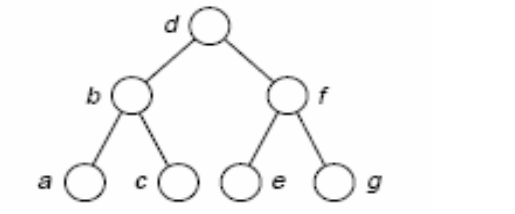
Bayangkan apabila kita ingin mencari sebuah data pada sebuah senarai berkait, tentunya tidak ada cara selain mencarinya secara sekuensial dari *pointer* elemen pertama senarai. Bandingkan jika kita melakukan pencarian di tabel kontigu dan dengan pencarian biner (*binary search*), tentunya pencarian akan lebih cepat. Dan sekarang bayangkan jika kita ingin melakukan operasi penambahan dan penghapusan elemen senarai. Operasi tersebut akan lebih lambat pada tabel kontigu daripada senarai berkait. Hal ini disebabkan karena operasi penambahan dan penghapusan pada tabel kontigu memerlukan pemindahan banyak entri data setiap saat, dibandingkan dengan senarai berkait yang hanya membutuhkan sedikit permainan *pointer*. [3]
Dari permasalahan diatas tentunya dapat kita simpulkan bahwa alangkah baiknya jika kita dapat melakukan operasi pencarian dengan kecepatan eksekusi tinggi(seperti pada tabel kontigu dan pencarian biner) dan operasi

penambahan dan penghapusan dengan kecepatan eksekusi tinggi (seperti pada senarai berkait).
Disinilah keunggulan *Binary Search Tree* akan sangat berguna, dimana *Binary Search Tree* dapat menjadi solusi permasalahan tersebut karena pencarian serta penambahan dan penghapusan elemennya memiliki kecepatan eksekusi yang tinggi. Operasi pencarian, penambahan, dan penghapusan pada *Binary Search Tree* memiliki *run-time* O (log n). [3]

3.1.2 Definisi Binary Search Tree

Binary Search Tree adalah sebuah pohon biner yang mempunyai properti tambahan. Properti ini adalah :

1. Semua elemen pada upapohon (subpohon) kiri nilainya lebih kecil atau sama dengan nilai akar.
2. Semua elemen pada upapohon kanan nilainya lebih besar dari nilai akar.
3. Upapohon kiri dan kanan merupakan *Binary Search Tree*. [6]



Gambar 6. Contoh Binary Search Tree

3.1.3 Operasi Pada Binary Search Tree

Terdapat 3 operasi yang sangat mendasar dan juga sangat penting pada *binary search tree* yaitu operasi pencarian, penambahan, dan penghapusan elemen. Ketiga operasi tersebut merupakan operasi yang sangat mendasar yang harus ada pada setiap struktur penyimpanan data.

3.1.3.1 Operasi Pencarian

Untuk melakukan operasi pencarian pada *Binary Search Tree*, pertama-tama kita bandingkan dahulu kunci data yang ingin kita cari dengan kunci dari data akar, jika tidak cocok maka cari pada upapohon kiri atau kanan sampai kunci data yang ingin dicari cocok.[3]
Berikut sketsa kasar algoritma operasi pencarian pada *Binary Search Tree* :
Algoritma Cari Simpul (kunci K, pohon P) [7]

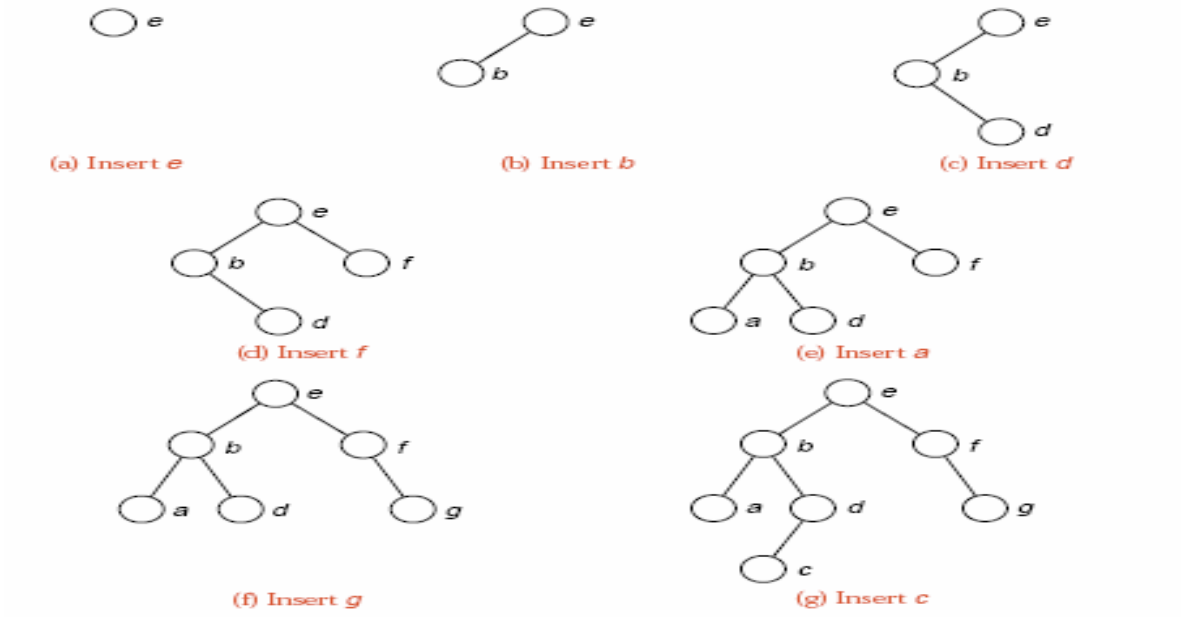
1. Jika pohon P kosong kembalikan nilai NULL (tak terdefinisi)
2. Jika kunci K cocok dengan kunci akar t maka kembalikan simpul P
3. Jika kunci K > kunci akar simpul P maka kembalikan nilai dari Cari Simpul (kunci K, upapohon kanan (P))

Jika kunci K <= kunci akar simpul P maka kembalikan nilai dari Cari Simpul (kunci K, upapohon kiri(P)

3.1.3.2 Operasi Penambahan Elemen

Dalam penambahan elemen pada *Binary Search Tree*, terdapat 2 kasus yang harus kita perhatikan yaitu penambahan ke pohon kosong dan penambahan ke pohon pencarian biner (*Binary Search Tree*) yang sudah terisi. Penambahan ke pohon kosong tentunya cukup mudah, sedangkan untuk penambahan ke *Binary Search Tree* yang sudah terisi maka kita harus membandingkannya dengan kunci dari akar pohon yang bersangkutan. Jika kuncinya lebih kecil maka tambahkan simpul baru ke upapohon kiri, jika lebih besar maka tambahkan ke upapohon kanan, dan jika kunci nya sama atau simpul dengan kunci yang ingin ditambahkan sudah ada, maka tidak akan melakukan apa-apa.[3]
Berikut sketsa kasar penambahan elemen pada *Binary Search Tree* :
Algoritma Insert (kunci K, pohon P) [7]

1. Jika pohon P kosong maka kembalikan pohon baru dengan kunci akar = K <basis 0>
2. Jika kunci akar P = K, kembalikan P (tanpa diubah)
3. Jika kunci K > kunci akar P,maka Insert (kunci K, upapohon kanan(P))
4. Jika kunci K < kunci akar P, maka Insert (kunci K, upapohon kiri(P))

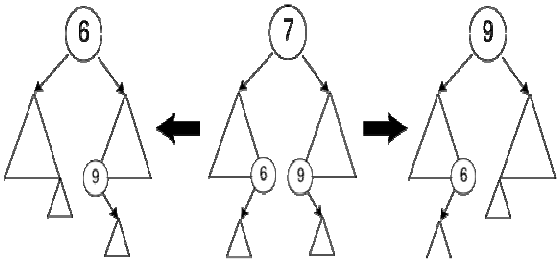


Gambar 7. Operasi Penambahan Elemen pada Binary Search Tree

3.1.3.3 Operasi Penghapusan Elemen

Penghapusan elemen pada *Binary Search Tree* merupakan operasi yang paling kompleks dari ketiga operasi ini. Dalam penghapusan elemen ini ada 3 kasus yang harus kita perhatikan, antara lain :

1. Menghapus daun : menghapus simpul yang tidak mempunyai upapohon.
2. Menghapus simpul yang mempunyai sebuah upapohon kiri/kanan : hapus dan gantikan dengan upapohonnya
3. Menghapus simpul yang mempunyai 2 upapohon : timpa simpul yang ingin dihapus dengan simpul yang mempunyai kunci terbesar pada upapohon kiri



Gambar 8. Operasi Penghapusan Elemen Pada Binary Search Tree

Berikut sketsa kasar penghapusan elemen pada *Binary Search Tree* :

Algoritma DelNode(kunci K, pohon P) [5]

1. Cari simpul yang mengandung kunci K
2. Jika simpul dengan kunci K ketemu maka :
   2.1. Jika upapohon kiri kosong maka tukar simpul P dengan upapohon kanan P, lalu hapus simpul yang sudah ditukar tersebut
   2.2. Jika upapohon kanan kosong maka tukar simpul P dengan upapohon kiri P, lalu hapus simpul yang sudah ditukar tersebut

3.2 AVL-Tree

Kebanyakan aplikasi saat ini melakukan operasi penambahan dan penghapusan elemen secara terus-menerus tanpa urutan yang jelas urutannya. Oleh karena itu sangatlah penting untuk mengoptimasi waktu pencarian dengan menjaga agar pohon tersebut mendekati seimbang sepanjang waktu. Dan hal ini telah diwujudkan oleh 2 orang matematikawan Russia , G.M. Adel'son-Vel'skii dan E.M. Landis. Oleh karena itu *Binary Search Tree* ini disebut AVL-tree yang diambil dari nama kedua matematikawan Russia tersebut.[3]

Tujuan utama dari pembuatan AVL-Tree ini adalah agar operasi pencarian, penambahan, dan penghapusan elemen dapat dilakukan dalam waktu O(log n) bahkan untuk kasus terburuk pun. Tidak seperti *Binary Search Tree* biasa yang dapat mencapai waktu O(1.44 log n) untuk kasus terburuk. [3]

3.2.1 Definisi AVL-Tree

Dalam pohon yang benar-benar seimbang, upapohon kiri dan kanan dari setiap simpul mempunyai tinggi yang sama. Walaupun kita tidak dapat mencapai tujuan ini secara sempurna, setidaknya dengan membangun *Binary Search Tree* dengan metode penambahan elemen yang nantinya akan kita bahas, kita dapat meyakinkan bahwa setiap upapohon kiri dan kanan tidak akan pernah berselisih lebih dari 1.

Jadi, sebuah *AVL-Tree* merupakan *Binary Search Tree* yang upapohon kiri dan kanan dari akarnya tidak akan berselisih lebih dari 1 dan setiap upapohon dari *AVL-Tree* juga merupakan *AVL-Tree*. Dan setiap simpul di *AVL-Tree* mempunyai faktor penyeimbang (*balance factor*) yang bernilai *left-higher* (upapohon kiri > kanan), *equal-height* (upapohon kiri = kanan), *right-higher* (upapohon kiri < kanan). [3,4]

3.2.2 Operasi Pada AVL-Tree

Operasi yang akan kita bahas ini adalah operasi yang sangat mendasar yaitu operasi pencarian, penambahan, dan penghapusan elemen.

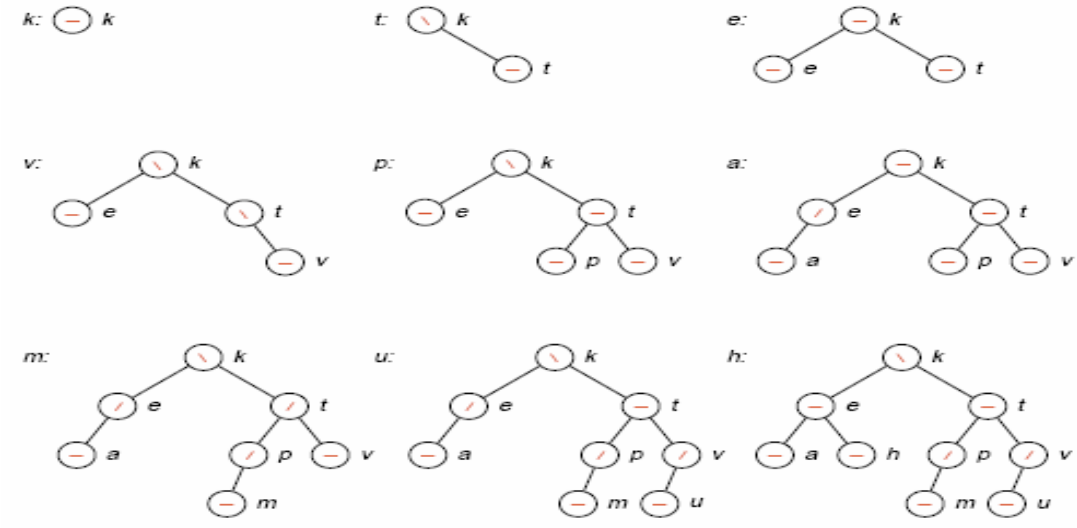
3.2.2.1 Operasi Pencarian Elemen

Operasi pencarian elemen pada *AVL-Tree* relatif sama dengan operasi pencarian elemen pada *binary search tree*. Hal ini tentunya karena *AVL-Tree* itu sendiri merupakan *binary search tree* yang sedikit dimodifikasi dengan menambah properti khusus yaitu keseimbangan pohon.

3.2.2.2 Operasi Penambahan Elemen

Karena *AVL-Tree* ini juga merupakan *binary search tree* maka penambahan elemen pada *AVL-Tree* ini juga dapat kita lakukan dengan metode yang sama dengan *binary search tree.* Pada kebanyakan kasus, penambahan elemen itu sering sekali tidak membuat tinggi dari upapohonnya meningkat sedemikian sehingga dapat membuat selisih tinggi dari upapohon kiri dan kanan lebih dari 1. Namun, bukan tidak mungkin hal ini bisa terjadi. Satu-satunya kasus dimana masalah ini muncul adalah ketika simpul baru itu ditambahkan ke upapohon yang lebih tinggi dari upapohon lainnya sehingga tinggi nya akan bertambah dan membuat selisih tinggi nya menjadi lebih dari 1. [3]

Berikut gambar contoh penambahan simpul sederhana, kita menggunakan '/' sebagai tanda bahwa faktor penyeimbang bernilai *left-higher*, '-' untuk *equal-height* dan '\' untuk *right-higher*. [3]



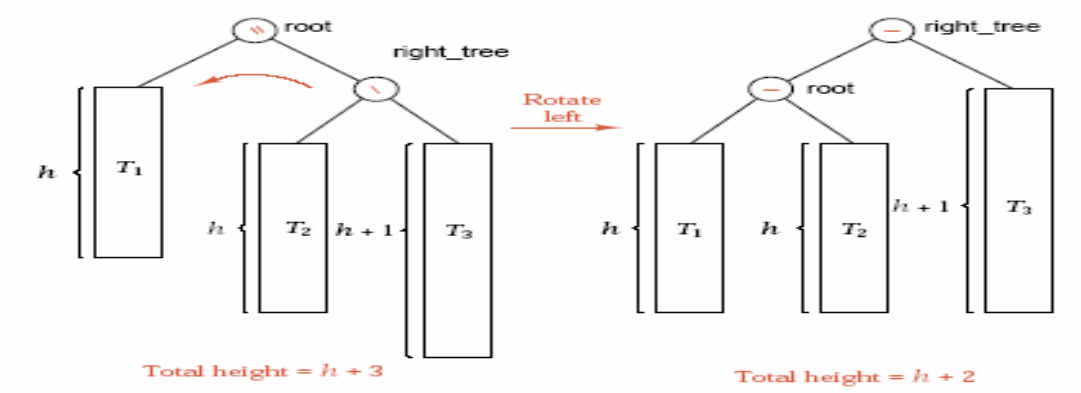
Gambar 9. Penambahan Elemen Pada AVL-Tree yang Sederhana

Sekarang mari kita tinjau masalah dimana penambahan elemen tersebut akan membuat *AVL-Tree* tidak lagi memenuhi syarat sebuah *AVL-Tree*. Solusi yang ditawarkan untuk menyelesaikan masalah ini adalah dengan merotasi upapohon tersebut sedemikian sehingga

selisih tinggi setiap upapohon nya tidak lebih dari 1. Lebih jelasnya lagi, mari kita asumsikan kita telah menambahkan simpul baru sehingga selisih tinggi setiap upapohonnya lebih dari 1. Ada 3 kasus yang muncul pada keadaan ini yakni:

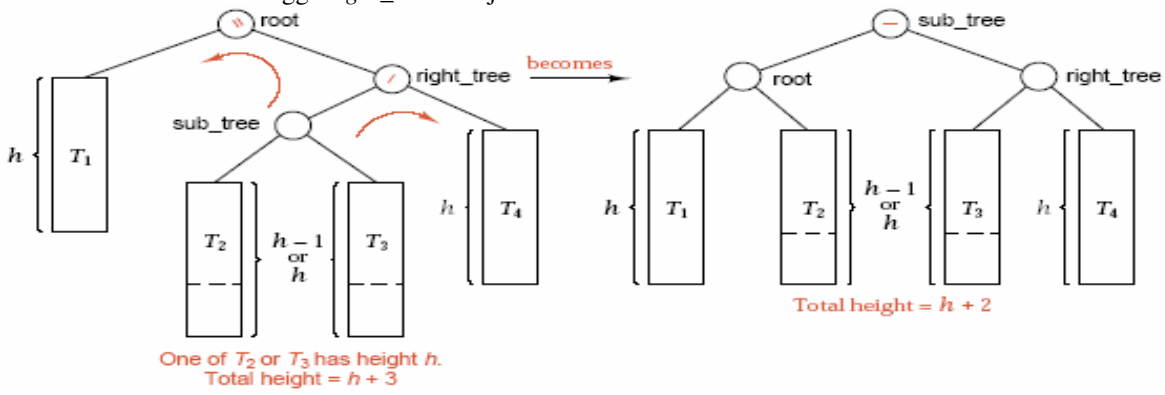
1. Upapohon kanan lebih tinggi

Yang harus kita lakukan adalah merotasi kiri / *left rotation* (seperti pada gambar.10) yaitu merotasi *right_tree* tersebut ke *root* dan menjadikan *root* tersebut menjadi upapohon kiri dari *right_tree*. Lalu T2 dijadikan upapohon kanan dari *root*.



Gambar 10a. Penyeimbangan AVL-Tree dengan Merotasi kiri (*Left Rotation*)

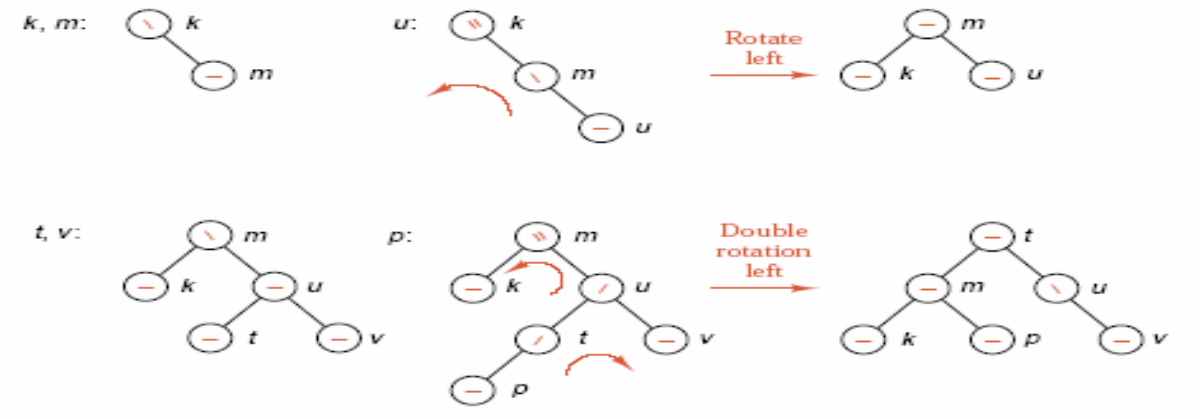
2. Upapohon kiri lebih tinggi
   Pada kasus kedua ini permasalahannya sedikit lebih rumit dimana kita harus memindahkan nya sejauh 2 tingkat ke simpul sub_tree (seperti pada gambar.11). Proses ini disebut rotasi ganda / *double rotation* karena transformasi dilakukan dalam 2 tahap. Pertama rotasi *sub_tree* ke *right_tree* sedemikian sehingga *right_tree* menjadi upapohon kanan dari *sub_tree* dan upapohon kanan *sub_tree* sebelumnya menjadi upapohon kiri *right_tree*. Lalu rotasi *sub_tree* ke *root* sedemikian sehingga *root* menjadi upapohon kiri dari *sub_tree* dan upapohon kiri *sub_tree* yang sebelumnya menjadi upapohon kanan dari *root*.



Gambar 10b. Penyeimbangan AVL-Tree dengan Merotasi ganda (*double Rotation*)

3. Upohon dengan tinggi yang sama
   Kasus terakhir adalah upapohon kiri dan kanan memiliki tinggi yang sama, namun pada kenyataannya kasus ini tidak akan pernah terjadi. [3]

Berikut contoh penambahan elemen *AVL-Tree* yang memerlukan rotasi :



Gambar 11. Penambahan AVL-Tree dengan Rotasi

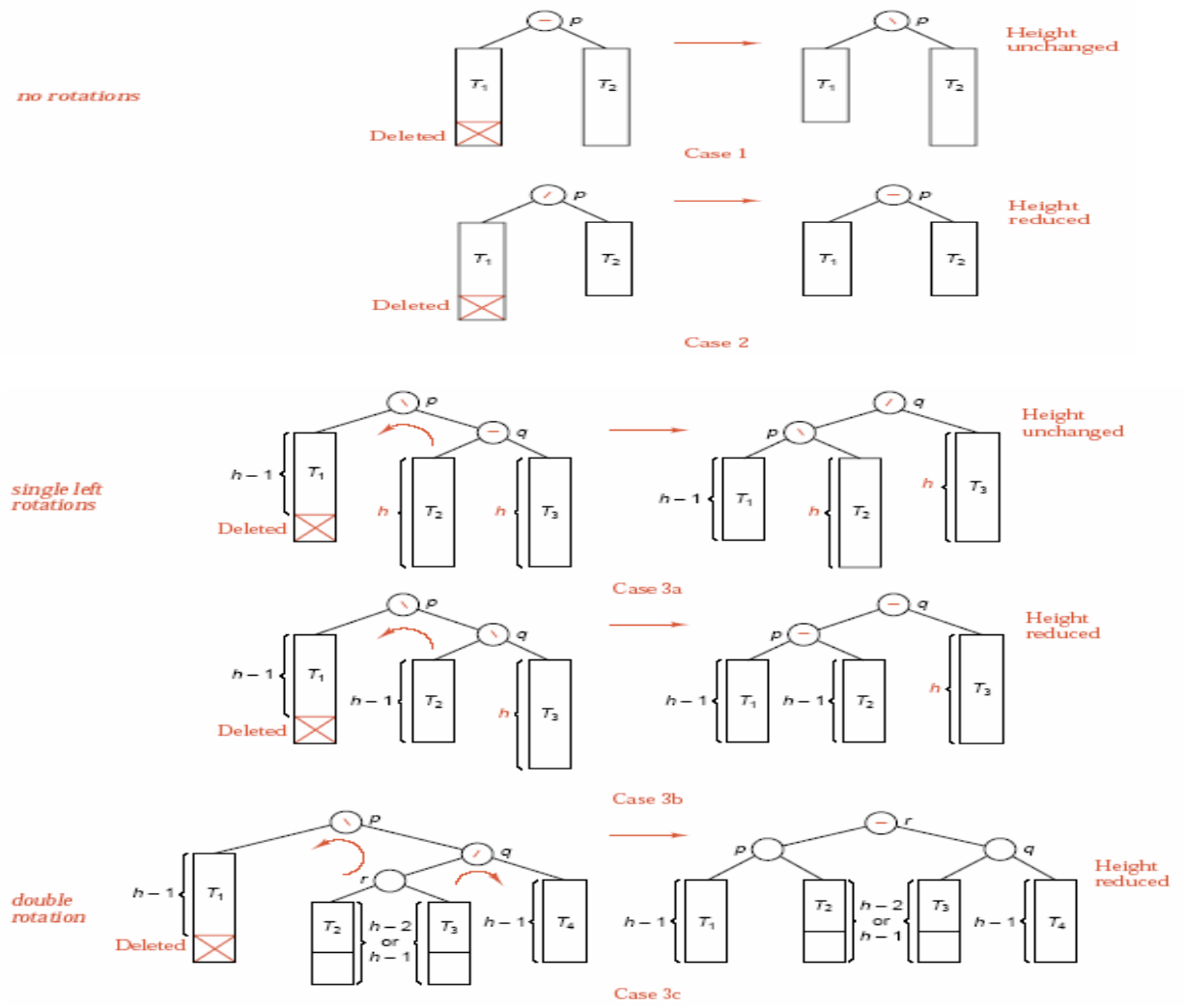
3.2.2.3 Operasi Penghapusan Elemen

Pada operasi penghapusan elemen *AVL-Tree* kita akan menggunakan ide dasar yang sama dengan penambahan elemen, seperti memakai rotasi tunggal dan ganda. [3]

Berikut metode penghapusan elemen *AVL-Tree* :

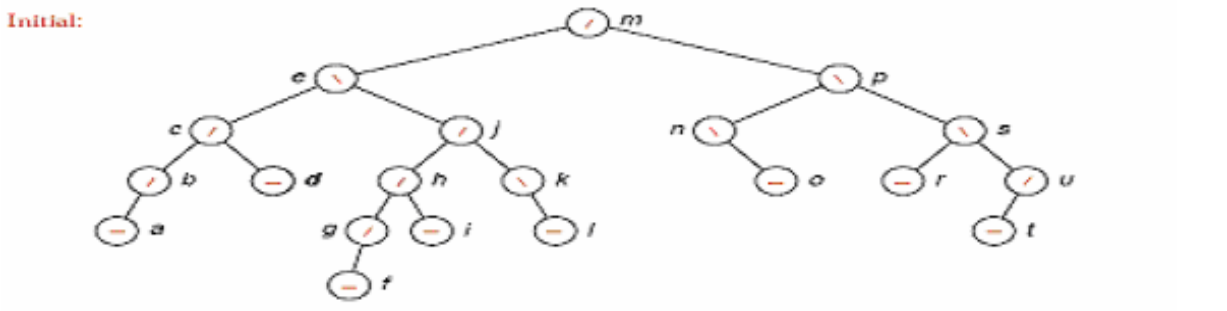
- Isolasi masalah sedemikian sehingga kita sampai pada kasus dimana simpul x yang ingin dihapus paling banyak mempunyai 1 anak. Jika x mempunyai 2 anak, cari simpul y yang merupakan *predecessor* x dan tidak mempunyai upapohon kanan. Letakkan y keposisi x
- Hapus simpul x, dari yang kita ketahui di tahap pertama tadi bahwa x maksimal memiliki 1 anak, sehingga kita tinggal menghapusnya dan memindahkan upapohonnya keposisi x. Namun kita harus memperhatikan efek yang ditimbulkan dari penghapusan x pada simpul-simpul sebelum x. Untuk itu kita pakai sebuah variable boolean *shorter* untuk melihat apakah tinggi dari upapohon menjadi lebih kecil.
- variabel *shorter* diinisialisasi *true*, berikut adalah kasus-kasus yang harus ditangani untuk setiap simpul p yang merupakan simpul yang terdapat diantara akar pohon sampai x. Dan kasus-kasus berikut hanya akan ditangani jika variabel *shorter* masih bernilai *true*, dan jika nilainya sudah false maka algoritma diterminasi.
- kasus 1 : simpul p mempunyai faktor penyeimbang *equal-height*, ubah faktor penyeimbangnya sesuai upapohon kiri atau kanan yang memendek, dan *shorter* menjadi false
- kasus 2 : faktor penyeimbang p bukan *equal-height,* dan upapohon yang diperpendek adalah upapohon yang lebih tinggi, maka ubah faktor penyeimbang menjadi *equal-height* , dan *shorter* tetap bernilai *true*.
- kasus 3 : faktor penyeimbang p bukan *equal-height,* dan upapohon yang diperpendek adalah upapohon yang lebih pendek. Maka syarat sebuah *AVL-Tree* tentunya telah dilanggar, dengan demikian kita harus melakukan rotasi untuk menyeimbangkan kembali *AVL-Tree*. Ambil q sebagai akar dari upapohon yang lebih tinggi dari p (yang tidak diperpendek). Dari sini kita mempunyai 3kasus berdasarkan faktor penyeimbang dari q, yakni :
  a. Faktor penyeimbang q *equal-height,* lakukan rotasi tunggal dan *shorter* menjadi *false*.
  b. Faktor penyeimbang q sama dengan faktor penyeimbang p, lakukan rotasi tunggal ubah faktor penyeimbang p dan q menjadi *equal-height* dan *shorter* tetap bernilai *true.*
  c. Faktor penyeimbang q berkebalikan dengan faktor penyeimbang p, lakukan rotasi ganda (pertama lakukan rotasi q dan sekitarnya lalu rotasi pada p dan sekitarnya), ubah nilai faktor penyeimbang menjadi *equal-height* dan faktor penyeimbang dari simpul lainnya sebagaimana mestinya, dan nilai *shorter* tetap *true*.

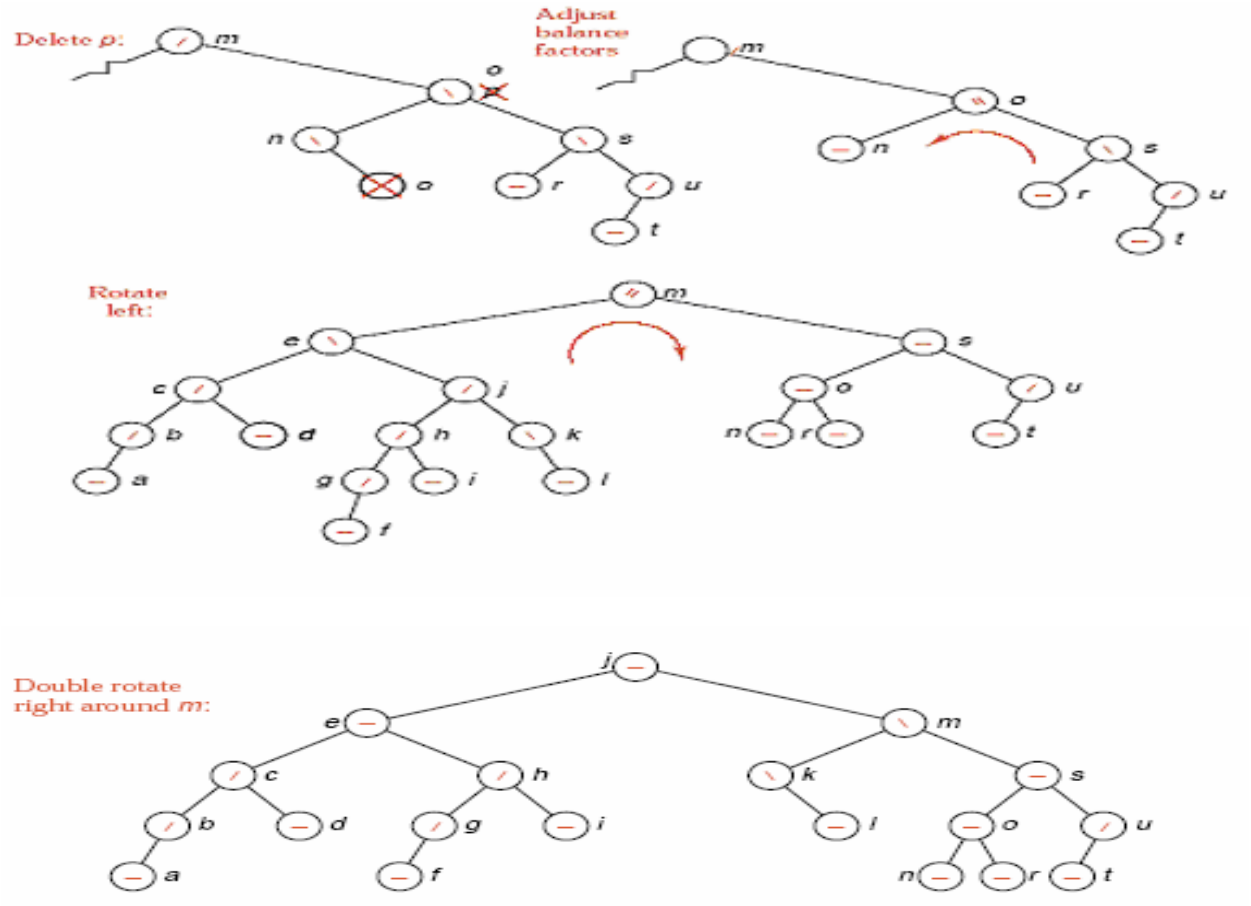
Berikut ilustrasi penghapusan elemen pada AVL-Tree :



Gambar 12a. Ilustrasi Penghapusan Elemen AVL-Tree

Untuk lebih memahami operasi penghapusan elemen *AVL-Tree*. Berikut contoh penghapusan *AVL-Tree* :





Gambar 12b. Contoh Penghapusan Elemen AVL-Tree

3.2 Splay-Tree

3.2.1 Definisi

Splay-Tree merupakan salah satu modifikasi *binary search tree* dengan tujuan tertentu. Tujuan utama dari konsep *splay-tree* ini adalah untuk memudahkan pencarian dan pengambilan data terutama data yang baru masuk dan yang paling aktif diakses atau dimodifikasi.

Perbedaan utama *splay-tree* dibandingkan *binary search tree* ataupun *AVL-Tree* adalah data baru atau yang frekuensi aksesnya tinggi berada dekat dengan akar pohon sehingga untuk mengakses data tersebut tidak diperlukan waktu yang lama dibandingkan dengan binary search tree atau AVL-Tree yang membuat kita harus menelusuri pohon sampai ke daun karena data baru akan ditempatkan sebagai daun. Untuk jenis data seperti inilah *splay-tree* sangat dibutuhkan dimana setiap kali adanya operasi penambahan atau juga pengambilan data, struktur pohonnya akan dirombak ulang dengan menaikkan data tersebut (jika berada dibagian bawah pohon / daun) sesuai dengan tingkat frekuensi akses dan keterbaharuannya. [3,4]

3.2.2 Penggunaan / Aplikasi

Contoh nyata dari struktur data yang menggunakan konsep *splay-tree* ini adalah struktur data sebuah rumah sakit.

Tentunya kita tahu bahwa sebuah rumah sakit itu harus menjaga *record* data pasien-pasiennya. *Record* data pasien-pasien yang masih berada dirumah sakit tentunya mempunyai keaktifan yang tinggi atau dengan kata lain frekuensi akses nya tinggi, sedangkan untuk pasien yang sudah keluar rumah sakit frekuensi aksesnya akan rendah. Dan sebuah rumah sakit tidak boleh menghapus data pasien yang sudah keluar dari rumah sakit tersebut karena tentunya data tersebut bisa saja akan dibutuhkan disuatu saat

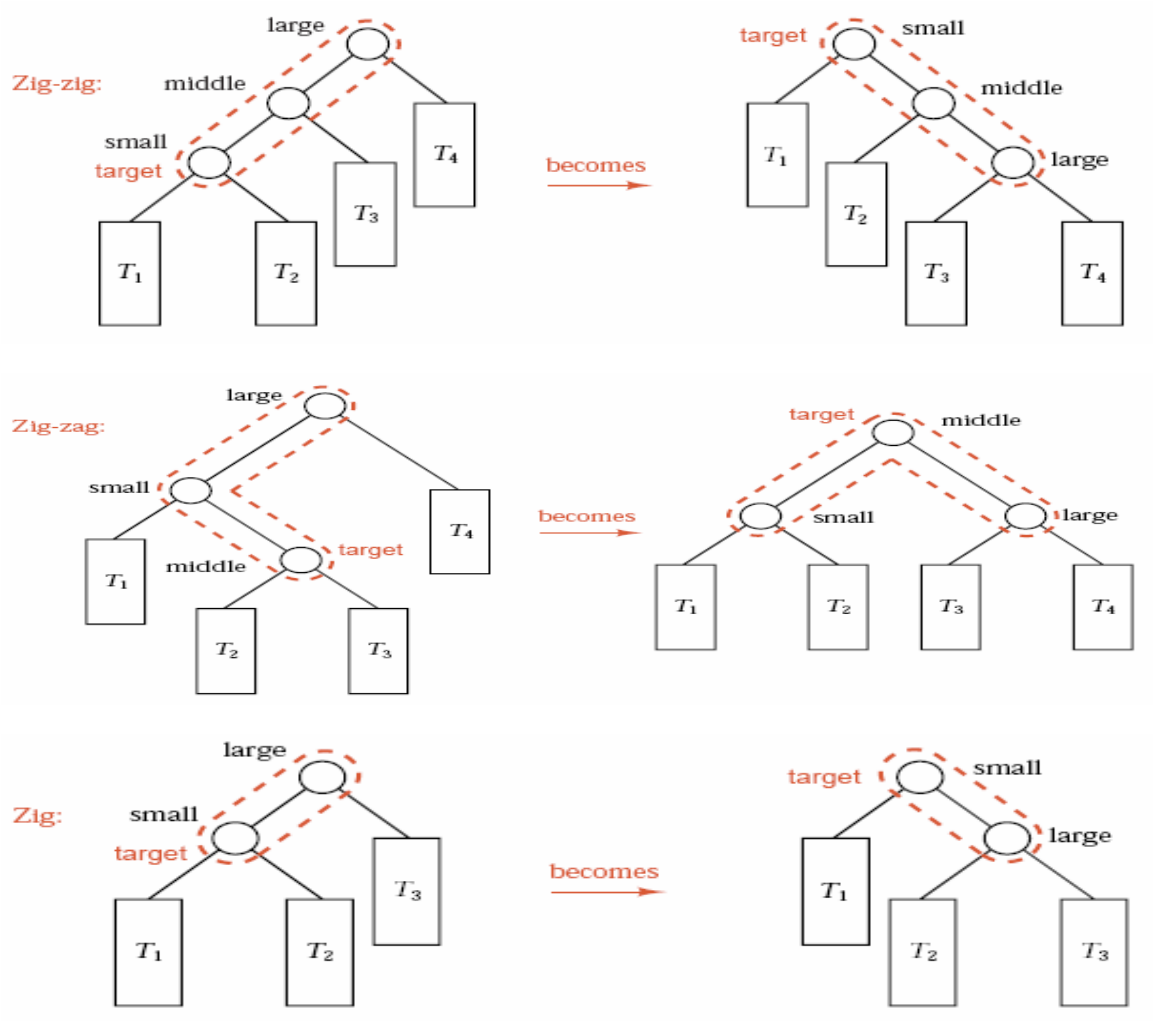
nanti oleh pihak-pihak lain misalnya. Dan untuk itulah akan sangat tidak efisien jika *record* data pasien baru atau yang frekuensi aksesnya tinggi berada pada tingkat daun pada struktur data pohon tersebut sehingga untuk setiap kali mengaksesnya diperlukan waktu yang lebih lama dibanding dengan *record* data yang tergolong sudah tidak aktif lagi. [3]

3.2.3 Pembuatan Splay-Tree

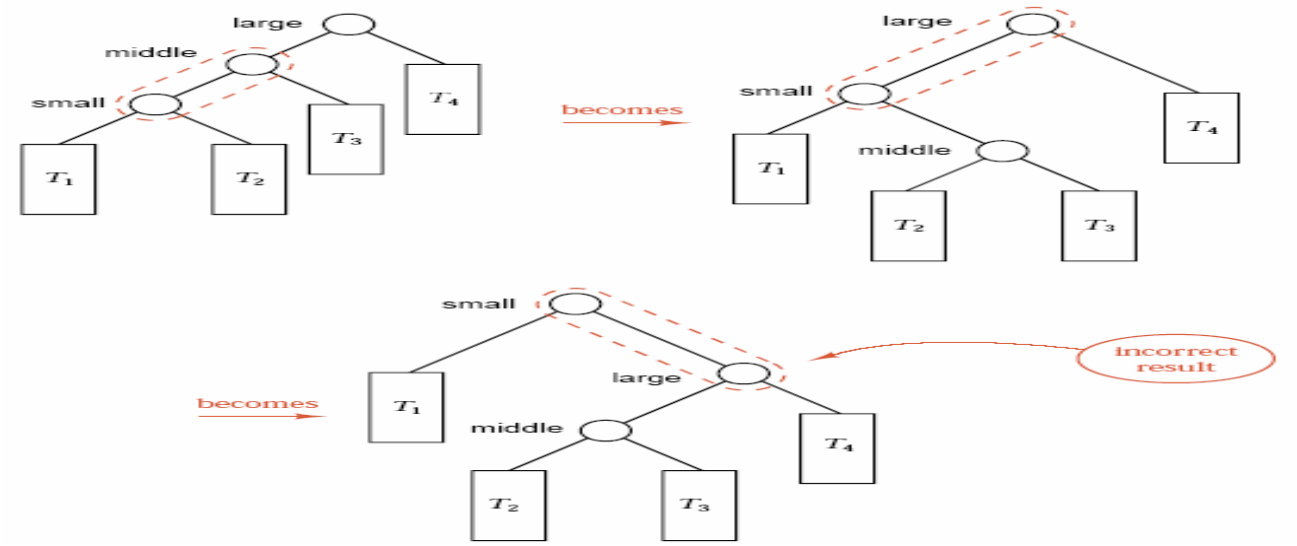
Seperti yang telah dijelaskan tadi, *splay tree* adalah sebuah *binary search tree* yang sedikit dimodifikasi. Oleh karena itu untuk membuat *splay tree* kita hanya perlu melakukan proses *splaying* pada *binary search tree*.

Ide utama dalam melakukan *splaying* adalah memindahkan simpul tujuan 2 level keatas dalam setiap langkahnya. [3]

Sekarang bayangkan lintasan dari akar sampai ke simpul yang telah diakses. Setiap kali kita bergerak kekiri disebut *zig* dan setiap kekanan disebut *zag*. Bergerak 2 kali kekiri disebut *zig-zig*, 2 kali kekanan disebut *zag-zag*, kekiri lalu kekanan disebut *zig-zag*, dan kekanan lalu kekiri disebut *zag-zig*. Inilah 4 kemungkinan yang dapat terjadi saat bergerak 2 level kebawah. Bagaimanapun jika panjang lintasannya genap, diperlukan satu langkah lagi baik kekiri ataupun kekanan. Berikut contoh *zig-zig*, *zig-zag*, dan *zag*. Sisanya yaitu *zag-zag*, *zag-zig*, dan *zig* merupakan refleksinya saja. [3]



Gambar 13a. Rotasi Pada Splay Tree

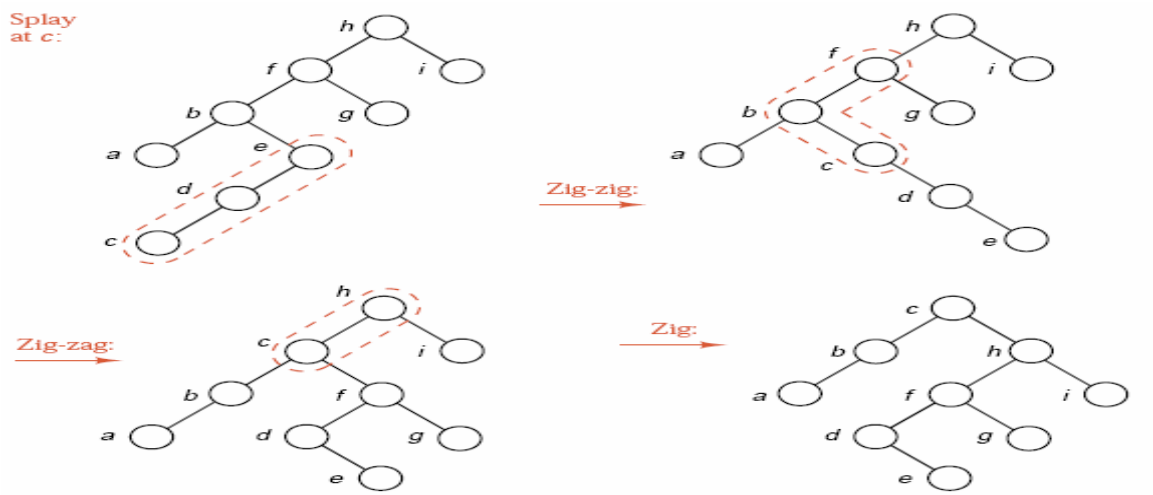


Gambar 13b. Kesalahan Dalam Proses zig-zig Dengan Rotasi Tunggal

Gambar diatas menunjukkan proses zig-zig yang salah karena menggunakan rotasi tunggal. Oleh karena itu kita harus selalu berpikir untuk mengangkat simpul yang dituju 2 level dalam setiap langkah (kecuali apabila yang tersisa diakhir hanya langkah *zig* atau *zag* saja). Juga harus diperhatikan bahwa hanya simpul yang berada pada lintasan antara akar sampai ke simpul tujuanlah yang posisinya diubah seperti pada gambar 13a yang dilingkari dengan garis merah putus-putus sedangkan upapohonnya tidak diubah hanya dipindahkan untuk mempertahankan pohon tersebut agar tetap merupakan *binary search tree.* [3]

3.2.4 Contoh

Berikut contoh *splaying* pada *binary search tree.* Kita mulai proses splaying pada simpul c. Lintasan untuk mencapai simpul c dari akar pohon adalah h,f,b,e,d,c. Dari e ke d ke c kita lakukan rotasi *zig-zig* (pada bagian yang dilinkari garis merah putus-putus pada pohon pertama).
Langkah berikutnya adalah lintasan dari f ke b ke c dengan menggunakan rotasi *zig-zag*, disini upapohon d dan e (diluar lintasan) tidak berubah bentuk tapi hanya dipindahkan ke posisi yang baru. Pada tahap ini simpul c tinggal 1 tingkat lagi ke akar, dan disini upapohon f juga hanya dipindahkan posisinya.[3]



Gambar 13c. Contoh Proses Splaying pada Binary Search Tree

4. Kesimpulan

Berdasarkan hal-hal diatas maka dapat diambil kesimpulan bahwa penerapan teori pohon sangatlah berguna dalam kajian struktur data dimana dengan teori pohon itu kita bisa mendapatkan struktur penyimpanan data alternatif yang relatif lebih baik dan efisien daripada struktur penyimpanan data lainnya, seperti representasi senarai berkait dan tabel kontigu misalnya.

Terbukti bahwa representasi data dengan *binary search tree* jauh lebih baik daripada representasi data dengan tabel kontigu ataupun senarai berkait.

Dan juga dapat kita tarik kesimpulan bahwa *binary search tree* sangat fleksibel dimana konsepnya ini masih dapat dikembangkan untuk menyelesaikan suatu permasalahan baru yang sekarang ini sudah umum ditemui.

Modifikasi konsep *binary search tree* yang cukup terkenal yaitu *AVL-Tree* dan *Splay Tree* yang sangat efisien dalam menyelesaikan kasus-kasus tertentu.

DAFTAR PUSTAKA

[1] Munir, Rinaldi (2005) Diktat Kuliah IF2153 Matematika Diskrit. Program Studi Teknik Informatika, Institute Teknologi Bandung.

[2] Liem, Inggriani (2005) Diktat Kuliah IF2182 Algoritma dan Struktur Data Program Studi Teknik Informatika Institute Teknologi Bandung.

[3] Data Structure and Program Design in C++, Robert L. Kruse and Alexander J. Ryba

[4] NIST. *(2004). National Institute of Standards and Technology.* **http://www.nist.gov/*dads/HTML/binarySearchTree.html*** **http://www.nist.gov/*dads/HTML/avltree.html*** **http://www.nist.gov/*dads/HTML/splaytree.html*** Tanggal akses: 20 Desember 2006 pukul 14:00

[5] Wikipedia *http://en.wikipedia.org/wiki/Binary_search_tree.html* Tanggal akses : 20 Desember 2006 pukul 14.00

[6] http://www.csd.abdn.ac.uk/~spt/teaching/CS2005/lectures/9.shtml Tanggal akses : 20 Desember 2006 pukul 14.00

[7] Aplikasi *Binary Search tree* pada *star catalog correlation* *http://www.adass.org/adass/proceedings/adass97/nguyend.html* Tanggal akses : 20 Desember 2006 pukul 14.00