Dasar Kerja Pengaplikasian *Self-balancing Binary Search Tree* untuk Pencarian sebagai Struktur Data yang Lebih Mangkus dan Sangkir

Michell Setyawati Handaka (135 08 045)

Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
Jl. Ganeca no. 10, Bandung
e-mail: if18045@students.if.itb.ac.id

ABSTRAK

Proses pencarian adalah salah satu operasi paling krusial dalam pengolahan data, khususnya mengingat perkembangan ruang memori yang makin lama, tak ayal lagi, akan menjadi –dapat dikatakan– tidak terbatas. Dengan proses pencarian secara traversal, kompleksitas waktu dalam notasi asimptotik setara dengan O(n) yang dalam hal ini tergolong algoritma polinomial yang mangkus dalam spektrum waktu. Akan tetapi, dengan pencarian biner, kompleksitas dapat ditekan sampai kepada tingkat O(log *n*) saja, yang tentunya jauh lebih baik.

Pencarian secara rekursif dapat dilakukan dengan memanfaatkan tipe struktur data yang juga rekursif, seperti pohon –yang dalam hal ini adalah pohon biner–. Ide pencarian biner hanya dapat dilakukan terhadap suatu data yang sudah terurut dengan baik, dalam struktur data pohon, tipe yang demikian ini dikenal dengan nama pohon biner terurut.

Persoalan berikutnya adalah, pencarian pada pohon biner terurut sangat bergantung kepada tinggi pohon, sehingga akan sangat menguntungkan jika kedalaman suatu pohon dapat dibuat minimum. Hal ini dapat dicapai dengan pendekatan pohon biner seimbang, dimana kedalaman pasti minimum dan banyak simpul per aras berjumlah maksimum.

Untuk membangun suatu pohon seimbang sebenarnya sangat mudah. Akan tetapi permasalahan yang dihadapi adalah ketika kita ingin pohon seimbang ini terurut dan proses penyeimbangan haruslah tidak mengganggu keterurutan data.

Untuk menangani kasus ini, dapat digunakan suatu struktrur data baru yang dikenal sebagai *self-balancing binary search tree*. Pada struktur data ini, kompleksitas waktu pencarian akan selalu bernilai O(log *n*). Hal yang demikian ini dapat tercipta karena pohon merepresentasikan pohon terurutnya dalam porsi yang seimbang. Untuk menjaga struktur data yang demikian ini, diperlukan suatu operasi tambahan, yakni pemeliharaan struktur, setiap kali penyisipan atau penghapusan dilakukan. Menariknya, operasi ini tidaklah mengubah kompleksitas waktu baik penyisipan maupun penghapusan tanpa pemeliharaan struktur : yakni tetap O(log *n*).

Kata kunci: pohon biner terurut, pohon biner seimbang, pencarian beruntun, pencarian biner, *self-balancing binary seacrh tree* : pohon *Red-black*, pohon *AVL*, pohon *AA*, pohon *Splay*, pohon *Scapegoat*, *Treap*.

1. PENDAHULUAN

Dewasa ini, kapasitas divais penyimpanan data / memori terus menerus bertumbuh membesar dengan laju yang mana kian lama kiat pesat. Sebagai acuan, dalam waktu satu tahun, kapasitas HDD –*Hard Disk Drive*– meningkat 50% dari yang semula hingga kini (tahun 2009) hanya mencapai 2TB akan segera menjadi 3TB pada tahun 2010[3].

Sementara itu, kapasitas memori selalu diidentikkan pertumbuhannya berlawanan dengan kecepatan pengolahan. Dapat diibaratkan, memori beranalogi dengan sebuah tas : semakin besar ukuran tas, semakin besar pula kapasitas tas tersebut untuk dapat menampung barang-barang yang akan dimasukkan ke dalamnya; akan tetapi, hal ini juga diimbangi dengan bertambah beratnya beban tas tersebut. Dengan demikian, salah satu hal yang menjadi *concern* masyarakat umum saat ini adalah mencari devais memori yang memiliki kapasitas besar namun tidak memberatkan, atau setidaknya memiliki efektivitas kinerja tinggi.

Salah satu operasi mendasar yang sering dilakukan dalam pengolahan maupun penggunaan data adalah *searching* atau yang lebih dikenal sebagai proses pencarian. Proses pencarian data mendapat porsi prioritas yang semakin tinggi seiring dengan meningkatnya ruang memori, mengingat hal ini menjadi krusial jika pengguna sedikit “jorok”. Bahkan, ketika pengguna berlaku apik sekalipun, bukan tidak mungkin tempat suatu data tertentu disimpan menjadi terlupakan akibat proses fragmentasi

memori yang terlampau banyak diakibatkan karena besarnya ukuran memori.

Adapun rata-rata jumlah waktu yang dibutuhkan untuk suatu proses pencarian menggunakan traversal dalam kompleksitas asimtotik adalah O(n); sementara itu, proses pencarian menggunakan *binary search tree* atau pohon biner terurut hanya memakan sejumlah O(log n) saja, yang tentunya lebih baik.

2. STRUKTUR DATA (TERLAMPIR)

3. DASAR TEORI PROSES PENCARIAN

Pada umumnya, algoritma untuk proses pencarian beruntun (*sequential search*) pada suatu list adalah sebagai berikut :

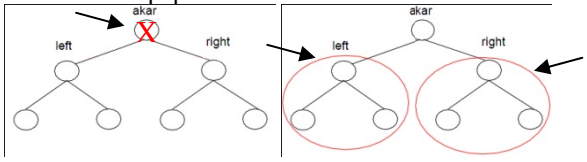
```
procedure SKEMAListSearch1 (input L : List, input X : InfoType,
                            output P : address, input Found : boolean)
{ I.S. List linier L sudah terdefinisi dan siap dikonsultasi, X terdefinisi }
{ F.S. P : address pada pencarian beurutan, dimana X diketemukan, P = Nil jika
tidak ketemu }
{ Found berharga true jika harga X yang dicari ketemu, false jika tidak ketemu }
{ Sequential Search harga X pada sebuah list linier L  }
{ Elemen diperiksa dengan instruksi yang sama, versi dengan boolean }
KAMUS LOKAL
ALGORITMA
   P ← First(L)
   Found ← false
   while (P ≠ Nil) and (not Found) do
      if (X = Info(P)) then
         Found ← true
      else
         P ← Next(P)
   { P = Nil or Found}
   { Jika Found maka P = address dari harga yg dicari diketemukan }
   { Jika not Found maka P = Nil }
```

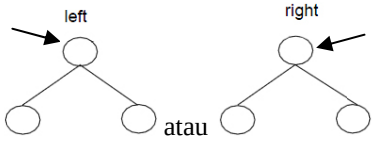
dengan $T(n) = \frac{n+1}{2}$ atau O(n).

Adapun skema pencarian untuk struktur data berupa pohon biner dapat digambarkan sebagai berikut :

```
function Search (P : BinTree, X : infotype) → boolean
{ Mengirimkan true jika ada node dari P yang bernilai X }
KAMUS LOKAL

ALGORITMA
   if (IsTreeEmpty(P)) then { Basis-0 }
      → false
   else { Rekurens }
      if (Akar(P) = X) then
         → true
      else
         → Search(Left(P),X) or Search(Right(P),X)
```

- Periksa akar pohon
- Jika ketemu −> selesai
- Jika tidak ketemu −> cari di upapohon kiri atau upapohon kanan





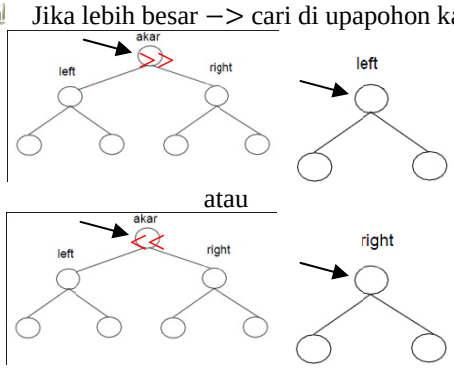
dengan proses rekursif yang demikian pun, kompleksitas waktunya adalah $T(n) = \frac{n+1}{2}$ atau O(n).

Jadi, keuntungan penggunaan struktur data pohon untuk proses pencarian beruntun hanyalah algoritma yang lebih ringkas karena penanganan kasus dilakukan secara rekursif.

Lain halnya, jika data yang kita miliki sudah kita urutkan sebelumnya. Ide inilah yang mendasari proses pencarian biner (*binary seacrh*).

Pada proses pencarian menggunakan pohon biner terurut, algoritmanya menjadi sebagai berikut :

- Periksa akar pohon
- Jika sama −> selesai
- Jika tidak sama −> bandingkan
- Jika X yang dicari lebih kecil daripada Akar(P) −> cari di upapohon kiri
- Jika lebih besar −> cari di upapohon kanan



dengan demikian, besar waktu yang diperlukan secara otomatis pasti berkurang mengingat sekarang pencarian berikutnya, jika diperlukan, hanya dilakukan pada upapohon kiri saja atau upapohon kanan saja.

Adapun bentuk suatu pohon biner terurut sangatlah bergantung kepada urutan masukan dan semakin tidak terurut masukan yang diberikan hasilnya akan cenderung semakin baik, sebagai contoh :

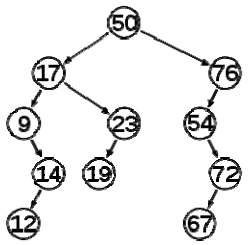
Data :

9	12	14	17	19	23	50	54	67	72	76

Urutan masukan :

50	17	9	23	14	19	12	76	54	72	67

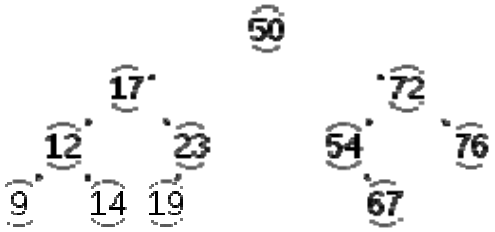
Hasil :



Urutan masukan :

50	17	12	23	9	14	19	72	54	76	67

Hasil :



Demikian pula jika data yang dimasukkan sudah terurut (membesar / mengecil), maka pohon yang terbentuk akan tentu berbeda pula, yakni terbentuk pohon condong (kanan / kiri).

Sementara itu, dengan algoritma pencarian pada pohon biner terurut, bentuk pohon yang menjadi input sangatlah berpengaruh terhadap kebutuhan waktu daripada suatu proses pencarian. Seandainya bentuk masukan adalah pohon biner terurut condong, maka kompleksitas waktunya adalah O(n). Akan tetapi, jika bentuk pohon yang diberikan adalah yang menyerupai pohon pada contoh terakhir, kompleksitas waktu menjadi hanya O(log n), yang tentunya tak dapat disangkal lagi akan lebih baik –lebih mangkus (efektif) dan lebih sangkir (efisien)–. Mengapa hal ini dapat terjadi?

Jika kita perhatikan, pada kasus pohon terakhir, pencarian selanjutnya hanya dikenakan pada setengah jumlah dari pencarian sebelumnya. Kasus yang demikian jelaslah menunjukkan suatu fungsi logaritmik dengan basis 2. Proses pencarian yang demikian ini dinamakan sebagai proses pencarian biner.

Pohon terurut yang dapat menekan kebutuhan waktu pencarian adalah pohon dengan jumlah kedalaman minimum dengan cara memaksimumkan jumlah simpul per arasnya. Pohon yang demikian ini adalah pohon biner seimbang.

Dengan demikian, masalah selanjutnya adalah bagaimana membuat suatu pohon biner yang seimbang. Adapun algoritma untuk pembuatan pohon biner seimbang secara umum adalah sebagai berikut ini :

```
function BuildBalancedTree (n : integer) → BinTree
{ Menghasilkan sebuah balanced tree }
{ Basis: n = 0: Pohon kosong  }
{ Rekurens: n>0: partisi banyaknya node anak kiri dan kanan,
  lakukan proses yang sama  }
KAMUS LOKAL
    P : address; L, R : BinTree; X : infotype
    nL, nR : integer
ALGORITMA
    if (n = 0) then { Basis-0 }
       → Nil
    else {Rekurens }
       { bentuk akar }
       input(X) { mengisi nilai akar }
       P ← Alokasi(X)
       if (P ≠ Nil) then
          { Partisi sisa node sebagai anak kiri dan anak kanan }
          nL ← n div 2; nR ← n - nL - 1
          L ← BuildBalancedTree(nL); R ← BuildBalancedTree(nR)
          Left(P) ← L; Right(P) ← R
       → P
```

Yang menjadi pertanyaan berikutnya adalah, bagaimana menggunakan algoritma untuk membentuk pohon biner seimbang tanpa merusak keterurutan pohon biner terurut. Jawaban dari persoalan ini adalah pohon biner terurut – seimbang (*self-balancing binary search tree*).

4. SELF-BALANCING BINARY SEARCH TREE[11]

Sebagian besar operasi pada pohon biner terurut membutuhkan waktu yang sebanding dengan kedalaman daripada pohon tersebut, sehingga akan jauh lebih menguntungkan jika kedalaman pohon dapat dijaga untuk tetap minimal. *Self-balancing binary search tree* / *height-balanced binary search tree* adalah pohon biner terurut yang secara otomatis mempertahankan kedalamannya untuk tetap minimum dengan cara melakukan operasi penyisipan dan / atau penghapusan.

Mengingat pohon biner dengan kedalaman h dapat memuat sebanyaknya $2^0 + 2^1 + \cdots + 2^k = 2^{k+1} - 1$ simpul, maka berlaku bahwa kedalaman minimum suatu pohon biner dengan jumlah simpul sebanyak n adalah $\log_2(n)$ dengan pembulatan ke bawah, atau dapat ditulis $\lfloor \log_2(n) \rfloor$.

Menjaga kedalaman suatu pohon biner untuk tetap pada nilai minimumnya –$\lfloor \log_2(n) \rfloor$– ternyata tidaklah dapat dilakukan mengingat algoritma pengolahannya melakukan eksploitasi yang berlebihan. Dengan demikian, kebanyakan algoritma *self-balanced* BST hanya menjaga agar kedalaman dari suatu pohon biner selalu dibatasi di atas oleh suatu konstanta yang merupakan suatu faktor dari batas minimum $\log_2(n)$.

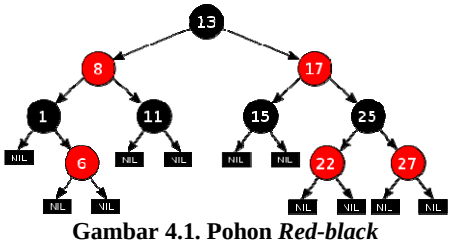
Adapun implementasi sesungguhnya dari *self-balancing binary search tree* adalah menggunakan struktur data seperti yang tertera di bawah ini, di antaranya :

<source-code / ilustrasi untuk setiap uraian di bawah ini terlampir>

RED-BLACK TREE / *SYMMETRIC BINARY B-TREES*[12]

Pohon *Red-black* memiliki struktur yang kompleks, tetapi hal ini diimbangi dengan kompleksitas waktu operasinya yang baik selain daripada kemangkusannya dalam dunia praktis : operasi pencarian, penyisipan, dan penghapusan hanya memiliki kompleksitas waktu asimtotik setara O(log *n*) saja.

Dalam penggunaanya, daun pada pohon *Red-black* tidaklah digunakan untuk menyimpan data sehingga tidak memakan ruang memori pada praktisnya; akan tetapi, jika daun-daun ini tetap disimpan secara eksplisit, beberapa algoritma pengolahannya menjadi lebih sederhana. Dalam prakteknya, pointer kepada daun dapat hanya satu buah elemen *dummy* / sentinel saja bagi satu pohon *Red-black* dengan semua elemen daun mengacu pada alamat tersebut.



Gambar 4.1. Pohon *Red-black*

Setiap simpul pada pohon *Red-black* memiliki atribut warna, yang mana hanya mungkin merah saja atau hitam saja. Adapun karakteristik lainnya –selain daripada karakteristik standar pada pohon biner terurut– dari pohon *Red-black* adalah sebagai berikut ini :

1. Suatu simpul hanya dapat merah saja atau hitam saja.
2. Atribut warna akar adalah hitam. (atribut warna suatu simpul yang menjadi akar hanya dapat berubah dari merah menjadi hitam).
3. Seluruh daun berwarna hitam.
4. Setiap anak dari seluruh simpul merah berwarna hitam.
5. Setiap lintasan sederhana dari suatu simpul yang diberikan kepada sebarang daun keturunannya berukuran sejumlah simpul hitam.

Batasan karakteristik di atas berimplikasi kepada sifat kritis dari sebuah pohon *Red-black* : lintasan terpanjang dari akar ke sebarang daun tidak mencapai dua kali ukuran lintasan terpendek dari akar ke daun lainnya. Dengan demikian, secara umum, pohon dapat dikatakan seimbang.

Karakteristik ini dapat terlaksana karena :

- tidak ada lintasan yang mengandung dua simpul merah secara berturutan.
- lintasan terpendek yang mungkin hanya mengandung simpul hitam, dan alternatif lintasan terpanjang yang mungkin mengandung simpul merah dan hitam berselingan.
- setiap lintasan terpanjang berukuran sejumlah seluruh simpul hitam sehingga tidak mungkin ada lintasan yang mencapai lebih dari dua kali panjang lintasan lainnya.

Dampak lain daripada karakteristik di atas adalah setiap simpul internal pada pohon *Red-black* pastilah memiliki dua anak, yang mana salah satu atau keduanya mungkin kosong.

Operasi pembacaan pada pohon *Red-black* menyerupai operasi standar pada BST. Akan tetapi, pada operasi penyisipan dan penghapusan, dibutuhkan suatu mekanisme perbaikan untuk menjaga agar karakteristik pohon *Red-black* tidak dilanggar. Operasi perbaikan ini memiliki kompleksitas waktu yang sangat baik, yakni sebesar O(log *n*) atau O(1) saja dan merupakan operasi yang dibutuhkan untuk mengubah warna simpul dan tidak lebih dari tiga operasi *tree rotation*. Dengan demikian, operasi penyisipan dan penghapusan tetap memiliki kompleksitan waktu O(log *n*).

- Penyisipan

  Penyisipan dimulai dengan menambahkan sebuah simpul, seperti pada penyisipan dalam BST, yang kemudian diwarnai dengan merah. Penyisipan selalu menggantikan sebuah daun hitam dengan simpul dalam yang berwarna merah dengan dua buah anak berupa daun hitam.

  Langkah selanjutnya ditentukan oleh warna simpul lain di sekitar simpul baru tersebut. Adapun kondisi pada saat penyisipan baru dilakukan adalah sebagai berikut ini :

  - Sifat (3) selalu tetap terjaga.
  - Sifat (4) terancam dilanggar hanya jika terjadi penambahan simpul merah, mewarnai ulang suatu simpul hitam menjadi merah, atau terjadinya rotasi.
  - Sifat (5) terancam dilanggar hanya jika terjadi penambahan simpul hitam, mewarnai ulang suatu simpul merah menjadi hitam, atau terjadinya rotasi.

  Catatan : N −> simpul yang disisipkan; P −> ayah N; G −> ayah P; U −> saudara P. Dalam bahasa C, simpul U dan G ditemukan dengan cara :

```
struct node *
grandparent(struct node *n)
{
   if ((n != NULL) && (n->parent != NULL))
      return n->parent->parent;
   else
      return NULL;
}
struct node *
uncle(struct node *n)
{
   struct node *g = grandparent(n);
   if (g == NULL)
      return NULL; // No grandparent means
no uncle
   if (n->parent == g->left)
      return g->right;
   else
      return g->left;
}
```

  KASUS 1 :

  SIMPUL **N** MENJADI AKAR POHON *RED-BLACK*, dengan demikian pewarnaan ulang menjadi hitam dilakukan untuk memenuhi sifat (2). Semenjak penambahan ini menambahkan satu simpul hitam ke seluruh lintasan sekaligus, sifat (5) tetap terjaga.

  KASUS 2 :

  SIMPUL **P** BERWARNA HITAM, dengan demikian sifat (4) tetap valid. Karena simpul **N** berwarna merah, sifat (5) tetap berlaku.

  KASUS 3 :

  Ketika baik simpul **P** MAUPUN SIMPUL **U** BERWARNA MERAH, maka kedua simpul ini dapat diwarnai ulang menjadi hitam sementara simpul **G** menjadi merah untuk menjaga sifat (5). Akan tetapi, hal ini (simpul **G** diwarnai merah) dapat berimbas kepada pelanggaran sifat (2) atau (4) −> jika ayah **G** berwarna merah. Untuk menyelesaikan masalah ini, dapat dilakukan secara rekursif dengan "**N**" yang baru adalah **G**.

  KASUS 4 :

  Pada kasus ini, SIMPUL **P** BERWARNA MERAH TETAPI SIMPUL **U** BERWARNA HITAM; JUGA BERLAKU SIMPUL **N** ADALAH ANAK KANAN DARI **P**, SEMENTARA **P** ADALAH ANAK KIRI DARI **G**. Dalam pada kasus ini, *left rotation* (pertukaran antara simpul **N** dan simpul **P**) dapat dilaksanakan untuk kemudian simpul **P** masuk ke dalam kasus 5 (penamaan ulang simpul **N** dan **P**) dikarenakan sifat (4) masih tetap terlanggar. Pada rotasi ini,

karena kedua simpul berwarna merah, maka sifat (5) tidak terganggu.

Catatan : untuk **P** sebagai anak kanan, *left* dan *right* tinggal dipertukarkan.

KASUS 5 :

SIMPUL **P** BERWARNA MERAH TETAPI SIMPUL **U** BERWARNA HITAM, SIMPUL **N** ADALAH ANAK KIRI DARI P, DAN **P** JUGA ADALAH ANAK KIRI DARI **G**. Dalam pada kasus ini, *right rotation* terhadap simpul **G** dilakukan; hasilnya adalah pohon dimana **P** menjadi ayah dari baik simpul **N** maupun simpul **G**. Simpul **G** pastilah berwarna hitam karena jika tidak, **P** tidak mungkin berwarna merah. Dengan demikian, yang perlu dilakukan adalah menukar warna antara **P** dan **G** sehingga sifat (4) terpenuhi. Sementara itu, sifat (5) tetap terjaga karena ketika sebelumnya hanya terdapat satu buah simpul hitam **G**, maka sekarang juga hanya terdapat satu buah simpul hitam **P**.

Catatan : untuk **P** sebagai anak kanan, *left* dan *right* tinggal dipertukarkan.

- Penghapusan

Pada BST biasa, penghapusan sebuah simpul dengan kedua anak yang bukan daun, dilakukan dengan pertama-tama mencari elemen dengan nilai maksimum pada upapohon kirinya atau elemen dengan nilai minimum pada upapohon kanannya, untuk kemudian memindahkan nilai tersebut ke simpul yang dibuang. Simpul yang nilainya disalin ini kemudian dibuang (simpul ini pasti memiliki anak-bukan-daun kurang dari dua). Karena proses penghapusan yang demikian tidaklah mengganggu validitas pohon *Red-black*, kasus yang perlu diperhatikan selanjutnya hanyalah penghapusan simpul dengan anak-bukan-daun sebanyaknya satu saja.

Ketika sebuah simpul merah dibuang, kita dapat dengan mudahnya mengganti simpul tersebut dengan simpul anak yang pasti berwarna hitam. Dalam proses ini, sifat (3) dan (4) akan tetap terjaga karena perubahan pada lintasan yang terjadi hanyalah berkurangnya sebuah simpul merah yang dilalui. Kasus khusus lainnya adalah ketika simpul yang dibuang berwarna hitam dan anaknya berwarna merah. Dengan hanya memindahkan simpul tersebut akan berdampak kepada kerusakan sifat (4) dan (5), tetapi dengan pewarnaan ulang simpul anak menjadi hitam, masalah terselesaikan.

Kasus sesungguhnya yang cukup kompleks untuk diselesaikan adalah ketika simpul yang ingin dihilangkan dan anaknya sama-sama berwarna hitam. Penyelesaian ini dimulai dengan menggantikan simpul yang akan dibuang dengan anaknya.

Catatan : N −> simpul anak yang telah menggantikan simpul yang ingin dibuang; P −> ayah N; S −> saudara N; S_L −> anak kiri S dan S_R −> anak kanan S karena S tidak mungkin daun. Warna putih dapat berarti hitam saja atau merah saja. Dalam bahasa C, simpul S dapat ditemukan dengan cara :

```
struct node *
sibling(struct node *n)
{
   if (n == n->parent->left)
      return n->parent->right;
   else
      return n->parent->left;
}
```

Fungsi `replace_node` menggantikan `child` ke tempat **N**.

```
void
delete_one_child(struct node *n)
{
   /*
    * Precondition: n has at most one non-
null child.
    */
   struct node *child = is_leaf(n->right) ?
n->left : n->right;

   replace_node(n, child);
   if (n->color == BLACK) {
      if (child->color == RED)
         child->color = BLACK;
      else
         delete_case1(child);
   }
   free(n);
}
```

Dengan perlakuan yang seperti ini, sifat (5) akan menjadi terlangkahi karena simpul yang dibuang berwarna hitam sehingga jumlah simpul hitam berkurang satu dan berakibat pada perbedaan panjang lintasan yang melalui simpul tersebut dan yang tidak.

KASUS 1 :

SIMPUL **N** MENJADI AKAR POHON *RED-BLACK*; dalam kasus ini, masalah selesai. Semenjak penghapusan ini mengurangi satu simpul hitam dari seluruh lintasan sekaligus, sifat (5) akan tetap terjaga. Dan akar yang baru berwarna hitam membuat seluruh sifat terpenuhi.

Catatan : Pada kasus 2, 5, dan 6, diasumsikan bahwa **N** adalah anak kiri dari **P**. Jika **N** adalah anak kanan, *left* dan *right* tinggal dipertukarkan.

KASUS 2 :

S BERWARNA MERAH; dalam kasus ini, kita menukarkan warna antara **P** dan **S**, untuk lalu kemudian melakukan rotasi kiri pada **P**, menyebabkan S menjadi ayah **P**.

KASUS 3 :

P, **S**, DAN ANAK DARI **S** BERWARNA HITAM; dalam kasus ini, kita cukup mewarnai ulang **S** menjadi merah. Pohon yang dihasilkan menyebabkan semua lintasan melalui **S**, yang berarti pasti tidak melalui **N**, memiliki lebih sedikit satu simpul hitam sebanyak satu buah. Faktanya adalah, pembuangan ayah asli dari **N** juga mengurangi satu simpul hitam pada setiap lintasan yang melalui **N**, sehingga masalah ini menjadi terpecahkan. Akan tetapi, sekarang semua lintasan yang melalui **P** menjadi

memiliki kurang satu buah simpul hitam dibandingkan dengan lintasan yang tidak melalui **P**, sehingga sifat (5) dilanggar. Untuk menyelesaikannya, dilakukan penyeimbangan ulang terhadap **P** dimulai dari kasus 1.

KASUS 4 :

S DAN ANAK DARI **S** BERWARNA HITAM, TETAPI **P** BERWARNA MERAH; dalam kasus ini, kita cukup menukarkan warna antara **P** dan **S**. Hal ini tidak akan berpengaruh terhadap jumlah dari simpul hitam pada lintasan yang melalui **S**, akan tetapi menambahkan satu simpul pada lintasan yang melalui N untuk menggantikan simpul yang dibuang.

KASUS 5 :

S BERWARNA HITAM, $\mathbf{S_L}$ BERWARNA MERAH, $\mathbf{S_R}$ BERWARNA HITAM, DAN **N** ADALAH ANAK KIRI; dalam kasus ini, kita melakukan rotasi kanan pada **S** sehingga$\mathbf{S_L}$ menjadi ayah **S** dan saudara baru dari **N**. Pertukaran warna juga dilakukan antara **S** dan ayah barunya. Seluruh lintasan akan mempunyai jumlah simpul hitam yang tetap, akan tetapi **N** kini mempunyai saudara yang mana anak kanannya berwarna merah (kasus 6). Baik N maupun ayahnya tidak terpengaruh oleh transformasi ini.

KASUS 6 :

S BERWARNA HITAM, $\mathbf{S_R}$ BERWARNA MERAH, DAN **N** ADALAH ANAK KIRI DARI **P**; dalam kasus ini, kita melakukan rotasi kiri pada **P** sehingga **S** menjadi ayah **P** dan $\mathbf{S_R}$. Pertukaran warna kemudian dilakukan antara **P** dan **S**, sementara $\mathbf{S_R}$ dibuat menjadi hitam. Upapohon yang terbentuk akan tetap mempunyai akar yang berwarna sama sehingga sifat (4) dan (5) tidaklah dilanggar. Akan tetapi, **N** kini mempunyai tambahan satu buah leluhur yang berwarna hitam : mungkin **P** yang menjadi hitam atau tambahan **S** yang juga hitam.

Sementara itu, lintasan yang tidak melalui **N** mempunyai dua buah kemungkinan :

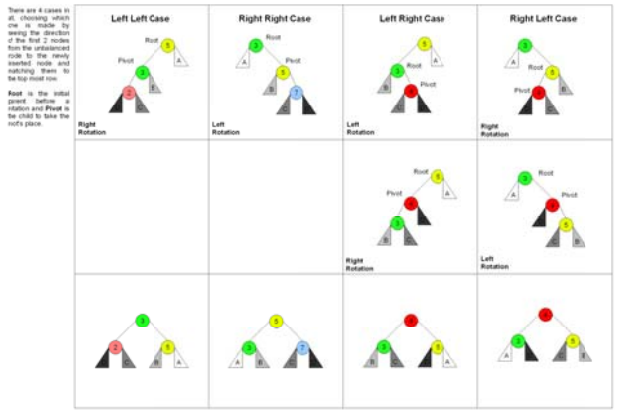
- lintasan melalui saudara baru **N** sehingga berarti melalui **S** dan **P** yang mana hanya bertukar tempat dan warna, sehingga lintasan tetap memilki jumlah simpul hitam yang tetap.
- lintasan melalui paman baru dari **N**, $\mathbf{S_R}$ sehingga berarti yang semula melalui **S**, ayah **S**, dan anak kanan **S** yang awalnya merah; menjadi melalui lintasan baru yang hanya melewati **S** yang kini warnanya mengikuti ayah lamanya dan $\mathbf{S_R}$ yang warnanya telah berubah menjadi hitam. Hasil akhirnya adalah jumlah simpul hitam yang dilalui berjumlah sama.

Yang manapun yang terjadi, pada akhirnya jumlah simpul hitam pada lintasan tidaklah berubah sehingga sifat (4) dan sifat (5) terjaga.

AVL TREE[13]

Pohon *AVL* memiliki struktur yang menyerupai BST tetapi ditambahkan dengan *balance factor*. *Balance factor* dari suatu simpul adalah kedalaman dari upapohon kanan dikurangi kedalaman dari upapohon kiri sehingga pohon yang seimbang adalah mereka yang memiliki *balance factor* bernilai -1, 0, atau 1. Pohon yang tidak seimbang akan mengalami rotasi.

- Penyisipan (Basis 0)



Gambar 4.11. Proses Penyeimbangan

Jika **P** adalah akar pohon tak seimbang, **L** adalah anak kiri, dan **R** adalah anak kanan, maka keempat kasus yang harus ditangani meliputi :

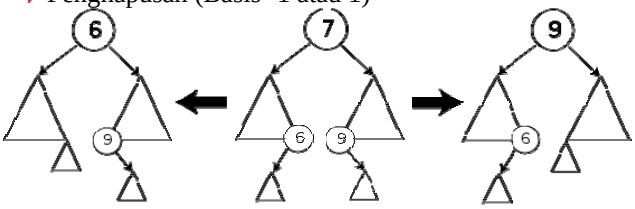
KASUS KANAN-KANAN ATAU KASUS KANAN-KIRI :

BALANCE FACTOR DARI **P** ADALAH -2; jika *balance factor* dari **R** adalah -1, maka rotasi kiri pada **P** dilakukan, sementara jika *balance factor* dari **R** adalah +1, maka rotasi dua kali harus dilakukan, yakni pertama rotasi kanan pada **R** lalu dilanjutkan dengan rotasi kiri pada **P**.

KASUS KIRI-KIRI ATAU KASUS KIRI-KANAN:

BALANCE FACTOR DARI **P** ADALAH +2; jika *balance factor* dari **L** adalah +1, maka rotasi kanan pada **P** dilakukan, sementara jika *balance factor* dari **L** adalah -1, maka rotasi dua kali harus dilakukan, yakni pertama rotasi kiri pada **L** lalu dilanjutkan dengan rotasi kanan pada **P**.

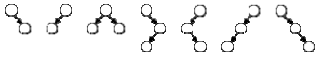
- Penghapusan (Basis -1 atau 1)



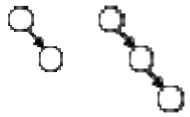
Gambar 4.12. Proses Penghapusan dengan Penggantian oleh Inorder Predecessor atau Inorder Successor

AA TREE[14]

Pohon *AA* adalah merupakan variasi dari pohon *Red-black* dimana simpul merah hanya dapat ditambahkan sebagai anak kanan dengan demikian menyederhanakan proses pemeliharaan keseimbangannya.



(a) Kasus pada Pohon *Red-black*

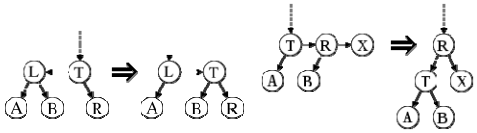


(b) Kasus pada Pohon *AA*
Gambar 4.13.

Karakteristik pohon *AA* :

1. Level dari daun adalah satu.
2. Level dari anak kiri pastilah lebih kecil dari ayahnya.
3. Level dari anak kanan dapat lebih kecil dari atau sama dengan ayahnya.
4. Level dari cucu kanan pastilah lebih kecil dari kakeknya.
5. Setiap simpul dengan level di atas satu pasti memiliki dua anak.

Hanya ada dua macam operasi untuk mempertahankan keseimbangan pohon *AA*, yakni *skew* dan *split*. *Skew* adalah rotasi kanan yang dilakukan ketika penyisipan atau penghapusan menghasilkan *left horizontal link* (*left red link* pada pohon *Red-black*) sementara *split* adalah suatu pengkondisian rotasi kiri akibat terbentuknya dua buah *horizontal right links* (*two consecutive red links* pada pohon *Red-black*).



Gambar 4.14. (a) *Skew* (b) *Split*

- Penyisipan

  Penyisipan dilakukan seperti terhadap BST standar. Jika terjadi sebuah *horizontal left link*, operasi *skew* akan dilakukan; sementara jika terjadi dua buah *horizontal right links*, operasi *split* akan dilakukan bahkan mungkin peningkatan level dari akar baru dari upapohon ini.

- Penghapusan

  Penghapusan dilakukan sama seperti pada BST, dimana suksesor (yang pada BST ditemukan dengan cara mengambil lintasan melalui semua anak kiri dari anak kanan simpul) dan predesesor (yang pada BST ditemukan dengan cara mengambil lintasan melalui semua anak kanan dari anak kiri simpul) adalah simpul dengan level 1, sehingga mudah dilakukan.

  Sesudah penghapusan, langkah pertama yang harus dilakukan untuk menjaga sifat pohon *AA* adalah menurunkan level dari semua simpul yang mana anaknya berada pada dua tingkat di bawah mereka, atau simpul-simpul yang kehilangan anaknya. Selanjutnya, pada seluruh level dilakukan operasi *skew* dan *split*.

  - Menurunkan level, jika diperlukan
  - Melakukan operasi *skew* pada level
  - Melakukan operasi *split* pada level

SPLAY TREE [15]

Pohon *Splay* adalah merupakan *self-balancing* BST yang mana memiliki karakteristik tambahan berupa elemen yang baru diakses dapat diakses ulang dengan cepat.

- *Splaying*

  Ketika simpul X diakses, operasi *splay* dilakukan untuk memindahkannya ke posisi akar. Adapun *splay steps* dipengaruhi oleh tiga buah faktor berikut ini :

  - **X** sebagai anak kiri atau anak kanan dari simpul **P**.
  - **P** sebagai akar,
  - atau jika bukan, **P** sebagai anak kiri atau anak kanan dari simpul **G**.

  Tipe-tipe *splay steps* :

  - *Zig Step* :

    Langkah ini dilakukan ketika **P** adalah akar dengan cara rotasi pada sisi antara **X** dan **P**. *Zig step* dilakukan hanya pada akhir dari operasi *splay* ketika **X** memiliki tingkat kedalaman yang ganjil pada saat awal operasi.
  - Zig-zig Step :

    Langkah ini dilakukan ketika **P** bukan akar dan baik **X** maupun **P** adalah sama-sama anak kiri atau sama-sama anak kanan. Langkah ini dilakukan dengan cara rotasi pada sisi antara **P** dan **G**, lalu kemudian dilanjutkan dengan rotasi pada sisi antara **X** dan **P**. *Zig-zig step* adalah satu-satunya perbedaan antara pohon *Splay* dengan metode *rotate to root.*
  - Zig-zag Step :

    Langkah ini dilakukan ketika **P** bukan akar dan **X** adalah anak kanan sementara **P** adalah anak kiri atau sebaliknya. Langkah ini dilakukan dengan cara rotasi pada sisi antara **X** dan **P**, lalu kemudian dilanjutkan dengan rotasi pada sisi antara **X** dan **G**.

- Penyisipan

  Pertama-tama, **X** dicari pada pohon *Splay*; jika tidak ditemukan, maka akan dicari simpul **Y**, ayah **X**. Langkah selanjutnya adalah melakukan operasi *splay* pada **Y** untuk lalu kemudian menyisipkan **X** sebagai akar dengan cara yang tepat. Dengan demikian **Y** akan menjadi anak kiri atau anak kanan dari **X**.

- Penghapusan

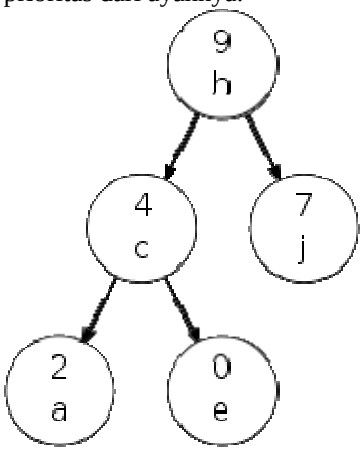
  - Lakukan operasi *splay* pada simpul X.
  - Lakukan operasi *tree rotation* pada anak kiri pertama dari **X** sehingga anak kiri tersebut tidak memiliki anak kanan.
  - Hapus **X** dari pohon dan gantikan akar dengan anak kiri pertama dari **X**.

SCAPEGOAT TREE [16]

TREAP [17]

Treap adalah gabungan antara *tree* dan *heap* dan merupakan pohon *Cartesian* dimana setiap *key* (dipilih secara acak) diberikan prioritas numerik. Urutan *inorder traversal* dari simpul sama dengan urutan *key* yang terurut. Struktur pohon ini ditentukan oleh kebutuhan bahwa pohon haruslah *heap-ordered* : yaitu nilai prioritas

dari semua simpul bukan akar harus lebih besar atau sama dengan nilai prioritas dari ayahnya.



Gambar 4.21. *Treap*

Operasi pada *Treap* :

- Proses pencarian sama seperti pada BST dengan tidak memperhatikan prioritas.
- Penyisipan *key* baru **X** pada *Treap* dilakukan dengan membangkitkan y, suatu prioritas sembarang pada **X**. Selama **X** bukan akar dan memiliki prioritas lebih kecil dari **Z**, ayahnya, dilakukan *tree rotation* untuk menukarkan posisi keduanya.
- Penghapusan simpul **X** dilakukan dengan cara menghilangkannya jika ia adalah daun; menggantikannya dengan **Z**, anak tunggal **X**; atau menukarkan posisinya dengan **Z**, suksesor langsung **X** dalam urutan, jika **X** memiliki dua anak. Dalam kasus yang terakhir, mungkin diperlukan operasi rotasi tambahan untuk menjaga sifat *heap-ordering*.
- Membagi *Treap* menjadi dua bagian di mana yang satu menjadi lebih kecil dari suatu nilai *key* **X** dan yang lain lebih besar dapat dilakukan dengan cara penyisipan **X** dengan prioritas terkecil. Proses ini akan menjadikan **X** sebagai akar, upapohon kiri sebagai bagian yang lebih kecil, dan upapohon kanan sebagai bagian yang lebih besar.
- Menggabungkan dua buah *Treap*, dapat dilakukan dengan mengasumsikan bahwa nilai terbesar pada *Treap* yang satu lebih kecil daripada nilai terkecil pada *Treap* lainnya dan melakukan operasi penyisipan **X** yang lebih besar dari nilai maksimum *Treap* yang satu tetapi lebih kecil dari nilai minimum *Treap* yang lain dan memiliki prioritas maksimum. Setelah penyisipan, **X** akan menjadi daun dan mudah untuk dihilangkan.

Untuk hasil yang lebih baik, prioritas yang lebih kecil diberikan kepada simpul yang sering diakses dengan cara memberikan prioritas acak yang lebih kecil dari prioritas semula.

V. KESIMPULAN

Self-balancing binary search tree memiliki operasi tambahan, yaitu operasi pemeliharaan, pada proses penyisipan ataupun penghapusan untuk menjaga strukturnya agar tetap terurut dan seimbang. Akan tetapi, operasi ini tidaklah mengubah nilai kompleksitas waktu secara keseluruhan dari proses penyisipan maupun pemeliharaan, atau dengan kata lain kompleksitasnya tetap O(log *n*). Namun demikian, proses lainnya –seperti pencarian–, menjadi lebih cepat dilaksanakan yaitu hanya pada tingkat O(log *n*) saja, dimana semula adalah O(n). Sebagai acuan, untuk n = 1.000.000, log n = 19.

Jadi, dapat dikatakan struktur data yang optimum untuk suatu proses pencarian adalah dengan menggunakan *self-balancing binary search tree*.

REFERENSI

[1] Liem, Inggriani, “Diktat Struktur Data”, Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung, 2008.
[2] Munir, Rinaldi, “Diktat Kuliah IF2091 Struktur Diskrit”, Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung, 2008.
[3] http://taghyr.wordpress.com/2009/08/15/tahun-2010-kapasitas-harddisk-akan-tembus-3tb/ (Tanggal Akses : 17 Desember 2009; 17.00)
[4] http://en.wikipedia.org/wiki/Tree_%28data_structure%29 (Tanggal Akses : 17 Desember 2009; 18.24)
[5] http://kuliah.itb.ac.id/mod/resource/view.php?id=5216 (Tanggal Akses : 17 Desember 2009; 20:50)
[6] http://en.wikipedia.org/wiki/Binary_tree (Tanggal Akses : 17 Desember 2009; 18.26)
[7] http://en.wikipedia.org/wiki/Binary_search_tree (Tanggal Akses : 17 Desember 2009; 20.44)
[8] http://www.informatika.org/~fazat/IF2030/IF2030-PohonBiner_bagian2.pdf (Tanggal Akses : 17 Desember 2009; 20:52)
[9] http://en.wikipedia.org/wiki/B-tree (Tanggal Akses : 18 Desember 2009; 11.15)
[10] http://wiki.answers.com/Q/What_is_a_balanced_tree_in_data_structures (Tanggal Akses : 18 Desember 2009; 11.16)
[11] http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree (Tanggal Akses : 17 Desember 2009; 20.46)
[12] http://en.wikipedia.org/wiki/Red-black_tree (Tanggal Akses : 16 Desember 2009; 19.52)
[13] http://en.wikipedia.org/wiki/AVL_tree (Tanggal Akses : 16 Desember 2009; 19.52)
[14] http://en.wikipedia.org/wiki/AA_tree (Tanggal Akses : 16 Desember 2009; 19.51)
[15] http://en.wikipedia.org/wiki/Splay_tree (Tanggal Akses : 16 Desember 2009; 19.51)
[16] http://en.wikipedia.org/wiki/Scapegoat_tree (Tanggal Akses : 16 Desember 2009; 19.51)
[17] http://en.wikipedia.org/wiki/Treap (Tanggal Akses : 16 Desember 2009; 19.52)

2. STRUKTUR DATA

2.1 GRAF

Graf *G* didefinisikan sebagai pasangan himpunan (*V*, *E*), yang dalam hal ini :

- *V* = himpunan tidak-kosong dari simpul-simpul (*vertices* atau *node*) = $\{v_1, v_2, \dots, v_n\}$, dan
- *E* = himpunan sisi (*edges* atau *arcs*) yang menghubungkan sepasang simpul = $\{e_1, e_2, \dots, e_n\}$

atau dapat ditulis singkat notasi *G = (V, E)* [2].

Jenis-jenis Graf[2]

Berdasarkan ada tidaknya gelang atau sisi ganda pada suatu graf, maka secara umum graf dapat digolongkan menjadi dua jenis :

- Graf sederhana (*simple graph*)
  Graf yang tidak mengandung gelang maupun sisi-ganda dinamakan graf sederhana.
- Graf tak-sederhana (*unsimple-graph*)
  Graf yang mengandung sisi ganda atau gelang dinamakan graf tak-sederhana. Ada dua macam graf tak-sederhana, yaitu **graf ganda** (*multigraph*) dan **graf semu** (*pseudograph*). Graf ganda adalah graf yang mengandung sisi ganda saja. Sisi ganda yang menghubungkan sepasang simpul bisa lebih dari dua buah. Graf semu adalah graf yang mengandung gelang (termasuk bila memiliki sisi ganda sekalipun).

Sisi pada graf dapat menpunyai orientasi arah. Berdasarkan orientasi arah pada sisi, maka secara umum graf dibedakan atas dua jenis :

- Graf tak-berarah (*undirected graph*)
  Graf yang sisinya tidak mempunyai orientasi arah disebut graf tak-berarah. Pada graf tak-berarah, urutan pasangan simpul yang dihubungkan oleh sisi tidak diperhatikan. Jadi, $(v_j, v_k) = (v_k, v_j)$ adalah sisi yang sama.
- Graf berarah (*directed graph* atau *digraph*)
  Graf yang setiap sisinya diberikan orientasi arah disebut sebagai graf berarah. Sisi berarah disebut busur (*arc*). Pada graf berarah, (v_j, v_k) dan (v_k, v_j) menyatakan dua buah busur yang berbeda, dengan kata lain $(v_j, v_k) \neq (v_k, v_j)$. Untuk busur (v_j, v_k), simpul v_j dinamakan **simpul asal** (*initial vertex*) dan simpul v_k dinamakan **simpul terminal** (*terminal vertex*).

Terminologi Dasar[2]

- Lintasan (*Path*)
  Lintasan yang panjangnya *n* dari simpul awal v_0 ke simpul tujuan v_n di dalam graf *G* ialah barisan berselang-seling simpul-simpul dan sisi-sisi yang berbentuk $v_0, e_1, v_1, e_2, v_2, \dots, v_{n-1}, e_n, v_n$ sedemikian sehingga $e_1 = (v_0, v_1), e_2 = (v_1, v_2), \dots, e_n = (v_{n-1}, v_n)$ adalah sisi-sisi dari graf G.
- Terhubung (*Connected*)
  Graf tak-berarah *G* disebut **graf terhubung** (*connected graph*) jika untuk setiap pasang simpul v_i dan v_j di dalam himpunan *V* terdapat lintasan dari v_i ke v_j (yang juga harus berarti ada lintasan dari v_j ke v_i). Jika tidak, maka *G* disebut **graf tak-terhubung** (*disconnected graph*).
  Graf berarah *G* dikatakan terhubung jika graf tak-berarahnya terhubung (graf tak-berarah dari *G* diperoleh dengan menghilangkan arahnya).

2.2 POHON (TERMINOLOGI)

Pohon bebas adalah graf tak-berarah terhubung yang tidak mengandung sirkuit, dengan sifat sebagai berikut :

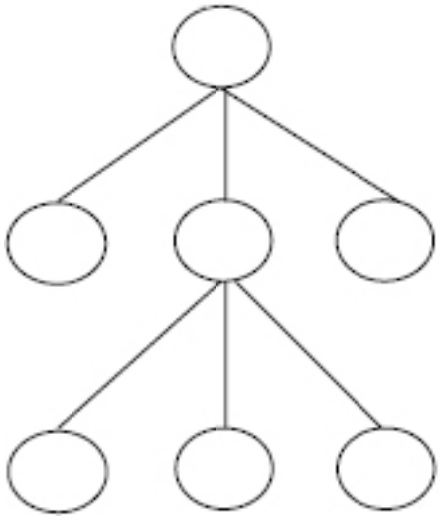
Misalkan *G = (V, E)* adalah graf tak-berarah sederhana dan jumlah simpulnya *n*. Maka, jika G adalah pohon bebas berlaku :

- Setiap pasang simpul di dalam G terhubung dengan lintasan tunggal,
- G terhubung dan memiliki *m = n – 1* buah jumlah sisi,
- G tidak mengandung sirkuit,
- Penambahan satu sisi pada graf akan membuat hanya satu sirkuit, dan
- Semua sisi G adalah jembatan (jembatan adalah sisi yang bila dihapus menyebabkan graf terpecah menjadi dua komponen)[2].

Pohon berakar (yang selanjutnya disebut hanya pohon) adalah pohon bebas yang sebuah simpulnya diperlakukan sebagai akar dan sisi-sisinya diberi arah sehingga menjadi graf berarah [2].

Pohon sebagai suatu struktur data rekursif adalah himpunan terbatas, tidak kosong, dengan elemen dibedakan sebagai berikut :

- Sebuah elemen yang dibedakan dari yang lain –> AKAR
- Elemen yang lain (jika ada) dibagi-bagi menjadi beberapa sub-himpunan yang disjoin dan masing-masing sub-himpunan itu adalah pohon –> SUBPOHON[5]



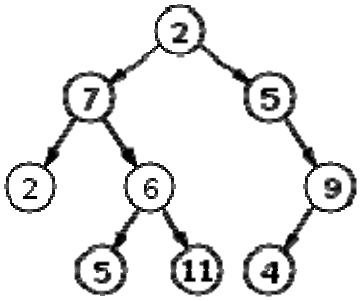
Gambar 2.1. Pohon

Pohon yang setiap simpul cabangnya mempunyai paling banyak *n* buah anak disebut **pohon *n*-ary**[2].

Pohon Biner[5]

Pohon biner merupakan kasus khusus pohon *n-ary* jika n = 2 dan adalah himpunan terbatas yang :

- mungkin **kosong**, atau
- terdiri atas sebuah simpul yang disebut **akar** dan dua buah himpunan lain yang *disjoint* yang merupakan pohon biner, yang disebut sebagai **sub pohon kiri** dan **sub pohon kanan** dari pohon biner tersebut.

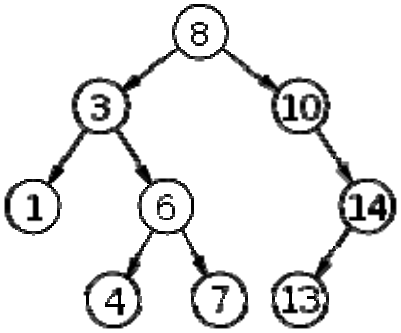


Gambar 2.2. Pohon Biner

Pohon Biner Terurut[8]

Pohon biner yang urutan anak-anaknya penting disebut pohon biner terurut atau pohon biner pencarian (*ordered binary tree* / *binary search tree* –***BST***–), dan memenuhi sifat :

- setiap simpul dalam BST mempunyai sebuah nilai
- subpohon kiri dan subpohon kanan merupakan BST
- jika P adalah sebuah BST :
  - semua simpul pada subpohon kiri < Akar(P)
  - semua simpul pada subpohon kanan > Akar(P)



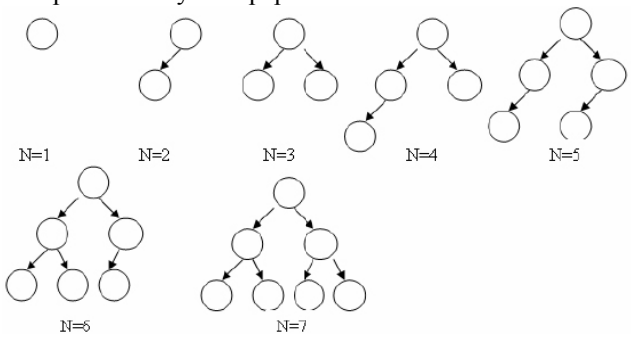
Gambar 2.3. Pohon Biner Terurut

Pohon Biner Seimbang[8]

Pohon biner seimbang (*balanced binary tree* / *B-tree*) adalah pohon dengan :

- perbedaan tinggi antara subpohon kiri dengan subpohon kanan maksimum 1.
- perbedaan banyaknya simpul subpohon kiri dan subpohon kanan maksimum 1.

Untuk menyeimbangkan tinggi upapohon (subpohon) kiri dan tinggi upapohon kanan –yaitu berbeda maksimal 1–, tinggi pohon secara keseluruhan harus, secara otomatis, dibuat seminimal mungkin. Untuk memperoleh tinggi minimum, setiap aras (level) harus mengandung jumlah simpul sebanyak mungkin. Hal ini dapat dibuat dengan cara menyebarkan setengah dari jumlah simpul di upapohon kiri dan setengah dari jumlah simpul yang merupakan sisanya di upapohon kanan[2].



Gambar 2.4. Pohon Biner Seimbang

RED-BLACK TREE / SYMMETRIC BINARY B-TREES[12]

- Penyisipan

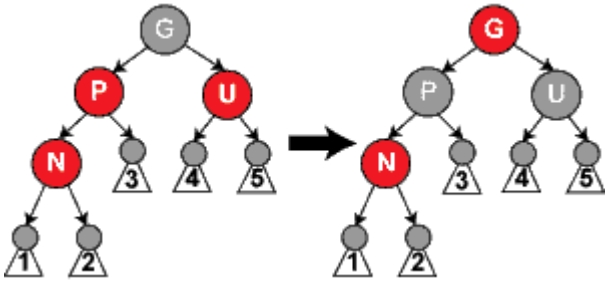
KASUS 1 :

```
void
insert_case1(struct node *n)
{
   if (n->parent == NULL)
      n->color = BLACK;
   else
      insert_case2(n);
}
```

KASUS 2 :

```
void
insert_case2(struct node *n)
{
   if (n->parent->color == BLACK)
      return; /* Tree is still valid */
   else
      insert_case3(n);
}
```

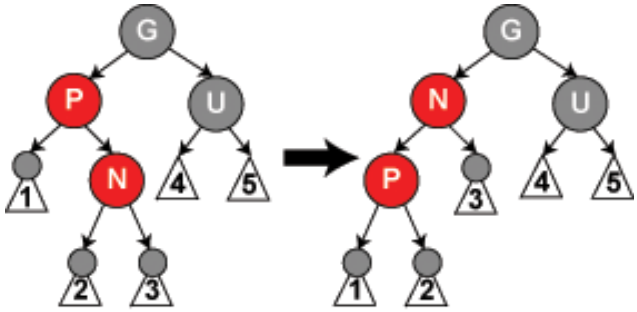
KASUS 3 :



Gambar 4.2. Kasus 3

```
void
insert_case3(struct node *n)
{
   struct node *u = uncle(n), *g;

   if ((u != NULL) && (u->color == RED)) {
      n->parent->color = BLACK;
      u->color = BLACK;
      g = grandparent(n);
      g->color = RED;
      insert_case1(g);
   } else {
      insert_case4(n);
   }
}
```

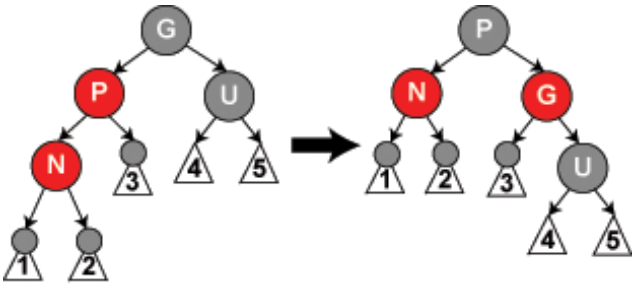
KASUS 4 :



Gambar 4.3. Kasus 4

```
void
insert_case4(struct node *n)
{
   struct node *g = grandparent(n);

   if ((n == n->parent->right) && (n->parent == g->left)) {
      rotate_left(n->parent);
      n = n->left;
   } else if ((n == n->parent->left) && (n->parent == g->right)) {
      rotate_right(n->parent);
      n = n->right;
   }
   insert_case5(n);
}
```

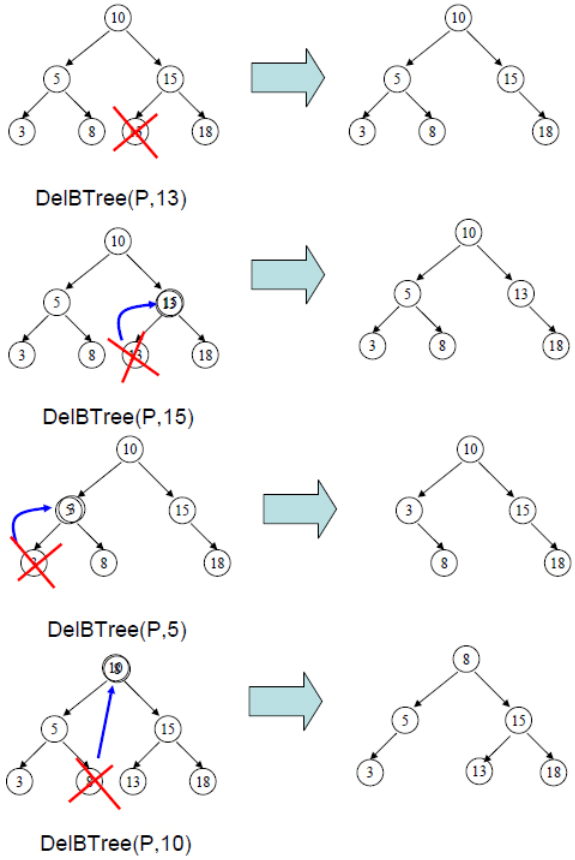
KASUS 5 :



Gambar 4.4. Kasus 5

```
void
insert_case5(struct node *n)
{
   struct node *g = grandparent(n);

   n->parent->color = BLACK;
   g->color = RED;
   if ((n == n->parent->left) && (n->parent == g->left)) {
      rotate_right(g);
   } else { /* (n == n->parent->right) and (n->parent == g->right) */
      rotate_left(g);
   }
}
```

- Penghapusan

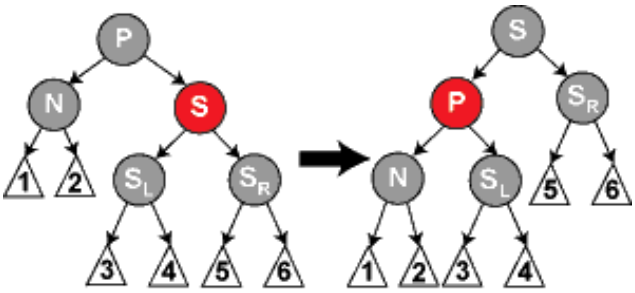


Gambar 4.5. Proses Penghapusan pada BST

KASUS 1 :

```
void
delete_case1(struct node *n)
{
   if (n->parent != NULL)
      delete_case2(n);
}
```

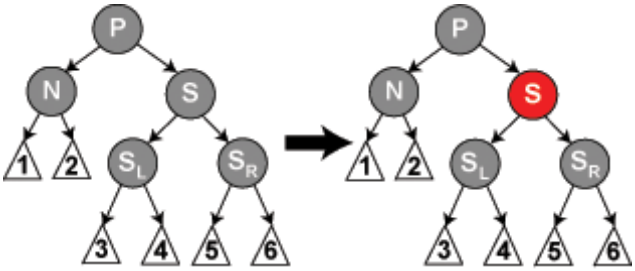
KASUS 2 :



Gambar 4.6. Kasus 2

```
void
delete_case2(struct node *n)
{
   struct node *s = sibling(n);

   if (s->color == RED) {
      n->parent->color = RED;
      s->color = BLACK;
      if (n == n->parent->left)
         rotate_left(n->parent);
      else
         rotate_right(n->parent);
   }
   delete_case3(n);
}
```

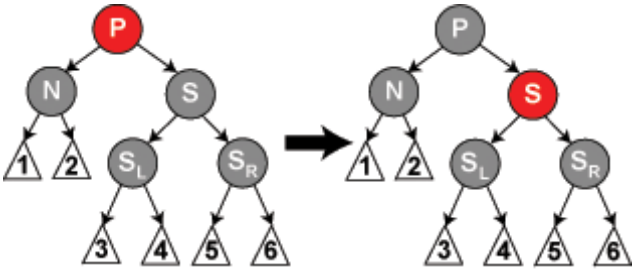
KASUS 3 :



Gambar 4.7. Kasus 3

```
void
delete_case3(struct node *n)
{
   struct node *s = sibling(n);

   if ((n->parent->color == BLACK) &&
       (s->color == BLACK) &&
       (s->left->color == BLACK) &&
       (s->right->color == BLACK)) {
      s->color = RED;
      delete_case1(n->parent);
   } else
      delete_case4(n);
}
```

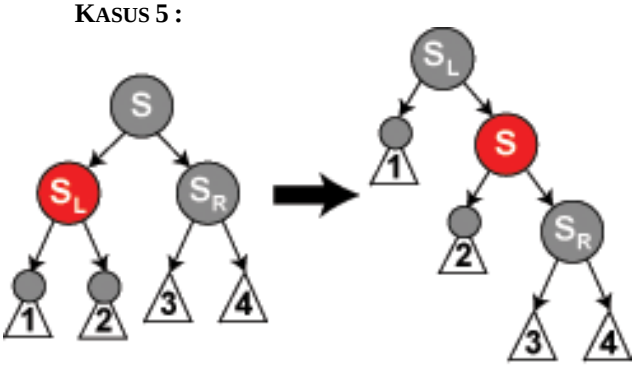
KASUS 4 :



Gambar 4.8. Kasus 4

```
void
delete_case4(struct node *n)
{
   struct node *s = sibling(n);

   if ((n->parent->color == RED) &&
       (s->color == BLACK) &&
       (s->left->color == BLACK) &&
       (s->right->color == BLACK)) {
      s->color = RED;
      n->parent->color = BLACK;
   } else
      delete_case5(n);
}
```

KASUS 5 :

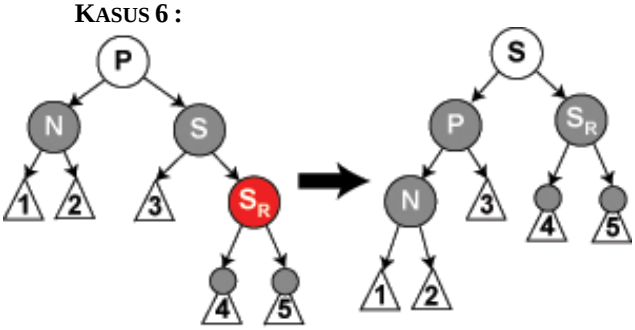


Gambar 4.9. Kasus 5

```
void
delete_case5(struct node *n)
{
   struct node *s = sibling(n);

   if  (s->color == BLACK) { /* this if
statement is trivial,
due to Case 2 (even though Case two changed
the sibling to a sibling's child,
the sibling's child can't be red, since no
red parent can have a red child). */
// the following statements just
force the red to be on the left of
the left of the parent,
// or right of the right, so case
six will rotate correctly.
      if ((n == n->parent->left) &&
          (s->right->color == BLACK) &&
          (s->left->color == RED)) { //
this last test is trivial too due
to cases 2-4.
         s->color = RED;
         s->left->color = BLACK;
         rotate_right(s);
      } else if ((n == n->parent->right) &&
                 (s->left->color == BLACK)
&&
```

```
                (s->right->color == RED))
{// this last test is trivial too
due to cases 2-4.
        s->color = RED;
        s->right->color = BLACK;
        rotate_left(s);
      }
   }
   delete_case6(n);
}
```

KASUS 6 :



Gambar 4.10. Kasus 6

```
void
delete_case6(struct node *n)
{
   struct node *s = sibling(n);

   s->color = n->parent->color;
        n->parent->color = BLACK;

   if (n == n->parent->left) {
                s->right->color = BLACK;
        rotate_left(n->parent);
   } else {
        s->left->color = BLACK;
        rotate_right(n->parent);
   }
}
```

AA TREE[14]

```
function skew is
    input: T, a node representing an AA tree that needs to be rebalanced.
    output: Another node representing the rebalanced AA tree.

    if nil(T) then
        return Nil
    else if level(left(T)) == level(T) then
        Swap the pointers of horizontal left links.
        L = left(T)
        left(T) := right(L)
        right(L) := T
        return L
    else
        return T
    end if
end function
```

```
function split is
    input: T, a node representing an AA tree that needs to be rebalanced.
    output: Another node representing the rebalanced AA tree.

    if nil(T) then
        return Nil
    else if level(T) == level(right(right(T))) then
        We have two horizontal right links.  Take the middle node, elevate it, and return it.
        R = right(T)
        right(T) := left(R)
        left(R) := T
        level(R) := level(R) + 1
        return R
    else
        return T
    end if
end function
```

- Penyisipan

```
function insert is
    input: X, the value to be inserted, and T, the root of the tree to insert it into.
    output: A balanced version T including X.

    Do the normal binary tree insertion procedure.  Set the result of the
    recursive call to the correct child in case a new node was created or the
    root of the subtree changes.
    if nil(T) then
        Create a new leaf node with X.
        return node(X, 1, Nil, Nil)
    else if X < value(T) then
        left(T) := insert(X, left(T))
    else if X > value(T) then
        right(T) := insert(X, right(T))
    end if
    Note that the case of X == value(T) is unspecified.  As given, an insert
    will have no effect.  The implementor may desire different behavior.

    Perform skew and then split.  The conditionals that determine whether or
    not a rotation will occur or not are inside of the procedures, as given
    above.
    T := skew(T)
    T := split(T)

    return T
end function
```

- Penghapusan

```
function decrease_level is
    input: T, a tree for which we want to remove links that skip levels.
    output: T with its level decreased.

    should_be = min(level(left(T)), level(right(T))) + 1
    if should_be < level(T) then
        level(T) := should_be
        if should_be < level(right(T)) then
            level(right(T)) := should_be
        end if
    end if
    return T
end function
```

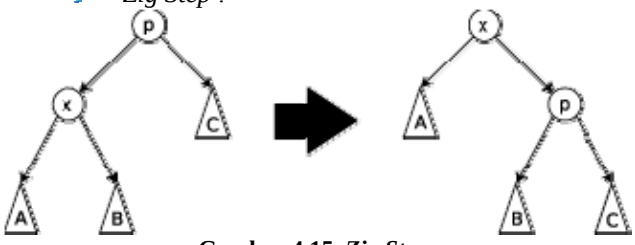
```
function delete is
    input: X, the value to delete, and T, the root of the tree from which it should be deleted.
    output: T, balanced, without the value X.

    if X > value(T) then
        right(T) := delete(X, right(T))
    else if X < value(T) then
        left(T) := delete(X, left(T))
    else
        If we're a leaf, easy, otherwise reduce to leaf case.
        if leaf(T) then
            return Nil
        else if nil(left(T)) then
            L := successor(T)
            right(T) := delete(L, right(T))
            value(T) := L
        else
            L := predecessor(T)
            left(T) := delete(L, left(T))
            value(T) := L
        end if
    end if

    Rebalance the tree.  Decrease the level of all nodes in this level if
    necessary, and then skew and split all nodes in the new level.
    T := decrease_level(T)
    T := skew(T)
    right(T) := skew(right(T))
    right(right(T)) := skew(right(right(T)))
    T := split(T)
    right(T) := split(right(T))
    return T
end function
```

SPLAY TREE[15]

- *Splaying*

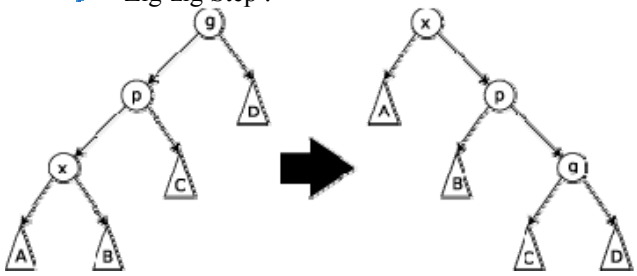
  Tipe-tipe *splay steps* :

  - *Zig Step* :



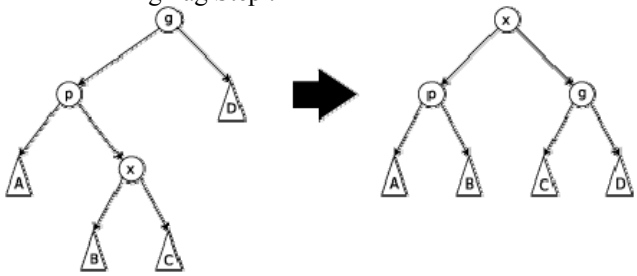
Gambar 4.15. *Zig Step*

  - Zig-zig Step :



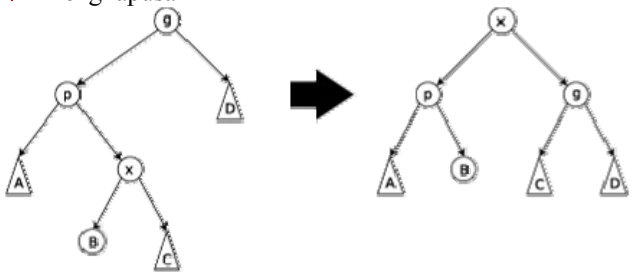
Gambar 4.16. *Zig-zig Step*

  - Zig-zag Step :

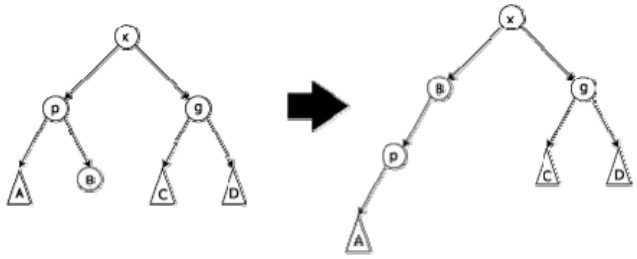


Gambar 4.17. *Zig-zag Step*

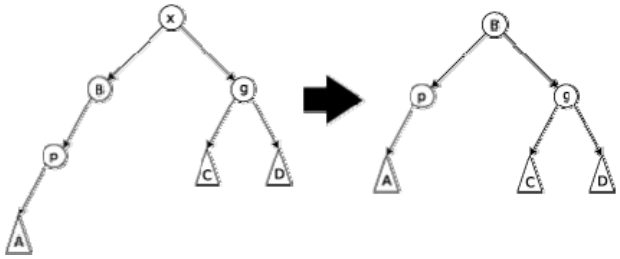
- Penghapusan



Gambar 4.18. Penghapusan Langkah 1



Gambar 4.19. Penghapusan Langkah 2



Gambar 4.20. Penghapusan Langkah 3