STUDI DAN REALISASI RED-BLACK TREE
(POHON MERAH HITAM)

Makalah

Oleh:

Harry Budiman – 13505061

Teknik Informatika, Institut Teknologi Bandung
Jl. Ganesha 10, Bandung

Abstrak

Makalah ini membahas tentang studi dan realisasi *Red-Black Tree* (Pohon Merah Hitam). Pohon merah hitam adalah suatu tipe dari *self-balancing binary tree*, struktur data yang digunakan dalam ilmu komputer, biasanya digunakan untuk mengimplementasikan *associative arrays*. Struktur aslinya ditemukan pada tahun 1972 oleh Rudolf Bayer yang menyebutnya "symmetric binary B-trees", namun baru mendapatkan nama modern dalam paper yang dibuat oleh Leo J. Guibas dan Robert Sedgewick pada tahun 1978. Struktur Pohon Merah Hitam (*Red-Black Tree* / RBT) ini memang kompleks namun memiliki running time yang baik untuk skenario kasus terburuk (*good worst-case scenario running time*) untuk operasi-operasinya dan efisien dalam penggunaan search, insert dan delete yang masing-masing memerlukan waktu O(log n), dengan n adalah jumlah elemen dalam pohon.

***Kata kunci**: Red-Black Tree*, dekripsi, realisasi

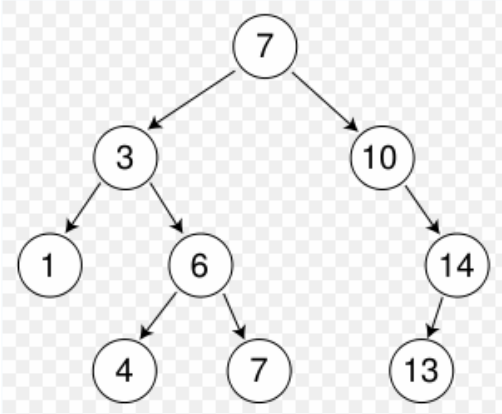
1.Pendahuluan

Dalam ilmu komputer, pohon biner adalah pohon struktur data yang tiap simpul mempunyai paling banyak 2 anak. Biasanya, disebut anak kiri dan kanan (*left, right*). Salah satu penggunaan umum dari pohon biner adalah *binary search tree* (BST).

Dalam ilmu komputer, *binary search tree* (BST) adalah pohon biner yang memiliki sifat sebagai berikut:

- tiap simpul memiliki nilai
- tiap nilai dari simpul disimpan secara terurut
- subpohon kiri hanya memiliki nilai yang lebih kecil dari nilai simpul
- subpohon kanan hanya memiliki nilai yang lebih besar dari nilai simpul

Kelebihan dari BST adalah algoritma sorting dan seracg yang memanfaatkan traversal terurut, hal ini bisa sangat efisien.



BST adalah struktur data fundamental yang digunakan untuk membentuk berbagai struktur data lainnya seperti himpunan, *multiset* dan *associative arrays*.

Jika sebuah BST terdiri dari nilai yang sama, maka representasi tersebut merupakan multiset. Jenis pohon ini menggunakan ketidaksamaan yang tidak ketat. Semua yang terdapat di subpohon kiri mempunyai nilai lebih kecil dari simpul sedangkan yang terdapat di pohon sebelah kanan bisa lebih besar atau sama dengan.

Jika sebuah BST tidak mempunyai nilai yang sama, maka pohon tersebut meprepresentasikan himpunan dengan nilai unik, seperti himpunan dalam konteks matematika. Pohon tanpa nilai duplikat menggunakan ketidaksamaan yang ketat, artinya subpohon kiri dari sebuah simpul hanya memiliki nilai yang lebih kecil dari simpul tersebut sedangkan subpohon kanan dari suatu simpul hanya memiliki nilai yang lebih besar dari simpul tersebut.

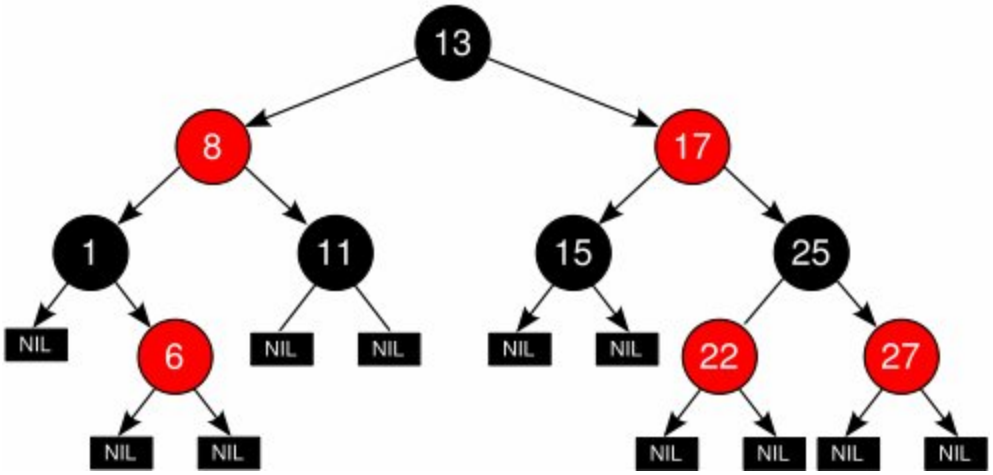
RBT, termasuk pohon AVL, menawarkan *best possible worst-case scenario* untuk *insertion time*, *deletion time*, dan *search time*. Kelebihan ini membuat RBT sangat berharga tidak hanya pada aplikasi-aplikasi yang sensitif terhadap waktu seperti aplikasi *real-time*, namun juga sebagai struktur data yang menjamin garansi kasus terburuk. Sebagai contoh, banyak struktur data yang digunakan pada komputasi geometri berdasarkan pada RBT.

RBT juga berguna dalam pemrograman fungsional karena RBT merupakan salah satu

struktur data yang paling "*persistent*", digunakan untuk membuat associative array dan himpunan yang bhisa mengambil kembali versi sebelumnya setelah perubahan. Versi "*persistent*" dari RBT memerlukan tambahan ruang O(log n) untuk setiap insersi atau delesi, selain waktu.

RBT isometri dari pohon 2-3-4. Artinya, untuk setiap 2-3-4 pohon, terdapat paling sedikit satu satu RBT dengan elemen data yang berurutan sama. Operasi insersi dan delesi pada pohon 2-3-4 equivalen dengan *color-flipping* (pertukaran warna) dan rotasi pada RBT. Ini membuat pohon 2-3-4 alat yang penting untuk memahami logika dibalik RBT, hal ini pula yang membuat berbagai text algoritma memperkenalkan pohon 2-3-4 sebelum RBT walaupun pohon 2-3-4 lebih jarang digunakan.

3.Properti



RBT adalah sebuah BST (*binary search tree*) dimana tiap simpul memiliki atribut warna yang bernilai merah atau hitam. Selain atribut yang dimiliki oleh BST, kita memerlukan persyaratan berikut untuk memnentukan validitas RBT :

1. Simpul berwarna hitam atau merah
2. Akar berwarna hitam
3. Semua daun berwarna hitam (termasuk anak-anak NIL)
4. Kedua anak dari tiap simpul merah harus berwarna hitam (dengan modus tollens, tiap simpul merah harus memiliki orang tua hitam)
5. Setiap jalan sederhana dari suatu simpul ke daun keturunan melewati jumlah simpul hitam yang sama.

Batasan-batasan ini memaksa properti kritis yang dimiliki oleh RBT: jalan terpanjang yang mungkin dari akar ke daun tidak lebih dari dua kali jalan terpendek yang mungkin. Hasilnya adalah pohon ini seimbang secara kasar. Karena operasi seperti insersi, delesi dan search memerlukan proporsi waktu kasus terburuk yang proporsional dengan tinggi dari pohon, pandangan teoretis berdasarkan tinggi pohon ini memungkinkan RBT lebih efisien dalam kasus terburuk dibandingkan dengan BST biasa.

4.Operasi

Read only operation (misal menuliskan isi pohon, *search*) pada RBT tidak memerlukan modifikasi dari yang digunakan oleh BST karena RBT merupakan spesialisasi dari BST sederhana. Namun insersi atau delesi secara langsung mungkin melanggar properti dari RBT. Memperbaiki properti RBT memerlukan O(log n) perubahan warna dan tidak lebih dari tiga rotasi (dua untuk insersi). Walaupun opearsi insert dan delete rumit, waktu pengerjaan mereka tetap O(log n)

4.1.Insertion

Kita mulai dengan menambahkan simpul seperti BST biasa dan memberinya warna merah. Yang dilakukan berikutnya tergantung dari warna simpul lainnya. Kita akan menggunakan istilah simpul paman untuk merujuk pada saudara orang tua simpul,

analogi dengani keluarga manusia. Catat bahwa :

- Syarat 3 selalu terpenuhi
- Syarat 4 terancam hanya jika menambahkan simpul merah, mewarnai simpul hitam menjadi merah, atau rotasi
- Syarat 5 terancam hanya jika menambahkan simpul hitam, mewarnai simpul merah menjadi hitam, atau rotasi

Catatan : Kita akan menggunakan label N untuk simpul yang akan dimasukkan (node), P untuk orang tua N (parent), G untuk kakek dari N (grandparent) dan U untuk paman dari N (uncle). Catat juga, pada kasus tertentu, kita menukar peran dan label dari simpul-simpul ini, namun pada setiap kasus, setiap label terus merepresentasikan simpul yang sama yang direpresentasikan pada awal kasus. Warna yang ditunjukkan pada diagram diasumsikan mengikuti syarat di atas.

Kita akan mendemonstrasikan tiap kasus dengan contoh kode C. simpul paman dan kakek dapat ditemukan oleh fungsi ini:

```
node grandparent(node n) {
    return n->parent->parent;
}

node uncle(node n) {
    if (n->parent == grandparent(n)->left)
        return grandparent(n)->right;
    else
        return grandparent(n)->left;
}
```

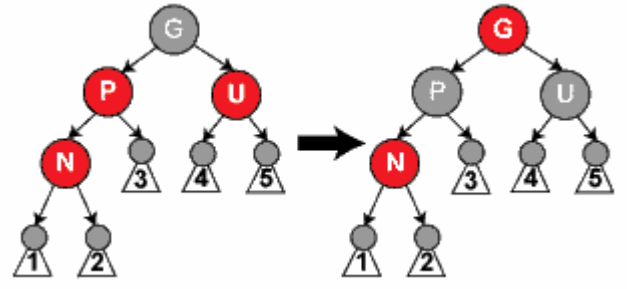
Kasus 1: simpul N terdapat di akar pohon. Maka kita mewarnainya dengan hitam untuk memenuhi persyaratan 2 (akar berwarna hitam). Karena tindakan ini menambahkan satu simpul hitam ke setiap jalan sekaligus, persyaratan 5 tidak dilanggar.

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

Kasus 2: Orang tua dari N, yaitu P berwarna hitam, jadi persyaratan 4 tidak dilanggar. Pada kasus ini, RBT masih valid. Persyaratan 5 juga tidak terancam, karena sinmpul baru N memiliki dua anak daun hitam, tapi karena N berwarna merah, jalan menuju tiap anak-anaknya memiliki jumlah yang sama dengan simpul hitam yang dilewati ketiak daun tersebut digantikan, yang berwarna hitam, jadi persyaratan ini tetap terpenuhi

```
void insert_case2(node n) {
    if (n->parent->color == BLACK)
        return; /* pohon masih valid */
    else
        insert_case3(n);
}
```

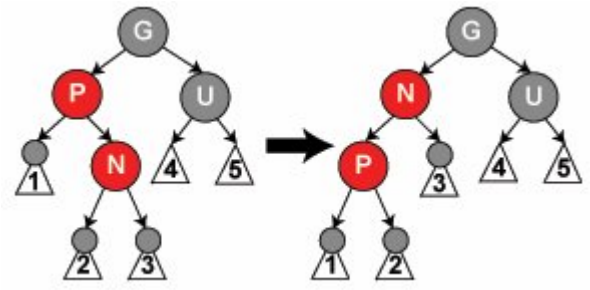
Catatan : pada kasus kasus berikutnya, kita mengasumsikan bahwa N memiliki simpul kakek G, karena orangtuanya, P, berwarna merah, dan jika G merupakan akar, maka ia berwarna hitam. Maka N juga memiliki paman U, walaupun mungkin menjadi daun pada kasus 4 dan 5.



Kasus 3: P dan U merah, maka kita mengecat ulang mereka menjadi hitam dan mengecat G menjadi merah (untuk mempertahankan persyaratan 5). Sekarang N memiliki orang tua hitam. Karena tiap jalan yang melewati orang tua atau paman harus melewati kakek, jumlah simpul hitam pada jalan ini belum berubah. Namun kakek G mungkin melanggar syarat 2 atau 4. Untuk memperbaikinya, kita melakukan prosedur secara rekursif dari pada G dari kasus 1. Catat bahwa ini hanya pemanggilan rekursif dan terjadi pada rotasi manapun, yang membuktikan bahwa jumlah rotasi konstan terjadi.

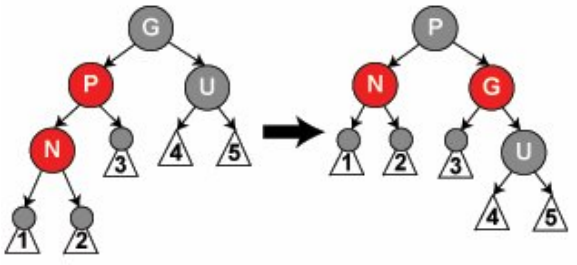
```
void insert_case3(node n) {
    if (uncle(n) != NULL && uncle(n)->color == RED) {
        n->parent->color = BLACK;
        uncle(n)->color = BLACK;
        grandparent(n)->color = RED;
        insert_case1(grandparent(n));
    }
    else
        insert_case4(n);
}
```

Catatan : pada kasus selanjutnya, ktia mengasumsikan bahwa P adalah anak kiri dari orangtuanya. Jika P merupakan anak kanan, kiri dan kanan seharusnya ditukar pada kasus 4 dan 5.



Kasus 4: P berwarna merah namun U berwarna hitam, juga N merupakan anak kanan dari P dan P adalah anak kiri dari G. Pada kasus ini, kita melakukan rotasi kiri yang menukar peran dari N dan P, maka kita mengatasi mantan orang tua P menggunakan kasus 5 (melabelkan ulang P dan N) karena syarat 4 masih dilanggar. Rotasi menyebabkan beberapa jalan (subtree 1) untuk melewati simpul baru yang tidak seperti sebelumnya, namun kedua simpul tersebut merah, jadi syarat 5 tidak dilanggar.

```
void insert_case4(node n) {
    if (n == n->parent->right && n->parent == grandparent(n)->left) {
        rotate_left(n->parent);
        n = n->left;
    } else if (n == n->parent->left && n->parent == grandparent(n)->right) {
        rotate_right(n->parent);
        n = n->right;
    }
    insert_case5(n);
}
```



Kasus 5: Orang tua P merah namun paman U hitam (“kelanjutan” dari kasus 4), simpul baru N adalah anak kiri dari P dan P adalah anak kiri dari kakek G. Pada kasus ini, kita melakukan rotasi kanan pada kakek G; hasilnya adalah pohon dimana bekas orang tua P sekarang orang tua dari simpul baru N dan kakek G. Kita tau bahwa G berwarna hitam karena jika tidak, mantan anak P berwarna merah. Kita kemudian menukar warna P dan G, dan hasilnya memenuhi syarat 4. Syarat 5 juga terpenuhi karena semua jalan yang melewati simpul manapun dari ke-3 simpul ini sebelumnya melewati G, sekarang melewati P. Pada kedua kondisi tersebut, ini adalah satu-satunya simpul hitam dari ke-3 simpul ini (N,P,G).

```
void insert_case5(node n) {
    n->parent->color = BLACK;
    grandparent(n)->color = RED;
    if (n == n->parent->left && n->parent == grandparent(n)->left) {
        rotate_right(grandparent(n));
    } else {
        /* Here, n == n->parent->right && n->parent == grandparent(n)->right */
        rotate_left(grandparent(n));
    }
}
```

Catat bahwa operasi insert sebenarnya bersifat *in-place,* karena semua pemanggilan di atas sebenarnya menggunakan rekursif ekor (*tail recursion*).

4.2.Deletion

Pada BST normal, ketika menghapus sebuah simpul dengan 2 anak-bukan-daun, kita mencari elemen maksimum pada subpohon kiri atau elemen minimum dari subpohon kanan lalu menmindahkan nilanya ke simpul yang sedang dihapus. Lalu kita menghapus simpul yang kita kopi nilai sebelumnya, yang harus memiliki kurang dari dua anak-bukan-daun. Karena mengkopi nilai tidak melanggar aturan syarat-syarat RBT, ini mengurangi persoalan menghapus menjadi masalah menghapus simpul dengan paling banyak satu anak. Tidak penting apakah simpul ini simpul yang pada awalnya ingin kita hapus atau simpul yang sudah kita kopi nilainya.

Untuk selanjutnya, kita mengasumsikan bahwa kita menghapus simpul yang paling banyak satu anak-bukan-daun, yang akan kita sebut anak. Jika kita menghapus simpul merah, kita bisa menggantikannya dengan anaknya, yang pasti berwarna hitam. Semua

jalan menuju simpul yang dihapus akan melewati satu kurangnya simpul merah dibanding sebelumnya dan orang tua dan anak simpul yang dihapus harus berwarna hitam, jadi syarat 3 dan 4 masih terpenuhi. Kasus yang lebih sederhana adalah menghapus simpul hitam dan anaknya merah. Hanya mengahpus simpul hitam tersebut akan melanggar syarat 4 dan 5 namun jika kita mengecat ulang anaknya menjadi hitam, kedua syarat itu akan dapat dipertahankan.

Kasus yang rumit adalah ketika kedua simpul ingin dihapus dan anaknya berwarna hitam. Kita mulai dengan mengganti simpul yang ingin dihapus dengan anaknya. Kita akan menyebut anaknya ini (pada posisi yang baru) sebagai N, dan saudaranya (sibling, anak lain dari orang tua barunya) S. Pada diagram di bawah, kita juga akan menggunakan P untuk orang tua baru N, SL untuk anak kiri S dan SR untuk anak kanan R (bisa ditunjukkan kalau S tidak mungkin daun)

Catatan : Pada beberapa kasus, kita menukar aturan dan label dari simpul-simpul, namun pada setiap kasus, setap label terus mempertahankan simpul yang sama yang dipertahankan pada awal kasus. Warna yang dipakai diasumsikan mengikuti syarat di atas. Putih merepresentasikan warna yang tidak diketahui (bisa hitam maupun putih).

Kita bisa menembukan saudara dengan menggunakan fungsi ini :

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

Catatan : Agar pohon tetap dapat terdefinisi, kita harus membuat tiap daun null tetap menjadi daun setelah semua transformasi. Jika simpul yang sedang kita hapus memiliki anak yang bukan daun, mudah melihat bahwa syarat tersebut terpenuhi. Namun, jika N akan menjadi daun null, ini bisa diverifikasi dari diagram (atau kode) untuk semua kasus bahwa syarat ini juga terpenuhi.

Kita bisa melakukan langkah-langkah di atas dengan kode berikut, dimana fungsi replace_node mengsubstitusi anak ke tempat N di pohon tersebut. Untuk kemudahan, kode pada bagian ini akan mengasumsikan bahwa daun null direpresentasikan oleh object sebenarnya bukan NULL (kode pada bagian Insertion bekerja dengan kedua representasi).

```
void delete_one_child(node n) {
    /*  Prekondisi:  n  harus  paling
banyak satu anak tidak null */
    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);
}
```

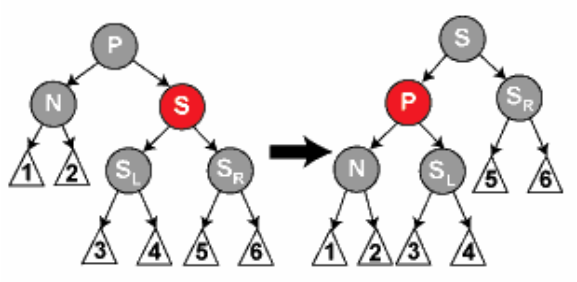
Catatan : Jika N adalah daun null dan kita tidak merepresemtasikan daun null sebagai obyek simpul, kita bisa memodifikasi algoritma dengan pertama-tama mencantumkan delete_case1() pada orangtuanya dan menghapusnya berikutnya. Kita bisa melakukan ini karena orangtuanya hitam jadi bersifat sama dengan daun null (kadang disebut daun hantu / *phantom leaf*). Dan kita bias dengan aman menhapusnya di akhir karena N akan tetap menjadi daun setelah semua operasi, seperti ditunjukkan diatas.

Jika N dan orang tua aslinya hitam, maka menghapus orang tua asli ini akan menyebabkan jalan yang melewati N memiliki jumlah simpul hitam satu lebih sedikit dibanding jalan yang tidak melewatinya. Karena ini melanggar syarat 5, pohon tersebut harus diseimbangkan ulang. Ada beberapa kasus yang perlu dipertimbangkan:

Kasus 1: N adalah akar baru. Pada kasus ini, kita selesai. Kita menghapus satu simpul hitam dari setiap jalan dan akar yang baru berwarna hitam, maka semua syarat dipertahankan

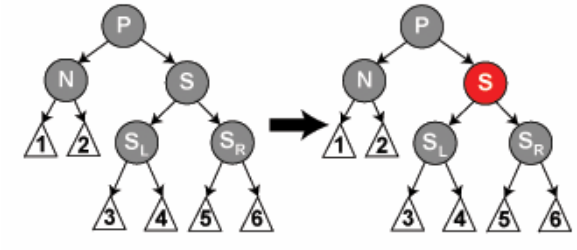
```
void delete_case1(node n) {
    if (n->parent == NULL)
        return;
    else
        delete_case2(n);
}
```

Catatan : Pada kasus 2, 5, dan 6 kita mengasumsikan N adalah anak kiri dari orang tua P. Jika N merupakan anak kanan, maka kiri dan kanan harus ditukar.



Kasus 2: S adalah merah. Pada kasus ini kita menukar warna P dan S, kemudian merotasi kiri di P, merubah S menjadi kakek N. Catat bahwa P harus hitam karena dia memiliki anak merah. Walaupun semua jalan masih memiliki jumlah simpul hitam yang sama, sekarang N memiliki saudara hitam dan orang tua merah, jadi kita bisa melanjutkan ke langkah 4,5 atau 8 (saudara barunya berwarna hitam karena dulunya dia adalah anak dari S merah) Pada kasus selanjutnya, kita akan melabelkan ulang S menjadi saudara baru N.

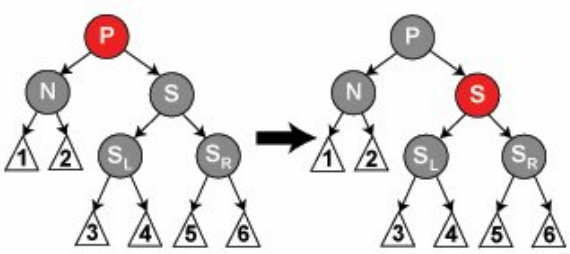
```
void delete_case2(node n) {
    if (sibling(n)->color == RED) {
        n->parent->color = RED;
        sibling(n)->color = BLACK;
        if (n == n->parent->left)
            rotate_left(n->parent);
        else
            rotate_right(n->parent);
    }
    delete_case3(n);
}
```



Kasus 3 : P, S dan anak-anak S adalah hitam. Pada kasus ini kita bisa mengecat ulang S menjadi merah. Hasilnya adalah semua jalan yang melewati S, yang tepatnya jalan tersebtu tidak melewati N, memiliki satu kurang simpul hitam, ini membuat keadaan menjadi seimbang. Namun semua jalan menuju P sekarang memiliki satu kurang simpul hitam daripada jalan yang tidak melewati P, jadi syarat 5 masih dilanggar. Untuk memperbaikinya, kita melakukang prosedure penyeimbangan ulang, mulai dari kasus 1.

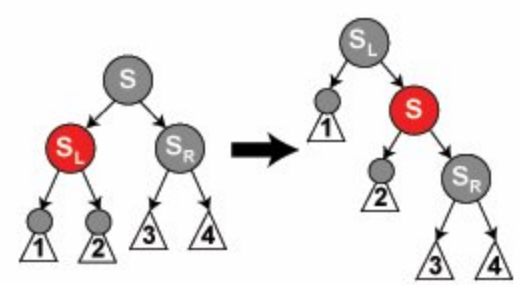
```
void delete_case3(node n) {
    if (n->parent->color == BLACK &&
        sibling(n)->color == BLACK &&
        sibling(n)->left->color      ==
BLACK &&
        sibling(n)->right->color     ==
BLACK)
    {
        sibling(n)->color = RED;
        delete_case1(n->parent);
    }
    else
        delete_case4(n);

}
```



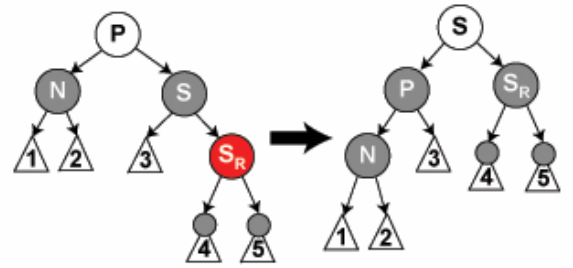
Kasus 4 : S dan anak-anak S berwarna hita,, namun P berwarna merah. Pada kasus ini, kita menukar warna S dan P. Ini tidak mempengaruhi jumlah simpul hitam di jalan yang tidak melewati N, namun menambah satu simpul hitam di jalan yang melewati N, seimbang dengan simpul hitam yang dihapus pada jalan tersebut.

```
void delete_case4(node n) {
    if (n->parent->color == RED &&
        sibling(n)->color == BLACK &&
        sibling(n)->left->color      ==
BLACK &&
        sibling(n)->right->color     ==
BLACK)
    {
        sibling(n)->color = RED;
        n->parent->color = BLACK;
    }
    else
        delete_case5(n);
}
```



Kasus 5: S hitam, anak kiri S merah, anak kanan S hitam, dan N adalah anak kiri dari orangtuanya. Pada kasus ini kita merotasi kanan di S, jadi anak kiri S menjadi orang tua S dan saudara baru N. Kita lalu menukar warna S dan orangtuanya. Semua jalan masih memiliki banyaknya simpul hitam yang sama, tapi sekarang N memiliki saudara hitam yang anak kanannya merah, jadi kita masuk ke kasus 6. Baik N maupun orang tuanya tiadk terpengaruh oleh transformasi ini (lagi, untuk kasus 6, kita memberi label ulang untuk S menjadi saudara baru N).

```
void delete_case5(node n) {
    if (n == n->parent->left &&
        sibling(n)->color == BLACK &&
        sibling(n)->left->color  ==  RED
&&
        sibling(n)->right->color     ==
BLACK)
    {
        sibling(n)->color = RED;
        sibling(n)->left->color       =
BLACK;
        rotate_right(sibling(n));
    }
    else if (n == n->parent->right &&
             sibling(n)->color == BLACK
&&
             sibling(n)->right->color
== RED &&
             sibling(n)->left->color ==
BLACK)
    {
        sibling(n)->color = RED;
        sibling(n)->right->color      =
BLACK;
        rotate_left(sibling(n));
    }
    delete_case6(n);
}
```



Kasus 6: S hitam, anak kanan S merah, dan N adalah anak kiri dari orangtuanya P. Pada kasus ini kita merotasi kiri di P, sehingga S menjadi orang tua dari P dan anak kanan S. Kita kemudian menukar warna P dan S dan menjadikan anak kanan S hitam. Subpohon masih mempunyai warna yang sama di akarnya jadi properti 4 dan 5 tidak dilanggar. Namun, N sekarang mempunyai satu leluhur hitam tambahan: antara P menjadi hitam atau P sebelumnya hitam dan S ditambahkan sebagai kakek hitam. Maka jalan menuju N melewati satu tambahan simpul hitam.

Sementara itu, jika jalan tidak melewati N, maka ada dua kemungkinan:

- jalan tersebut melewati saudara baru N. Maka ia pasti melewati S dan P yang hanya bertukar warna. Maka jalan tersebut memiliki jumlah simpul hitam yang sama.
- Jika melewati paman baru N, SR. Maka ia sebelumnya melewati S, orang tua S, dan SR, namun sekarang melewati S, yang telah mengambil warna dari mantan orangtuanya, dan SR yang sudah berubah warna dari merah ke hitam. Hasilnya jalan ini pun melewati jumlah simpul hitam yang sama.

Jalan manapun yang di[piilih, jumlah simpul hitam di jalan ini tidak berubah. Maka klita sudah memperbaharui syarat4 dan 5. Simpul putih di diagram bisa merah ataupun putih, tapi harus merujuk pada warna yang sa,a sebelum dan sesudah transformasi.

```
void delete_case6(node n) {
    sibling(n)->color    =    n->parent-
>color;
    n->parent->color = BLACK;
    if (n == n->parent->left) {
        /* Di sini, sibling(n)->right-
>color == RED */
        sibling(n)->right->color      =
BLACK;
        rotate_left(n->parent);
```

```
    }
    else
    {
        /* Di sini, sibling(n)->left-
>color == RED */
        sibling(n)->left->color       =
BLACK;
        rotate_right(n->parent);
    }
}
```

Lagi, pemanggilan fungsi menggunakan rekursif ekor (*tail recursion*) maka alogritmanya bersifat *in-place*. Sebagai tambahan, tidak ada pemanggilan rekursif lagi setelah rotasi jadi jumlah rotasi yang konstan (maksimal 3) berhasil diwujudkan.

5.Bukti *asymptotic bound*

Sebuat RBT yang mempunya n simpul dalam mempunyai ketinggian O(log n).

Definisi :

- h(v) = tinggi dari subpohon berakar di simpul v
- bh(v) = jumlah simpul hitam (tidak menghitung v jika hitam) dari v ke daun manapun di subpohon (*black-height*)

Lemma: Sebuah subpohon berakar di simpul v memiliki paling sedikit 2bh(v) – 1 simpul dalam.

Bukti dari Lemma (dengan induksi tinggi):

Basis: h(v) = 0;

Jika v mempunyai tinggi 0 maka pastilah nil, maka bh(v) = 0. Jadi:

$2^{bh(v)} - 1 = 2^0 - 1 = 1 - 1 = 0$

Hipotesis Induktif : v sedemikian hingga h(v) = k memiliki $2^{bh(v)} - 1$ simpul dalam, mengimpilikasikan bahwa v' sedeimikian hingga h(v') = k + 1 memiliki $2^{bh(v')} - 1$ simpul dalam

Karena v' memiliki h(v') > 0 adalah simpul dalam. Maka dia memiliki 2 anak yang memiliki black-height yang mempunyai bh(v') atau bh(v') (bergantung v' merah atau hitam). Dengan hipotesis induktif tiap anak memiliki paling sediki $2^{bh(v')} - 1 - 1$ simpul dalam, jadi v' memiliki:

$2^{bh(v')} - 1 - 1 + {}^{2bh(v')} - 1 - 1 + 1 = 2^{bh(v')} - 1$ simpul dalam.

Menggunakan lemma ini kita bias menunjukkan bahwa tinggi dari pohon bersifat logaritmik. Karena paling tidak setengah dari ssimpul di jalan manapun dari akar menuju daun adalah hitam (syarat 4), black-height dari pohon paling tidak h(root)/2. Dengan lemma ini kita mendapatkan

$$n \geq 2^{\frac{h(root)}{2}} - 1 \leftrightarrow \log_2(n+1) \geq \frac{h(root)}{2} \leftrightarrow h(root) \leq 2\log_2(n+1)$$

Maka tinggi dari akar adalah O(log n).

6.Kesimpulan

Di kode pohon hanya terdapat satu loop dimana simpul dari akar dari propertimerah-hitam yang ingin kita pulihkan,x, bisa dipindahkan naik ke atas pohon satu tingkat untuk setiap iterasi.

Karena tinggi sebenarnya dari pohon adalah O(log n), maka terdapat O(log n) iterasi. Jadi secara keseluruhan insert, delete dan search routine memiliki kompleksitas O(log n).

DAFTAR PUSTAKA

[1] Mathworld: Red-Black Tree, *http://mathworld.wolfram.com/Red-BlackTree.html.* Tanggal akses: 1 Januari 2007 pukul 15.00.

[2] Whitney, Roger. San Diego State University: CS 660: Red-Black tree notes. *http://www.eli.sdsu.edu/courses/fall95/cs660/notes/RedBlackTree/RedBlack.html#RTFToC.* Tanggal akses: 1 Januari 2007 pukul 15.00.

[3] Pfaff, Ben (June 2004). Performance Analysis of BSTs in System Software. StanfordUniversity. *http://www.stanford.edu/~blp/papers/libavl.pdf*. Tanggal akses: 1 Januari 2007 pukul 15.00.

[4] Red-Black Tree C Code. *http://web.mit.edu/~emin/www/source_code/red_black_tree/index.html*. Tanggal akses: 1 Januari 2007 pukul 15.00.

[5] Red-black tree – Wikipedia, the free encyclopedia. *http://en.wikipedia.org/wiki/Red-black_tree.* Tanggal akses: 1 Januari 2007 pukul 15.00.