Struktur Data Pohon Untuk Memodelkan Struktur Data Himpunan

Wiwit Rifa'i - 13513073[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]wiwit.rifai@students.itb.ac.id

***Abstrak*—Perkembangan ilmu informatika telah menghasilkan berbagai macam struktur data yang semakin mangkus dan dapat diimplementasikan dengan mudah dalam berbagai macam bahasa pemrograman. Bahkan struktur data yang satu bisa manfaakan struktur data yang lain. Himpunan merupakan salah satu struktur data yang sering ditemui dalam ilmu matematika diskrit. Ada berbagai macam cara merepresentasikan himpunan dalam suatu program, salah satunya adalah dengan memanfaatkan struktur data pohon. Makalah ini mengulas tentang bagaimana memanfaatkan struktur data pohon untuk merepresentasikan struktur data himpunan.**

***Keywords*—Struktu Data, Himpunan, *Union-Find Disjoint Sets*, Pohon AVL.**

I. PENDAHULUAN

Sekarang ini sudah ada begitu banyak struktur data yang bisa digunakan dalam membuat suatu program. Bahkan untuk beberapa bahasa pemrograman memiliki suatu *Standard Template Library* (*STL*) yang mencakup berbagai macam data struktur yang biasanya sering digunakan. Sehingga *STL* ini bisa digunakan secara langsung tanpa harus kita mengimplentasikan sendiri struktur data - struktur data tersebut. Beberpa contoh bahasa pemrograman yang sudah memiliki *STL* adalah C++ dan Java. Beberapa contoh struktur data yang sudah ada dalam *STL* C++ adalah vector (array dinamis), list(senarai), queue (antrian), stack (tumpukan), dan lain-lain.

Meskipun sudah terdapat *STL* yang tinggal dipakai tanpa harus memikirkan bagaimana cara mengimplementasikan struktur datanya, tetapi *programmer* yang baik tentu juga harus mengerti bagaimana *STL* tersebut diimplementasikan sehingga dia bisa menetahui kemangkusan dari data struktur tersebut.

Struktur data yang juga sering dipakai dalam membuat program adalah struktur data himpunan (*set*). Dalam matematika biasanya himpunan dinyatakan dengan mendaftarkan semua anggota dari himpunan tersebut satu per satu (*Roster Method*) atau dengan cara menyatakan karakteristik-karakteristik yang harus dipunyai oleh semua anggota-anggota himpunan tersebut (*Set Builder*). Namun himpunan yang dibahas dalam makalah ini adalah himpunan yang dinyatakan dengan cara mendaftarkan semua anggotanya (*Roster Method*) dan banyaknya anggota himpunan tersebut haruslah terbatas. Hal ini dikarenakan dengan cara inilah kita bisa mengimplementasikan struktur data himpunan dalam komputer yaitu dengan memakai struktur data yang bisa menyimpan sejumlah informasi mengenai anggota-anggota dari himpunan. Sedangkan menyatakan himpunan dengan cara menyatakan karakteristik semua anggotanya akan lebih sukar untuk diimplementasikan sebab karakteristik setiap himpunan akan berbeda dan dalam menuliskannya dalam kode pun tidak mudah. Dan jumlah anggota himpunan tersebut haruslah terbatas karena media penyimpanan yang disediakan oleh komputer untuk menyimpan data-data anggota himpunan tersebut juga terbatas.

Oleh karena itu struktur data yang bisa digunakan untuk merepresentasikan suatu himpunan adalah struktur data yang mampu menyimpan informasi-informasi dari anggota himpunan tersebut. Dan struktur data yang memenuhi hal tersebut diantaranya adalah struktur data lanjar (seperti array dan list), graf dan pohon. Struktur data pohon akan menjadi fokus utama dalam makalah ini untuk mengimplementasikan struktur data himpunan sebab terdapat beberapa kelebihan memanfaatkan pohon dibandingkan dengan struktur data lain.

Semua kode yang diberikan dalam makalah ini dituliskan dalam bahasa C. Penulis memilih bahasa C karena bahasa C adalah bahasa yang sudah cukup populer dan bisa digunakan dihampir semua platform, serta karena penulis lebih terbiasa dengan bahasa C dibandingkan dengan bahasa lain.

II. DASAR TEORI

A. *Himpunan (Set)*

Himpunan adalah sekumpulan objek berbeda yang tak memperhatikan keterurutan objek-objek tersebut. Objek-

objek dalam himpunan sering disebut sebagai anggota atau elemen. Notasi a ∈ A menyatakan bahwa a merupakan elemen dari himpunan A. Sedangkan notasi a ∉ A menyatakan sebaliknya yaitu a bukan elemen dari himpunan A.

Ada beberapa cara mendefinisikan suatu himpunan, yaitu :

a) Dengan cara medeskripsikan karakteristik himpunan tersebut dengan kata-kata.
   Contohnya :
   - A adalah himpunan bilangan bulat positif ganjil yang kurang dari 20.
   - B adalah himpunan nama-nama hari dalam satu pekan.
   - C adalah himpunan semua solusi dari persamaan $5x^4 + x^3 + 3x^2 - 2x + 1 = 0$.

b) Dengan cara menyatakan karakteristik-karakteristik yang harus dipunyai oleh semua elemen-elemennya.
   Contohnya :
   - A = { a | 0 < a < 20 dan a adalah bilangan bulat ganjil }
   - C = { x | $5x^4 + x^3 + 3x^2 - 2x + 1 = 0$ }

c) Dengan cara mendaftarkan semua anggota himpunannya.
   Contohnya :
   - A = { 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 }
   - B = { Senin, Selasa, Rabu, Kamis, Jumat, Sabtu, Minggu }

Jumlah elemen suatu himpunan biasanya disebut sebagai **kardinalitas**, yang dinotasikan dengan | A | atau n(A). Kardinalitas suatu himpunan bisa saja sama dengan 0, yang artinya himpunan ini tidak memiliki elemen satupun. Himpunan ini biasanya disebut sebagai himpunan kosong yang disimbolkan dengan ∅ atau {}.

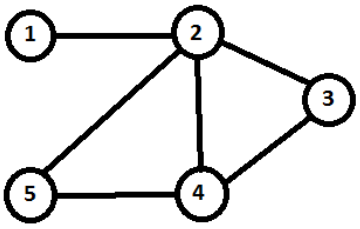
Himpunan A dikatakan himpunan bagian dari himpunan B jika dan hanya jika untuk setiap elemen dari A juga merupakan elemen dari himpunan B. Dua buah himpunan dikatakan sama jika dan hanya jika setiap elemen dari himpunan yang satu merupakan elemen himpunan yang lain dan juga sebaliknya. Dua buah himpunan dikatakan saling lepas jika dan hanya jika keduanya tidak memiliki elemen yang sama.

Dalam makalah ini operasi-operasi himpunan yang akan dipakai hanyalah operasi dasar yang berkenaan dengan definisi dari himpunan, yaitu seperti penambahkan elemen, penghapusan elemen, dan pencarian elemen. Sedangkan operasi-operasi lain seperti gabungan, irisan, komplemen, selisih, dan beda setangkup, tidak akan dibuat implementasinya kecuali untuk beberapa kasus yang memang dianggap perlu.

B. Graf

Graf adalah suatu struktur diskrit yang terdiri atas simpul-simpul dan sisi-sisi yang menghubungkan sepasang simpul. Sebuah graf G = (V, E) terdiri atas V yaitu suatu himpunan tak kosong simpul-simpul, dan E yaitu suatu himpunan sisi-sisi yang menghubungkan simpul-simpul yang ada pada V.

Contoh graf :



Gambar 1 : Contoh graf

Graf pada gambar 1 merepresentasikan suatu graf G = (V,E) sebagai berikut :

V = { 1, 2, 3, 4, 5 }

E = {(1,2), (2,3),(2,4),(2,5),(3,4),(4,5)}

C. Pohon

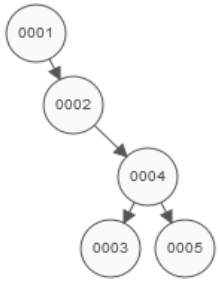
Pohon adalah suatu graf terhubung yang tak berarah dan tidak mengandung suatu sirkuit. Suatu pohon memiliki beberapa sifat, sebagai berikut :

- Selalu terdapat tepat satu lintasan yang menghubungkan sepasang simpul.
- Pohon yang memiliki n buah simpul yang terhubung akan selalu memiliki n-1 buah sisi.
- Penambahan satu sisi akan menghasilkan satu buah sirkuit.
- Penghapusan satu sisi akan membuat pohon terbagi menjadi dua buah komponen.

Pohon berakar adalah pohon yang menjadikan salah satu simpulnya sebagai sebuah akar, kemudian semua sisi-sisinya diberi arah yang selalu menjauhi akar tersebut. Namun karena arahnya yang selalu menjauhi akar maka dalam penggunaannya arah-arah tersebut bisa diabaikan. Setiap pohon bisa diubah menjadi pohon berakar dengan hanya memilih salah satu simpul sebagai akarnya. Yang dimaksud pohon yang akan dipakai sebagai struktur himpunan dalam makalah ini sebenarnya adalah sebuah pohon berakar, namun untuk mempermudah kita sebut saja sebagai pohon.

Dalam pohon berakar jika ada sisi yang menghubungkan simpul a dan b dengan arah dari a ke b, maka dapat dikatakan bahwa a adalah orang tua dari b dan b adalah anak dari a. Pohon biner adalah pohon berakar yang setiap simpulnya paling banyak 2 anak yaitu anak kiri dan anak kanan.

Contoh pohon berakar :



Gambar 2 : Contoh pohon berakar

III. IMPLEMENTASI HIMPUNAN DALAM POHON

Operasi yang akan sangat sering dipakai dalam himpunan adalah operasi pencarian elemen. Hal ini dikarenakan proses pengecekan apakah suatu objek elemen dari suatu himpunan merupakan hal yang sangat vital sehingga akan sangat sering dilakukan, dan pengecekan tersebut dilakukan dengan melakukan pencarian objek tersebut diantara elemen-elemen yang ada dalam suatu himpunan. Oleh karena itu, dengan penggunaan struktur pohon sebagai representasi himpunan diharapkan proses pencarian akan lebih optimal.

Di dalam pohon, elemen himpunan disimbolkan sebagai suatu simpul. Sedangkan himpunan itu sendiri disimbolkan sebagai satu pohon yang terhubung. Sehingga jika terdapat dua buah simpul yang terhubung maka kedua simpul tersebut berada dalam himpunan yang sama.

Ada beberapa alternatif yang bisa digunakan untuk merepresentasikan suatu himpunan menjadi sebuah pohon yang bergantung pada kasus-kasus yang bisa ditanganinya, yaitu :

A. Alternatif I : Union-Find Disjoint Sets

Untuk alternatif I ini hanya bisa dipakai jika himpunan-himpunan yang dipakai adalah himpunan yang saling lepas. Sehingga setiap elemen hanya boleh menjadi elemen dari satu himpunan saja. Sturktur data ini sering disebut sebagai *Union-Find Disjoint Sets*, karena struktur data ini hanya memodelkan sekumpulan himpunan (*Sets*) yang saling lepas (*Disjont*) dan operasi yang paling diunggulkan adalah operasi pencarian (*Find*) suatu elemen terletak pada himpunan yang mana dan operasi penggabungan (*Union*) yang menggabungkan dua buah himpunan yang saling lepas menjadi satu buah himpunan gabungan yang lebih besar.

Ide dari struktur data ini adalah dengan menyimpan siapa orang tua dari setiap simpul. Khusus untuk simpul yang menjadi akar, orang tuanya adalah dirinya sendiri. Misalkan kita mempunyai Nmax buah bilangan dari 0 sampai Nmax-1 sebagai objek-objek himpunan. Kemudian untuk menyimpan informasi orang tuanya bisa kita gunakan array yang sebanyak Nmax buah, misalkan array tersebut adalah `int Ortu[Nmax];` sehingga pada awalnya Ortu[x] = x.

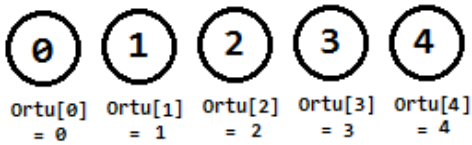
Kemudian untuk melakukan penggabungan himpunan kita hanya perlu mengubah orang tua dari salah satu akar menjadi akar yang lain, sehingga kedua pohon yang awalnya saling lepas kemudian menjadi bersatu. Sedangkan untuk mengecek dua buah objek terletak dalam himpunan yang sama maka cukup mengecek apakah akar pohon dari simpul yang satu sama dengan akar pohon dari simpul yang lainnya. Dalam mencari akar dari pohon dimana simpul tersebut berada juga sekaligus mengganti orang tua simpul tersebut menjadi akar dari pohon tersebut. Tujuannya adalah agar dalam pencarian selanjutnya jarak antara simpul tersebut dengan akar tidak terlalu jauh lagi.

Berikut ini adalah potongan kode dalam bahasa C :

```
/* untuk menyimpan orang tua dari setiap node
gunakan array Ortu[] */
int Ortu[Nmax], i;

/* melakukan inisialisasi himpunan */
void inisialisasi() {
   for(i = 0; i<Nmax; i++)
       Ortu[i] = i;
}

/* Mencari akar dari pohon simpul i */
int cariAkar(int i) {
   if(Ortu[i] == i) /* jika i adalah akar */
      return i;
   Ortu[i] = cariAkar(Ortu[i]); /* mengganti
nilai orang tua menjadi akarnya */
  return Ortu[i];
}

/* Mengecek 2 simpul dalam pohon yang sama */
bool pohonSama(int i, int j) {
   return cariAkar(i) == cariAkar(j);
   /* benar jika akar kedua simpul sama */
}

/* menggabungkan 2 buah himpunan */
void gabung(int i, int j) {
   Ortu[cariAkar(i)] = cariAkar(j);
   /* mengubah orang tua akar simpul i menjadi
akar dari simpul j */
}
```

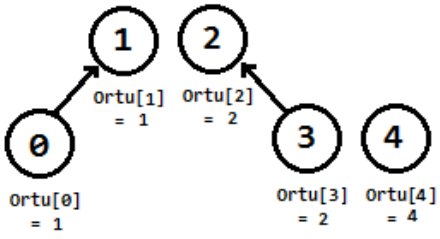
Berikut ini adalah ilustrasi bagaimana kode tersebut bekerja dengan misalkan Nmax = 5 :

```
inisialisasi(); /* Semua Ortu[i] = i */
```



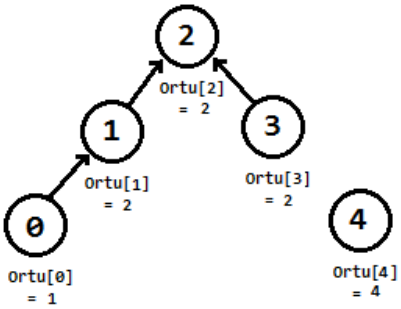
Gambar 3 : Pada awalnya setiap simpul menjadi himpunan tersendiri

```
gabung(0, 1); /* Ortu[0] = 1 */
gabung(3, 2); /* Ortu[3] = 2 */
```



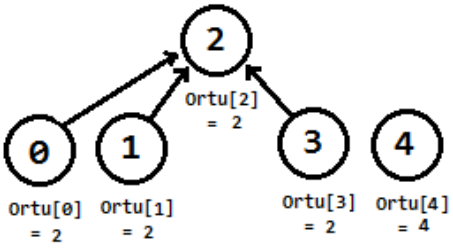
Gambar 4 : Menyisakan 3 buah himpunan yaitu {0,1}, {2,3} dan {4}.

```
gabung(1, 3); /* Ortu[1]=cariAkar(3)=2 */
```



Gambar 5 : Menyisakan 2 buah himpunan yaitu {0,1,2,3} dan {4}.

```
pohonSama(0, 3); // bernilai benar(true)
pohonSama(1, 4); // bernilai salah(false)
```

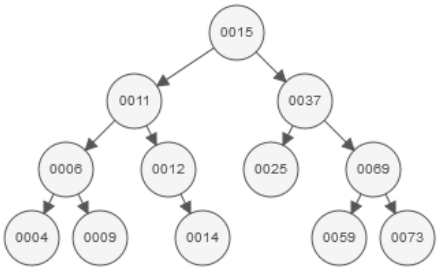


Gambar 6 : Ortu[0] mengalami perubahan disebabkan oleh pemanggilan cariAkar(0) yang juga meng*update* Ortu[0] menjadi akarnya.

2) Alternatif II : Pohon Pencarian Biner Seimbang (Balanced Binary Search Tree)

Meskipun pada dasarnya himpunan tidak memperhatikan keterurutan dari elemen-elemennya, tetapi justru kita bisa memanfaatkan sifat keturutan elemen-elemen tersebut agar proses pencarian elemen menjadi lebih optimal. Itulah yang menjadi ide dari struktur data ini, yaitu struktur data yang mampu melakukan pencarian yang optimal dengan memanfaatkan sifat keterurutan elemen-elemennya. Namun akibatnya struktur data ini hanya bisa dipakai untuk elemen-elemen yang bisa diurutkan. Struktur data yang memenuhi ini adalah struktur data Pohon Pencarian Biner Seimbang (Balanced Binary Search Tree).

Pohon biner ini memiliki aturan bahwa setiap simpul dalam pohon haruslah berlaku anak kanan selalu memiliki nilai yang lebih besar (atau sama) dengan nilai pada simpul itu sendiri, dan nilai dari anak kiri selalu lebih kecil dibandingkan nilai simpul tersebut. Operasi pencarian elemen di struktur data ini dilakukan dengan membandingkan nilai yang dicari dengan simpul-simpul pada pohon tersebut dimulai dari akarnya. Jika nilai yang dicari lebih besar dari nilai pada suatu simpul maka pencarian akan dilanjutkan ke anak kanan simpul tersebut, jika nilainya lebih kecil maka pencarian akan dilanjutkan ke anak kiri dari simpul tersebut. Pencarian akan berhenti sampai ada simpul yang nilainya sama dengan nilai yang dicari atau karena simpul yang terakhir dikunjungi tidak memiliki anak yang akan dijadikan pencarian selanjutnya yang menandakan bahwa nilai yang dicari tidak ditemukan. Karena pohon biner ini seimbang maka kompleksitas algoritma pencarian ini adalah O(log n) dengan n adalah kardinal dari himpunan tersebut.



Gambar 7 : Contoh sebuah pohon pencarian biner seimbang

Operasi penambahan elemen dilakukan dengan melakukan pencarian elemen terlebih dahulu, jika elemen yang dicari sudah ada maka penambahan elemen dibatalkan, sedangkan jika tidak maka akan ditambahkan satu simpul sebagai anak dari simpul yang terakhir dikunjungi dalam pencarian sedemikian sehingga keterurutan simpul-simpul tersebut tetap berlaku. Kemudian pohon tersebut di ubah lagi sedemikian sehingga pohon tersebut menjadi seimbang lagi. Dan operasi penghapusan elemen juga dilakukan dengan hal yang sama, yaitu mencari elemen yang akan dihapus terlebih dahulu. Kemudian jika simpul yang dicari diketemukan maka ada beberapa kemungkinan, jika simpul tersebut tidak mempunyai anak maka simpul tersebut langsung kita hapus. Jika simpul yang akan dihapus hanya memiliki satu anak maka posisi simpul tersebut digantikan oleh posisi anaknya dan kemudian simpul tersebut dihapus. Dan jika simpul tersebut memunyai dua anak maka simpul tersebut tidak dihapus tetapi nilainya diganti dengan nilai minimum yang ada pada upapohon anak kanannya dan yang dihapus adalah daun terkiri pada upapohon anak kanannya. Dengan cara demikian keturutan elemen-elemen tersebut bisa terjaga. Karena dalam operasi penambahan dan penghapusan elemen ini didominasi oleh operasi pencarian maka bisa dibilang bahwa kompleksitas penambahan dan penghapusan elemen ini juga O(log n).

Hal yang menjadi kesulitan dari struktur data ini adalam bagaimana caranya agar pohon tersebut tetap seimbang agar operasi pencarian, penambahan, dan penghapusan elemen tetap O(log n). Ada beberapa pohon yang cukup terkenal yang sudah mampu menyeimbangkan pohon itu dengan menggunakan beberapa rotasi diantara simpul-simpul tersebut yaitu pohon Adelson-Velski Landis (AVL) dan pohon Red-Black (RB). Pohon AVL ditemukan oleh Geogii Adelson-Velski (lahir pada 1922) dan Evgenii Mikhailovich Landis (1921-1997) pada tahun 1962 dan pohon RB ditemukan oleh Rudolf Bayer (lahir pada 1939) yang merupakan seorang profesor informatika di *Technical University of Munich*[2]. Pohon RB ini dipakai dalam merealisasikan struktur data *set* dan *map* pada STL C++ serta *TreeMap* dan *TreeSet* dalam STL Java.

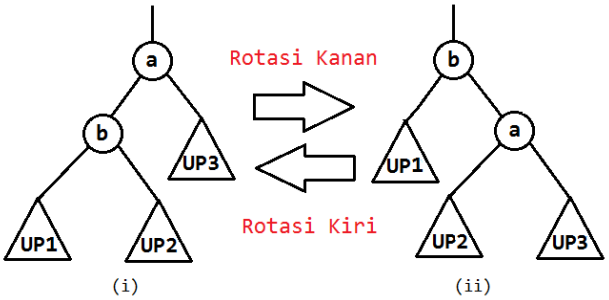
Namun yang akan kita bahas dalam makalah ini adalah pohon AVL sebab pohon AVL relatif lebih seimbang dibandingkan pohon RB sehingga penggunaan pohon AVL akan relatif lebih cepat dibandingkan pohon RB[3].

Pohon AVL ini dikatakan seimbang jika selisih ketinggian dari upapohon kiri dan kanan tidak lebih dari satu. Untuk operasi pencarian, penambahan, dan penghapusan elemen pada pohon AVL ini adalah sama seperti penjelasan sebelumnya, namun yang akan dijelaskan setelah ini adalah bagaimana caranya pohon AVL ini melakukan penyeimbangan kembali simpul-simpulnya setelah penambahan elemen ataupun penghapusan elemen.

Berikut ini adalah bagaimana mendeklarasikan simpul dalam pohon dan sekaligus fungsi konstruktotnya dalam bahasa C :

```
struct simpul
{
    int nilai;
    struct node *kiri;
    struct node *kanan;
    int ketinggian;
};
/* Fungsi konstruktor untuk membuat suatu
simpul baru */
struct simpul* buatSimpul(int nilai)
{
    struct simpul* simpulBaru = (struct
simpul* malloc(sizeof(struct simpul));
    simpulBaru->nilai    = nilai;
    simpulBaru->kiri     = NULL;
    simpulBaru->kanan    = NULL;
    simpulBaru->ketinggian = 1;
    return(simpulBaru);
}
```

Setalah melakukan operasi penambahan elemen untuk mengatur agar pohon AVL kembali menjadi seimbang maka diperlukan 2 buah operasi dasar yang akan kita pakai yaitu rotasi kiri dan rotasi kanan. Misalkan a, b adalah adalah simpul dalam pohon, dan UP1, UP2, UP3 adalah 3 buah upapohon. Maka yang dimaksud dengan rotasi kanan dan rotasi kiri adalah sebagai berikut :



Gambar 8 : Rotasi Kanan & Rotasi Kiri

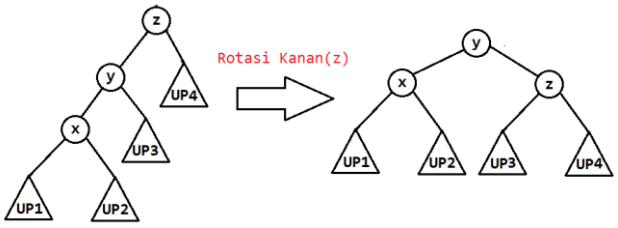
Nilai-nilai dalam kedua upapohon tersebut harus terurut sesuai aturan yang sudah dijelaskan sebelumnya yaitu nilai_akar(UP1) < nilai(b) < nilai_akar(UP2) < nilai(a) < nilai(UP3).

Berikut ini adalah potongan kode untuk implementasi prosedur rotasi kanan dan rotasi kiri dalam bahasa C :

```
struct simpul *rotasiKanan(struct simpul
*y)
{
    struct simpul *x = y->kiri;
    struct simpul *T2 = x->kanan;
  // Rotasi
  x->kanan = y;
  y->kiri = T2;
  // memperbarui ketinggian
  y->ketinggian = max(ketinggian(y->kiri),
ketinggian(y->kanan))+1;
  x->ketinggian = max(ketinggian(x->kiri),
ketinggian(x->kanan))+1;

  return x; // mengembalikan akar baru
}

Struct simpul *rotasiKiri (struct simpul *x)
{
    struct simpul *y = x->kanan;
    struct simpul *T2 = y->kiri;

    // rotasi
    y->kiri = x;
    x->kanan = T2;
    //  memperbarui ketinggian
    x->ketinggian = max(ketinggian(x->kiri),
ketinggian(x->kanan))+1;
    y->ketinggian = max(ketinggian(y->kiri),
ketinggian(y->kanan))+1;

    return y; // mengembalikan akar baru
}
```

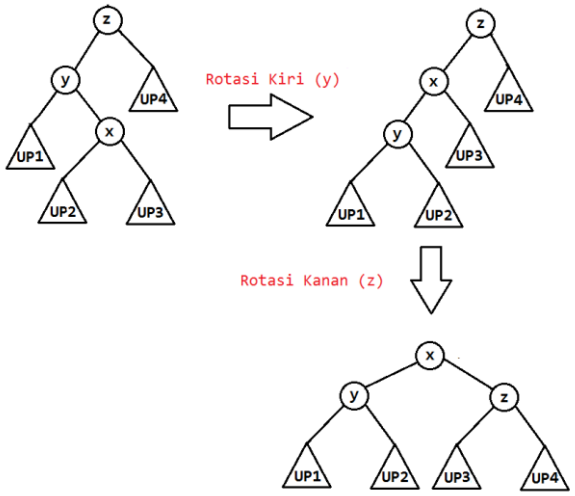
Setelah melakukan penambahan satu elemen maka ada kemungkinan bahwa pohon AVL tidak lagi seimbang. Sehingga perlu dilakukan pencarian simpul pertama yang tidak seimbang dimulai dari simpul yang baru ditambahkan sampai akar. Jika tidak ditemukan berarti pohon tersebut sudah seimbang. Namun jika ditemukan misalnya simpul tersebut adalah simpul z maka perlu dilakukan penyeimbangan kembali pada simpul tersebut. Misalkan y adalah anak dari simpul z sedemikian sehingga ketinggian y lebih dari ketinggian anak dari z yang lain, dan x adalah anak dari simpul y sedemikian sehingga ketinggian x lebih dari ketinggian anak dari y yang lain. Sehingga akan ada 4 kemungkinan yang terjadi :

a) Kasus ke-1 : jika y adalah anak kiri z dan x adalah anak. kiri y.



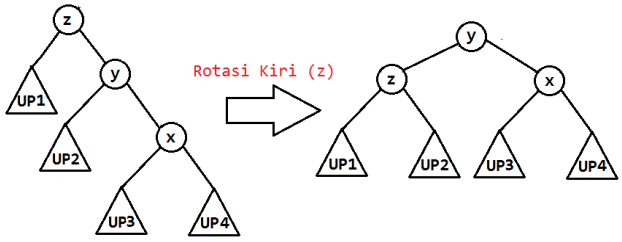
Gambar 9 : Penanganan pada kasus 1 dengan rotasi kanan pada simpul z.

b) Kasus ke-2 : jika y adalah anak kiri z dan x adalah anak kanan y.



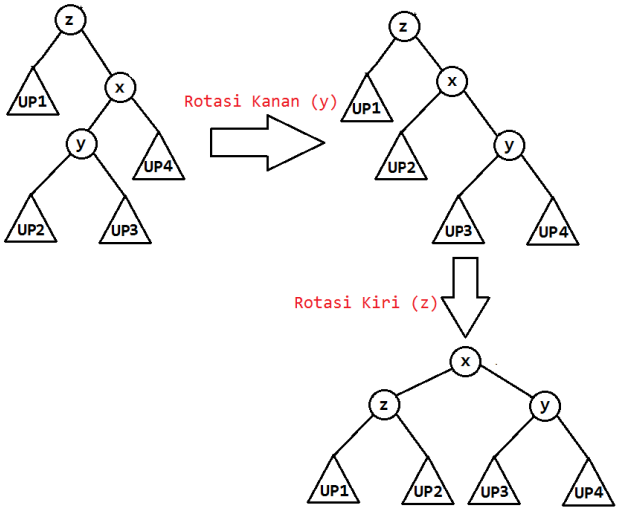
Gamabar 10 : Penanganan untuk kasus ke-2 dengan cara rotasi kiri (y) kemudian rotasi kanan(z).

c) Kasus ke-3 : jika y adalah anak kanan z dan x anak adalah kanan y.



Gambar 11 : Penanganan untuk kasus ke-3 dengan cara rotasi kiri (z).

d) Kasus ke-4 : jika y adalah anak kanan z dan x adalah anak kiri y.



Gambar 12 : Penanganan untuk kasus ke-4 dengan cara rotasi kanan (y) kemudian rotasi kiri (z).

Kemudian untuk melakukan operasi penghapusan, caranya sama seperti yang telah dijelaskan diawal, kemudian karena penghapusan elemen bisa mengakibatkan pohon AVL tidak lagi seimbang maka perlu dilakukan pengecekan dan penanganan yang sama seperti pada operasi penambahan elemen yang sudah dijelaskan sebelumnya.

V. KESIMPULAN

Struktur data pohon ternyata juga bisa memodelkan suatu himpunan. Ada beberapa alternatif yang bisa digunakan untuk memodelkan himpunan menggunakan pohon seperti Union-Find Disjont Sets dan Pohon Pencarian Biner Seimbang (Balanced Binary Search Tree). Dan struktur data Pohon Pencarian Biner Seimbang yang cukup terkenal adalah pohon Adelson-Velski Landis (AVL) dan pohon Red-Black (RB).

VII. UCAPAN TERIMAKASIH

Saya mengucapkan terima kasih kepada Bu Harlili dan Pak Rinaldi Munir atas bimbingannya dalam kuliah IF2120 Matematika Diskrit selama satu semester ini. Tak lupa pula saya mengucapkan terimakasih kepada teman-teman seperjuangan yang telah memberikan bantuan dan dorongannya selama ini.

REFERENSI

[1] K. H. Rosen, Discrete Mathematics and Its Applications 7th. New York: McGraw-Hill, 2012
[2] Halim, Steven, Competitive Programming 3, Steven & Felix Halim, 2013.
[3] Pfaff, Ben (June 2004). "Performance Analysis of BSTs in System Software". Stanford University.
[4] http://ww.geeksforgeeks.org/avl-tree-set-1-insertion/ diakses pada 11 Desember 2014 pukul 07.00 WIB.
[5] http://www.geeksforgeeks.org/avl-tree-set-2-deletion/ diakses pada 11 Desember 2014 pukul 07.00 WIB.
[6] https://www.cs.usfca.edu/~galles/visualization/AVLtree.html diakses pada 11 Desember 2014 pukul 08.00 WIB.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 11 Desember 2014

Wiwit Rifa'i - 13513073