Pembahasan Kejuaraan Sofita

Deskripsi Singkat

Tim Esdea mengadakan ajang kejuaraan dimana kejuaraan dilakukan dengan format DUO. Sebelum mengadakan kejuaraan, tim Esdea menerapkan pendaftaran peserta yang memiliki power level. Sehingga terdapat dua format query pada permasalahan ini yaitu:

- **MASUK** $P\ S$: Peserta bernama P yang memiliki *power level* sebesar S melakukan pendaftaran kepada Tim Esdea.
- **DUO** $K\ B$**:** Tim Esdea akan membentuk *duo* dengan peserta di antara *power level* K dan B **(inklusif).** *Duo* yang akan dibentuk perlu memiliki peserta yang berbeda dengan **selisih *power level* terbesar**.

MASUK

Query MASUK ini mengeluarkan output banyak peserta lain yang memiliki power level lebih rendah dari power level peserta yang baru masuk. Untuk mencapai tujuan ini, salah satu cara yang efektif adalah membuat atribut `count` pada tiap Node di AVL Tree. `count` pada masing-masing Node menyimpan **banyak peserta dari setiap child Node tersebut**.

Selain itu, perhatikan bahwa dua atau lebih peserta mungkin saja memiliki power level yang sama. Untuk mengakomodasi hal tersebut dan untuk memudahkan query DUO nantinya, stack dapat dimanfaatkan untuk menyimpan duplikasi power level yang ada. Stack pada setiap node juga akan dimanfaatkan pada saat menghitung keluaran untuk query MASUK.

Untuk dapat menerapkan `count` pada tiap Node, maka hal yang dilakukan adalah sebagai berikut

1. Insert

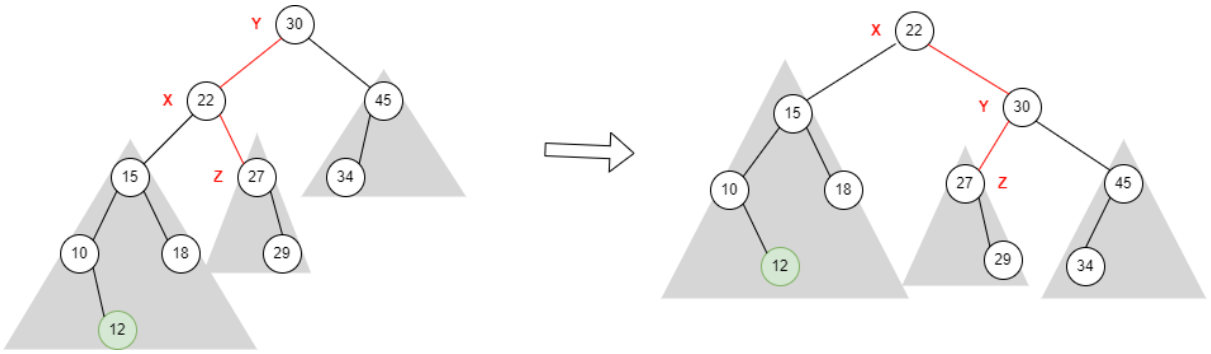
```
Node insertNode(Node node, int power, Peserta peserta) {
   if (node == null) {
       Node newNode = new Node(power);
       newNode.stackPeserta.push(peserta); // Penggunaan Stack untuk query DUO
       return newNode;
   }

   if (power == node.power) {
       // Hitung count

       node.stackPeserta.push(peserta);
```

```
        return node;
    }

    if (power < node.power) {
        node.ownChild++;
        node.left = insert(node.left, power, peserta);
    } else {
        // Hitung count

        node.ownChild++;
        node.right = insert(node.right, power, peserta);
    }

    // Lakukan balancing jika perlu (Right/Left Rotate)

    return node;
}
```

Untuk menghitung count dari child maka setiap kali kita traverse node maka apabila `power < node.power` ataupun `power > node.power` maka count node yang dikunjungi akan bertambah 1 setiap insert terjadi.

Karena AVL Tree merupakan Tree yang selalu balance maka kita perlu memanipulasi perhitungan saat terjadi balancing. Balancing dilakukan dengan 2 cara yaitu right rotate dan left rotate. Maka dari itu kita perlu memanipulasi rumus perhitungan agar nilai count child pada node tetap benar nilainya.

2. Right Rotate



Sebelum Right Rotate

```
X.count = Z.count + Z.stack.size() + X.left.count + X.left.stack.size()
Y.count = X.count + X.stack.size() + Y.right.count + Y.right.stack.size()
```

Setelah Right Rotate

Karena Z menjadi child dari Y dan Y menjadi child dari X, maka expected Y.count setelah right rotate adalah

```
Y_after.count = Y.right.count + Y.right.stack.size() + Z.count + Z.stack.size()
Y_after.count = Y.count - (X.count + X.stack.size()) + (Z.count + Z.stack.size())
```

Dengan expected Y.count maka didapatkan juga expected X.count

```
X_after.count = X.left.count + X.left.stack.size() + Y_after.count + Y_after.stack.size()
X_after.count = X.count - (Z.count + Z.stack.size()) + (Y_after.count +
                Y_after.stack.size())
```

3. Left Rotate

Sebelum Left Rotate

```
Y.count = Z.count + Z.stack.size() + Y.right.count + Y.right.stack.size()
X.count = Y.count + Y.stack.size() + X.left.count + X.left.stack.size()
```

Setelah Left Rotate

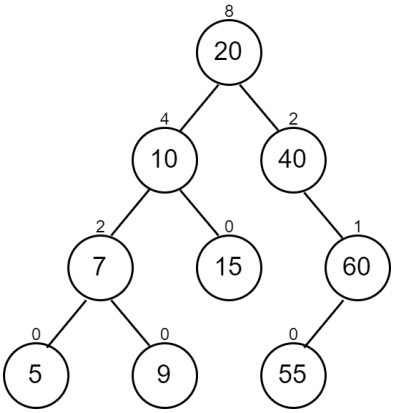
Karena Z menjadi child dari X dan X menjadi child dari Y, maka expected X.count setelah left rotate adalah

```
X_after.count = X.left.count + X.left.stack.size() + Z.count + Z.stack.size()
X_after.count = X.count - (Y.count + Y.stack.size()) + (Z.count + Z.stack.size())
```

Dengan expected X.count maka didapatkan juga expected Y.count

```
Y_after.count = Y.right.count + Y.right.stack.size() + X_after.count +
                X_after.stack.size()
Y_after.count = Y.count - (Z.count + Z.stack.size()) + (X_after.count +
                X_after.stack.size());
```

Setelah selesai dengan implementasi count pada tiap Node, hal selanjutnya yang perlu diimplementasikan adalah bagaimana cara memanfaatkan count pada tiap Node itu untuk mendapatkan jumlah peserta dengan power level yang lebih rendah dari peserta baru. Mari kita analisis contoh AVL Tree berikut (dengan asumsi masing-masing power level hanya dimiliki 1 orang).



IDE: MENGAKUMULASI BANYAK POWER LEVEL YANG LEBIH KECIL DILAKUKAN KETIKA TRAVERSING TREE SAAT MELAKUKAN INSERT

KONSEP: BILANGAN YANG LEBIH KECIL DARI SUATU NODE ADALAH SEMUA BILANGAN YANG BERADA PADA LEFT CHILDNYA!

Misal kita ingin memasukkan **13** ke AVL Tree. Maka pada tahap insertion, 13 ini akan melewati Node **20, 10, 15** hingga akhirnya berada di posisi yang semestinya. Mari kita analisis setiap traversenya.

1. **20:** 20 tidak lebih kecil dari 13, sehingga skip saja
2. **10:** 10 lebih kecil dari 13. Sesuai konsep tadi, maka Node apapun yang berada di left child dari Node 10 juga pasti lebih kecil dari 13. Jumlah peserta pada semua left child Node 10 adalah **Count pada Node 7 (left childnya) + Jumlah peserta pada Node 7 itu sendiri = 2 + 1 = 3 peserta.** Karena 10 juga lebih kecil dari 13, jangan lupa ditambahkan dengan jumlah peserta pada Node 10, yaitu 1. **Sehingga, totalnya adalah 4**
3. **15:** 15 tidak lebih kecil dari 13, sehingga skip saja

Dari contoh ini, dapat kita peroleh pola yang menarik.

- Ketika sedang traverse dan sedang berada di Node 20 lalu proses traversenya selanjutnya akan ke Node 10 (left child), tidak dilakukan apa apa.
- Ketika sedang traverse dan sedang berada di Node 10 lalu proses traversenya selanjutnya akan ke Node 15 (right child), ada kalkulasi yang dilakukan sehingga bisa mendapatkan power level yang lebih kecil.

Dari sini dapat kita tarik kesimpulan bahwa ada tindakan khusus apabila kita berada di suatu Node dan Node untuk di traverse selanjutnya adalah right child dari Node sekarang. Apabila Node untuk di traverse selanjutnya adalah left child, tidak ada tindakan apa apa. Namun perlu diingat, kalkulasi tadi melibatkan left child dari Node dan tidak semua Node memiliki left child sehingga perlu dicek ada atau tidaknya left child.

Untuk mengumpulkan akumulasi yang dikumpulkan selama traversing, kita bisa menambahkan atribut baru pada class AVL Tree bernama **currentCount**. Atribut ini dipanggil di method main setiap saat melakukan insertion dan di-reset valuenya menjadi 0 setelah mengetahui jumlahnya. Dengan demikian, kode untuk implementasi tersebut adalah sebagai berikut

```
Node insertNode(Node node, int power, Peserta peserta) {
    if (node == null) {
        Node newNode = new Node(power);
        newNode.stackPeserta.push(peserta); // Penggunaan Stack untuk query DUO
        return newNode;
    }

    if (power == node.power) {
        if (node.left != null) {
            currentCount += node.left.ownChild + node.left.stackPeserta.size();
        }
        node.stackPeserta.push(peserta);
        return node;
    }

    if (power < node.power) {
        node.ownChild++;
        node.left = insert(node.left, power, peserta);
    } else {
        if (node.left != null) {
            currentCount += node.left.ownChild + node.left.stackPeserta.size();
        }
        currentCount += node.stackPeserta.size();
        node.ownChild++;
        node.right = insert(node.right, power, peserta);
    }

    // Lakukan balancing jika perlu (Right/Left Rotate)

    return node;
}
```

DUO

Query DUO mengeluarkan output nama dari *duo* yang memiliki selisih *power level* terbesar dengan format $NAMA1$<spasi>$NAMA2$ dengan ketentuan:

- $NAMA1$ lebih kecil secara leksikografis dari $NAMA2$
- Apabila terdapat lebih dari satu kemungkinan duo yang terbentuk dengan selisih power level terbesar, maka peserta yang akan dipilih adalah peserta yang terakhir kali mendaftar ke Tim Esdea. Peserta-peserta yang telah terpilih tidak akan terpilih lagi dan dihilangkan dari daftar peserta Tim Esdea
- Apabila duo tidak dapat dibentuk, maka cetak -1<spasi>-1

Untuk memenuhi semua kebutuhan di atas kita membutuhkan **STACK** untuk menyimpan peserta yang terakhir kali mendaftar pada setiap node power. Untuk mencari nama dari peserta yang terpilih berdasarkan selisih terbesar kita dapat menggunakan lower bound dan upper bound dari input yang diberikan.

1. Lower Bound

```
Node lowerBound(Node node, int inputLow) {
   Node lowest = new Node(-1);

   // Base case
   if (node == null) {
       return lowest;
   }

   if (node.power == inputLow) {
       return node;
   }

   if (node.power < inputLow) {
       return lowerBound(node.right, inputLow);
   }

   Node lower = lowerBound(node.left, inputLow);
   return (lower.power >= inputLow) ? lower : node;
}
```

Pada Lower Bound, kita perlu mengecek apakah node yang dikunjungi lebih kecil dari `inputLow` atau tidak. Jika iya, maka node tersebut tidak memenuhi sebagai lower bound karena sudah melewati batas terendah, sehingga kita perlu mengecek right child-nya. Sedangkan jika node tidak lebih kecil dari `inputLow`, maka `node.left` akan menjadi Lower Bound terbarunya.

2. Upper Bound

```
Node upperBound(Node node, int inputHigh) {
    Node highest = new Node(Integer.MAX_VALUE);

    // Base case
    if (node == null) {
        return highest;
    }

    if (node.power == inputHigh) {
        return node;
    }

    if (node.power > inputHigh) {
        return upperBound(node.left, inputHigh);
    }

    Node upper = upperBound(node.right, inputHigh);
    return (upper.power <= inputHigh) ? upper : node;
}
```

Pada Upper Bound, kita perlu mengecek apakah node yang dikunjungi besar kecil dari `inputHigh` atau tidak. Jika iya, maka node tersebut tidak memenuhi sebagai upper bound karena sudah melewati batas tertinggi, sehingga kita perlu mengecek left child-nya. Sedangkan jika node tidak lebih besar dari `inputLow`, maka `node.right` akan menjadi Upper Bound terbarunya.

Setelah mendapatkan nilai dari `lowerNode` (hasil dari `lowerBound()`) dan nilai dari `upperNode` (hasil dari `upperBound()`) maka kita perlu melakukan beberapa pengecekan kondisional:

- Jika salah satu dari `lowerNode` atau `upperBound` tidak ditemukan, maka output-nya -1 -1
- Jika `lowerNode.power > upperNode.power`, maka output-nya -1 -1
- Jika `lowerNode == upperNode` dan `stackPeserta.size() == 1`, maka output-nya -1 -1
- Jika tidak termasuk kondisi di atas, maka akan diambil nama dari masing-masing `lowerNode` dan `upperNode`, lalu dilakukan `deleteNode()`. Kemudian, nama-nama tersebut akan diurutkan secara leksikografis untuk di-output-kan.

Untuk implementasi `deleteNode()`, yang perlu dilakukan adalah melakukan pengecekan apakah jumlah peserta pada stack node tersebut lebih dari 1 atau tidak. Jika iya, maka kita hanya perlu pop stack. Sedangkan jika tidak, maka hapus nodenya. Selain itu, karena kita menghapus peserta dari tree, maka lakukan traverse seperti saat melakukan `insertNode()`, bedanya kita **mengurangi count dengan 1** pada setiap node yang dikunjungi.