MODUL 10: TREE BAGIAN 2

1.1. Deskripsi Singkat

Meskipun Binary Search Tree sederhana dan mudah dimengerti, ia memiliki satu masalah utama: tidak seimbang. Dalam praktikum kali ini kita akan mempelajari salah satu jenis binary tree lainnya, yaitu AVL Tree. AVL Tree adalah Binary Search Tree yang memiliki perbedaan tinggi/ level maksimal 1 antara subtree kiri dan subtree kanan. AVL Tree muncul untuk menyeimbangkan Binary Search Tree. Dengan AVL Tree, waktu pencarian dan bentuk tree dapat dipersingkat dan disederhanakan.

Selain AVL Tree, terdapat pula Height Balanced n Tree, yakni Binary Search Tree yang memiliki perbedaan level antara subtree kiri dan subtree kanan maksimal adalah n. Sehingga, dengan kata lain AVL Tree adalah Height Balanced 1 Tree.

1.2. Tujuan Praktikum

1) Mahasiswa mampu memecahkan persoalan dengan menggunakan AVL Tree pada bahasa pemrograman C

1.3. Material Praktikum

Kegiatan pada modul ini memerlukan material berupa software editor dan compiler (atau IDE) untuk bahasa pemrograman C.

1.4. Kegiatan Praktikum

Kita akan membuat program untuk menyimpan data ke dalam struktur data AVL Tree. Kita akan memodifikasi program Binary Search Tree (BST) yang telah kita buat pada praktikum sebelumnya **(Praktikum9A.c).** Simpan ulang **Praktikum9A.c** menjadi file baru dengan nama **Praktikum10A.c.**

A. Menambah Simpul Baru

1. Pertama kita akan memodifikasi structure yang digunakan untuk membuat simpul-simpul pada tree. Pada AVL Tree, kita memerlukan informasi height atau tinggi pohon untuk memeriksa keseimbangan pohon. Pada `struct node`, kita tambahkan satu elemen tambahan:

```
int height;
```

Saat kita ingin membuat sebuah simpul baru, kita menggunakan fungsi `struct node *newNode(int data)`. Pada fungsi tersebut, kita tambahkan satu baris program untuk menginisialisasi nilai `height`.

```
new_node->height = 1;
```

Simpul baru pada awalnya ditambahkan sebagai daun, oleh karena itu heightnya bernilai 1.

2. Kemudian, kita modifikasi fungsi insert. Perbedaan AVL Tree dibandingkan dengan BST adalah AVL Tree selalu memastikan keseimbangan pohon setiap kali ada simpul baru yang ditambahkan atau dihapus dari pohon (*self balancing binary search tree*). Tahapan untuk menambahkan simpul baru pada AVL Tree adalah sebagai berikut:
   1) Lakukan insert seperti halnya pada BST
   2) Perbarui height untuk seluruh node, mulai dari node yang terakhir ditambahkan hingga ke root. Height node adalah height subtree kiri atau subtree kanan yang ada di bawahnya ditambahkan dengan 1.
   3) Hitung balance factor untuk seluruh node, mulai dari node yang terakhir ditambahkan hingga ke root, untuk menentukan apakah node balanced atau tidak
   4) Jika node unbalanced, lakukan rotasi. Ada 4 kasus rotasi, bergantung kepada posisi node terbawah.
   5) Hasil akhir fungsi adalah mengembalikan node yang telah balanced.

   Jika kita perhatikan, tahapan-tahapan di atas dilakukan secara berulang-ulang untuk node-node yang ada pada tree. Oleh karena itu, tahapan tersebut akan diimplementasikan secara rekursif. Jika diimplementasikan, kelima tahapan di atas dapat dituliskan seperti berikut ini.

```
struct node* insert(struct node* root, int new_data)
{

// 1. Lakukan BST insert biasa
if (root == NULL)
return(newNode(new_data));
// asumsi tidak boleh ada nilai yang sama dalam BST
if (new_data < root->data)
  root->left = insert(root->left, new_data);
else if (new_data > root->data)
  root->right = insert(root->right, new_data);

// 2. Update height dari node baru dan seluruh ancestornya
root->height = 1 + max(getHeight(root->left),
getHeight(root->right));
```

```
// 3. Hitung balance factor untuk menentukan apakah node
unbalanced (node baru dan seluruh ancestornya)
int balance = getBalanceFactor(root);

// Jika tidak balanced, return hasil rotation
// Kasus 1: Left Left
if (balance > 1 && new_data < root->left->data)
  return rightRotate(root);
// Kasus 2: Right Right
if (balance < -1 && new_data > root->right->data)
  return leftRotate(root);

// Kasus 3: Right Left
if (balance < -1 && new_data < root->right->data)
{
  root->right = rightRotate(root->right);
  return leftRotate(root);
}

// Kasus 4: Left Right
if (balance > 1 && new_data > root->left->data)
{
  root->left = leftRotate(root->left);
  return rightRotate(root);
}

// return node jika balanced
return root;
}
```

Ganti fungsi `insert` yang sudah ada di **Praktikum10A.c** dengan fungsi `insert` di atas. Perhatikan bahwa pada fungsi `insert` di atas terdapat beberapa fungsi tambahan yang dipanggil untuk membantu proses insert simpul baru ke dalam AVL Tree, antara lain:

1) `max`, fungsi untuk mencari tinggi maksimum antara node kiri dan node kanan
2) `getHeight`, fungsi untuk mencari tinggi sebuah node
3) `getBalanceFactor`, fungsi untuk menghitung balance faktor sebuah node

4) `rightRotate`, fungsi untuk rotasi kanan
5) `leftRotate`, fungsi untuk rotasi kiri

```
int max(int a, int b)
{
    if (a > b) return a;
    else return b;
}

int getHeight(struct node* N)
{
  if (N == NULL)
    return 0;
  return N->height;
}

// Hitung Balance factor untuk node N
int getBalanceFactor(struct node *N)
{
  if (N == NULL)
    return 0;
  return getHeight(N->left) - getHeight(N->right);
}

struct node* rightRotate(struct node *T)
{
  struct node *new_root = T->left;
  struct node *orphan = new_root->right;

  // Lakukan rotasi
  new_root->right = T;
  T->left = orphan;

  // Update height
  T->height = max(getHeight(T->left), getHeight(T-
>right))+1;
```

```
  new_root->height = max(getHeight(new_root->left),
getHeight(new_root->right))+1;

  // Return root baru
  return new_root;
}

struct node *leftRotate(struct node *T)
{
  struct node *new_root = T->right;
  struct node *orphan = new_root->left;

  // Lakukan rotasi
  new_root->left = T;
  T->right = orphan;

  // Update height
  T->height = max(getHeight(T->left), getHeight(T-
>right))+1;
  new_root->height = max(getHeight(new_root->left),
getHeight(new_root->right))+1;

  // Return root baru
  return new_root;
}
```

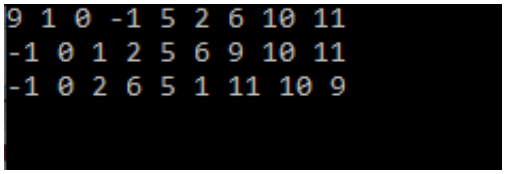
Untuk menguji apakah fungsi insert tersebut bisa dijalankan, cobalah untuk menginisiasi tree dan menambahkan beberapa simpul, lalu menampilkannya secara preorder, inorder, atau postorder. Contoh:

```
    struct Node *root = NULL;

    root = insert(root, 9);
    root = insert(root, 5);
    root = insert(root, 10);
    root = insert(root, 0);
    root = insert(root, 6);
```

```
    root = insert(root, 11);
    root = insert(root, -1);
    root = insert(root, 1);
    root = insert(root, 2);
```

Jika berhasil, maka akan muncul tampilan seperti ini di layar Anda.



B. Menghapus Sebuah Simpul

Tahapan pada penghapusan simpul AVL Tree mirip dengan tahapan penambahan simpul baru yang telah dijelaskan di atas. Penghapusan simpul pada AVL Tree dilakukan dengan cara yang sama seperti penghapusan simpul pada BST. Namun, setelahnya kita harus mengecek kembali tinggi dan balance factor untuk setiap node agar keseimbangan pohon tetap terjaga. Tahapan penghapusan simpul pada AVL Tree adalah sebagai berikut:

1) Lakukan delete seperti halnya pada BST
2) Perbarui height untuk seluruh node, mulai dari parent node yang dihapus dan seluruh ancestornya hingga ke root. Height node adalah height subtree kiri atau subtree kanan yang ada di bawahnya ditambahkan dengan 1.
3) Hitung balance factor untuk seluruh node, parent node yang dihapus dan seluruh ancestornya hingga ke root, untuk menentukan apakah node balanced atau tidak
4) Jika node unbalanced, lakukan rotasi. Ada 4 kasus rotasi, bergantung kepada posisi node terbawah.
5) Hasil akhir fungsi adalah mengembalikan node yang telah balanced.

Oleh karena itu, untuk penghapusan, kita tambahkan potongan program untuk menyeimbangkan pohon (tahap 2-5) pada fungsi `delete_node` yang sudah ada pada **Praktikum10A.c,** persisnya sebelum baris `return root;`.

```
// Jika setelah dilakukan delete, tree kosong maka return
root
    if (root == NULL)
      return root;

// 2. Update height dari node
```

```
root->height = 1 + max(getHeight(root->left),
getHeight(root->right));


// 3. Hitung balance factor untuk menentukan apakah root
unbalanced
int balance = getBalanceFactor(root);


// Jika tidak balanced, return hasil rotation

// Kasus 1: Left Left
    if (balance > 1 && getBalanceFactor(root->left) >= 0)
        return rightRotate(root);


// Kasus 2: Right Right
    if (balance < -1 && getBalanceFactor(root->right) <= 0)
        return leftRotate(root);


// Kasus 3: Right Left
    if (balance < -1 && getBalanceFactor(root->right) > 0)
    {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }


// Kasus 4: Left Right
    if (balance > 1 && getBalanceFactor(root->left) < 0)
    {
        root->left =  leftRotate(root->left);
        return rightRotate(root);
    }
```

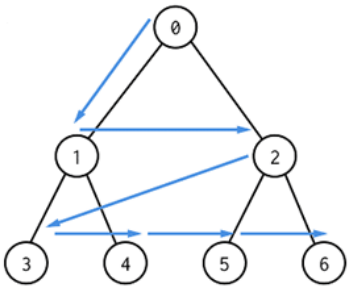
Untuk memastikan fungsi `delete_node` tersebut sudah berjalan, cobalah untuk menghapus salah satu node pada tree, lalu tampilkan kembali isi tree secara preorder, inorder, atau postorder. Contoh:

```
root=delete_node(root, 10);
```

1.5. Penugasan

Sekarang, Anda telah memahami cara membuat AVL Tree. Untuk memperdalam pemahaman Anda mengenai AVL Tree, modifikasi BST untuk menyimpan nama mahasiswa yang ada pada program **Praktikum9B.c** menjadi AVL Tree.

1. Simpan ulang **Praktikum9B.c** dengan nama **Praktikum10B_kelas_nim.c**, lalu lakukan modifikasi pada fungsi insert dan delete seperti yang kita lakukan pada kegiatan praktikum di atas.
2. Kemudian, tambahkan sebuah fungsi untuk menampilkan nama-nama mahasiswa yang ada pada tree dengan alur seperti yang diilustrasikan pada gambar di bawah ini:



Unggah file **Praktikum10B_kelas_nim.c** di Google Classroom sesuai dengan batas waktu yang telah ditetapkan **(jangan dizip/rar)**. Keterlambatan pengumpulan dikenakan sanksi pemotongan nilai sebesar 10 poin per jam.