Lab 5 : AVL Trees

Insertion:

```
struct Treenode {
        int val ; TreeNode* left ; TreeNode * right ; int ht ;
};

Void updateHeight(TreeNode* root)
{       if (root==NULL){root->ht=0; return; }.
        int ht = 1+ max(getHeight(root->left),
                                        getHeight(root->right));

        root->ht=ht;
        return ht;
}

TreeNode * insert(TreeNode* root, int x)
{       TreeNode * t=root;
        TreeNode * p=getNode(x);
        while(t!=NULL)
        {  if((t->val)<x)
            {  if (t->right==NULL)
                  { t->right=p;
                        return t;
                  }
               else  t=t->right;
            }
            elseif((t->val)>x)
            {    if(t->left==NULL)
                  { t->left=p;
                       return t;
                  }
                  else    t=t->left;
            }
        }
        ~~return t~~  UpdateHt(root);
}                      Balance(root, x);
        }
```

```
TreeNode Balance ( TreeNode* root, int x)
{
        Tree Node * t = root; Tree Node * recentImbalance;

        ~~while (t != NULL)~~
    ~~{       if (t -> data < x)~~
              ~~{     t = t -> ~~left~~ right;~~
                      ~~if ( ~~ht get~~ t ->~~
        while (t != NULL)

        {       if( t -> left -> ht - t -> right -> ht > 1 ||
                    t -> left -> ht - t -> right -> ht < -1)
                    recentImbalance = t;

                if (t -> data < x) t = t -> right;
                if (t -> data > x) t = t -> left;

                if (t -> data = = x) break;
        }

        ~~int~~ bool child;      // 0 - left
                bool grandchild  // 1 - right

        if (recentImb -> val < x)
          {       child = true;
                  ~~if (recentImb -> val -> right != NULL~~ -> v )
                  if (recentImb -> right -> val < x)
                          granchild = true;

                  else     grandchild = false;
          }
        if (recentImb -> val > x)
          {       child = false.

                  if (recentImb -> left -> val < x)
                          grandchid = false.

                  else
                          granchild = true;

          }
// Handle Cases
    Tree Node * ch ; Tree Node * gc;
  //case 1   left - left
 if ( !child && ! grandchild )
 {
        c = recentImb -> left;
       gc = recentImb -> left -> left;

       recentImb -> left = chi -> right;
       child -> right = recentImb;

 }

 //case 2    Right right
 if ( child && grandChild )
 {      C= recentImb -> right;
       gc = recentImb -> right -> right;

       recentImb -> right = ch -> left;
         ch -> left = recentImb;

 }
  // case 3    Left - right.
 if (!child && grandChild)
 {      ch = recentImb -> left;
       gc = recentImb -> right;

       ch -> right = gc -> left;
       recentImb -> left = gc -> right;

       gc -> left = ch;
       gc -> right = recentImb;
  }
  //case 4    Right-Left
  if ( child && ! grandchild)
   {        ch = recentImb -> right;  gc = recentImb -> left;
             ch -> ~~Right~~ left = gc -> right.
             recentImb -> right = gc -> left;
 }      }    gc -> left = recentImb;  gc -> right = ch;
```

```
TreedNode* ~~void~~  Delete ( int x) TreeNode * root)
{
        TreeNode* t = root; Treenode* to-be-deted;
                                TreeNode* to-be-delet-p = NULL
        while (t != NULL)
        {       ~~if (t -> left~~
                if (t -> val == x) { ~~to-be-deleted~~ to-be-deled = t; break;}
                if (t -> val > x)
                        { t = t -> left ; tobedeleted-p = t;
                if (t -> val < x)
                { to-bedeleted-p=t; t = t -> right; }
        }
//case1
        if ( to-be-deleted -> left == NULL && 
          to-be-deted -> right == NULL)
          { if (to-be-deled-p -> right == to-be-deleted)
               { to-be-deleted-p -> right = NULL;}
          else   to-be-deleted-p -> left = NULL;
                  return root;
          }

// case 2

    if( to-be-deleted -> left == NULL  ||
        to-be-deleted -> right == NULL)
    {        if ( to-be-deleted-p -> right == to-be-deleted)
             {   if (to-be-deleted -> right == NULL)
                  { to-be-deted-p -> right =
                          tobe-deteted -> left;
                  }
                 else
                   to-be-deleted-p -> right =
                            to-be deted -> right;
             }
         else   if (to-be-deleted -> left == NULL)
                 { to-be-deleted-p -> left = to-be-deleted -> right
                 else to-be-deleted-p -> left = to-be-deleted -> left
// case3 .
  i~~f( to-be-deteded->p->left == to-be-deleted )~~
  {
        TreeNode * succesor = succeusor (to-be-deletd);

          delete (succeusor);
          swap( succesor->value, to-be-deletd->val);

  UpdateHeight (root, x);
  Balance (root, x);

 }
}
```