```
class AVL {
    //-----------------------------------------------------------------
    //Type Node defines a Node for an AVL tree
     //-----------------------------------------------------------------
    struct Node {
        Node* LNode;            //Left subtree
        char* key;              //key
        Node* RNode;            //Right subtree
        int height;              //Height of the Node
        Node(char* key);        //Construct the Node
         ~Node();               //Reclaim storage for the Node
        void Print              //Print the node
        (ostream& o
        ,int depth
        );
    };
    Node* root;                 //Root of the AVL tree
    void Kill(Node* n);         //Remove all Nodes in the tree
    Node* Insert                //Insert a unique key in the tree
        (Node* n
        ,char* key
        );
    Node* Remove                //Remove a unique key from the tree
        (Node* n
        ,char* key
        );
    Node* SRR(Node* n);         //Single right rotation
    Node* SLR(Node* n);         //Single left rotation
    Node* RLR(Node* n);         //Right left rotation
    Node* LRR(Node* n);         //Left right rotation
    int Height(Node* n);        //Height of Node n
    Node* LBalance(Node* n);    //Balance on the left
    Node* RBalance(Node* n);    //Balance on the right
    Node* FindMin(Node* n);     //Find the Node having the smallest key
    int Max(int a, int b);      //Find the maximum of two integers
    void Graph                  //InOrder traversal
        (Node* n
        ,int depth
        ,ostream& o
        );
public:
    AVL();                      //Construct an empty AVL tree
    ~AVL();                     //Reclaim storage used by Nodes
    void Insert(char* key);     //Insert a key
    void Graph(ostream& o);     //Print an "ersatz" graph of the tree
    void Remove(char* key);     //Remove a key
};
```

Figure 1. class *AVL* definition**.**

```
//-----------------------------------------------------------------
//Function Insert Inserts keys into this AVL tree.
//-----------------------------------------------------------------
AVL::Node* AVL::Insert(Node* n,char* key)
{    if (!n) return new Node(key);
     if (strcmp(key,n->key)==0) return n;
     if (strcmp(key,n->key)<0) {
          n->LNode=Insert(n->LNode,key);
          if (Height(n->LNode)-Height(n->RNode)==2) {
               if (strcmp(key,n->LNode->key)<0) n=SRR(n); else n=LRR(n);
          }
     } else {
          n->RNode=Insert(n->RNode,key);
          if (Height(n->RNode)-Height(n->LNode)==2) {
               if (strcmp(key,n->RNode->key)>0) n=SLR(n); else n=RLR(n);
          }
     }
     n->height=Max(Height(n->LNode),Height(n->RNode))+1;
     return n;
}
```

Figure 2. Member function *Insert* implementation

```
//-----------------------------------------------------------------
//Function SRR performs a single right rotation on Node k2
//-----------------------------------------------------------------
AVL::Node* AVL::SRR(Node* k2)
{    Node* k1=k2->LNode;
     k2->LNode=k1->RNode;
     k1->RNode=k2;
     k2->height=Max(Height(k2->LNode),Height(k2->RNode))+1;
     k1->height=Max(Height(k1->LNode),k2->height)+1;
     return k1;
}
```

Figure 3. Member function *SRR* implementation

```
//-----------------------------------------------------------------
//Function LRR performs a left-right rotation on Node k3
//-----------------------------------------------------------------
AVL::Node* AVL::LRR(Node* k3)
{    k3->LNode=SLR(k3->LNode);
     return SRR(k3);
}
```

Figure 4. Member function *LRR* implementation