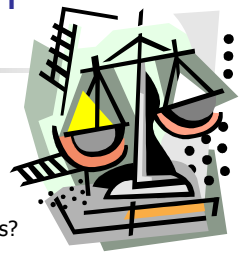
Lecture 8-3 – Data Structures

AVL Trees

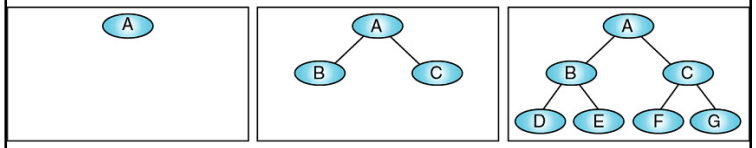
Still Searching …. with AVL Trees
Red – Black Trees



What Is Balance?
Why Is It Important Anyways?

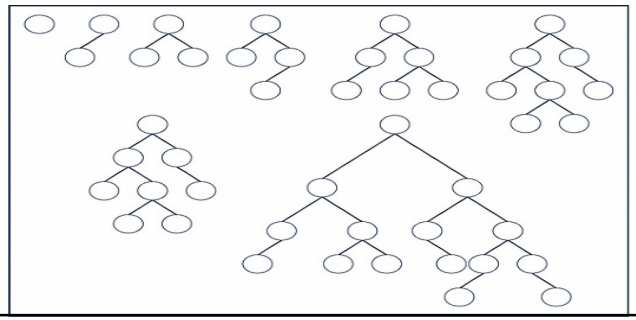
AVL (Height-balanced Trees)

- A *perfectly balanced* binary tree is a binary tree such that:
  - The *height* of the *left* and *right subtrees* of the root are *equal*
  - The *left* and *right subtrees* of the root are *perfectly balanced binary trees*

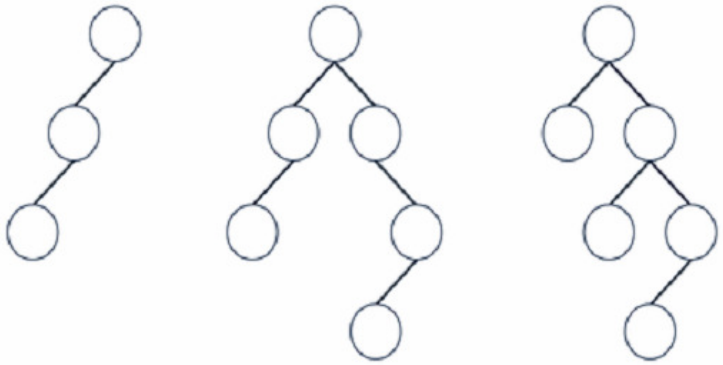


AVL (Height-balanced Trees)

- An **AVL tree** (or **height-balanced tree**) is a *binary search tree* such that:
  - The *height* of the *left* and *right subtrees* of the root *differ by at most 1*
  - The *left* and *right subtrees* of the root are *AVL trees*

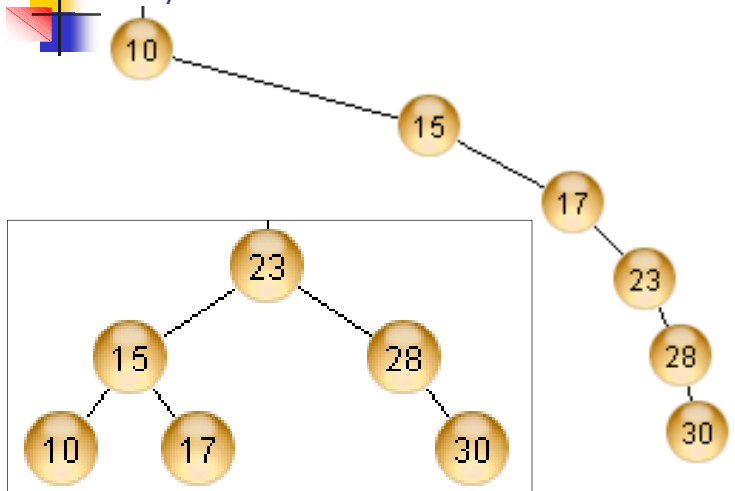


Non-AVL Trees



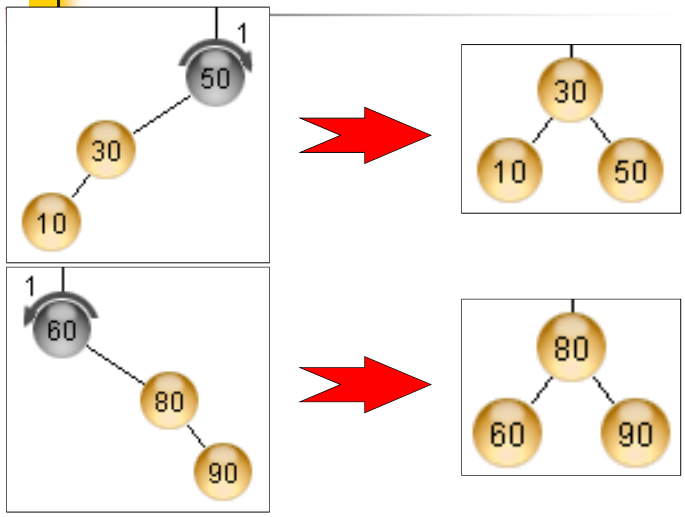
5

Binary search trees and balance



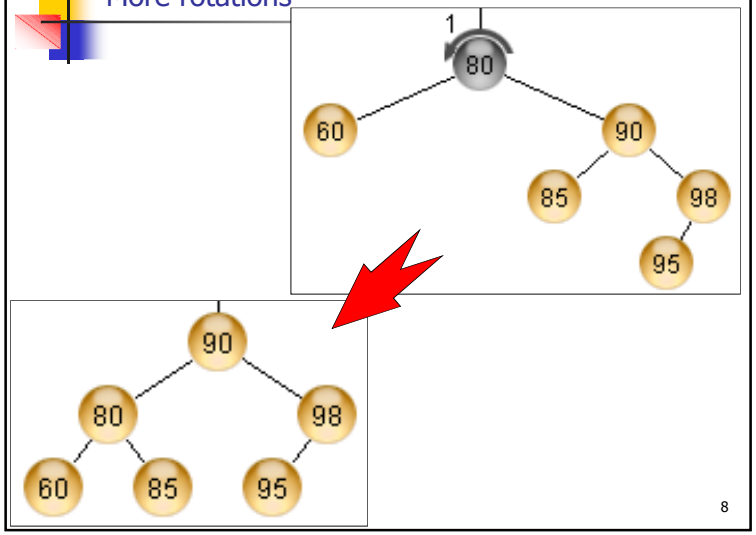
6

Rotations for balance



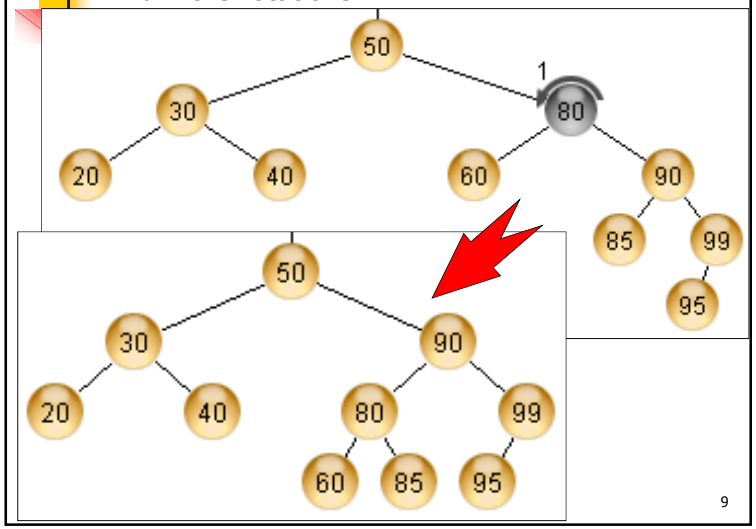
7

More rotations



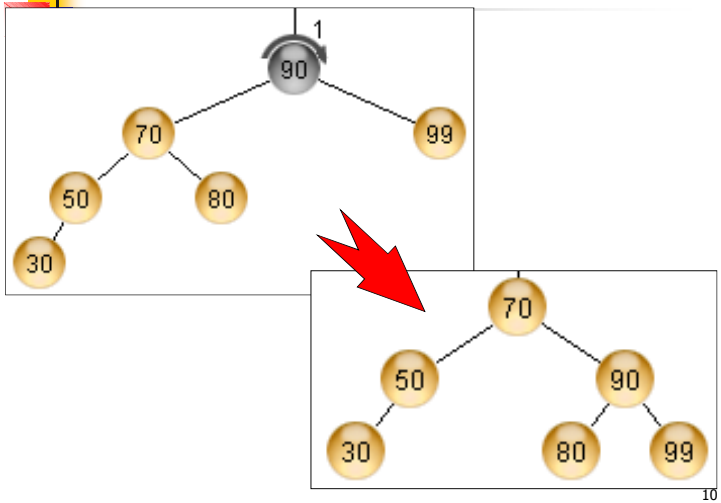
8

And more rotations



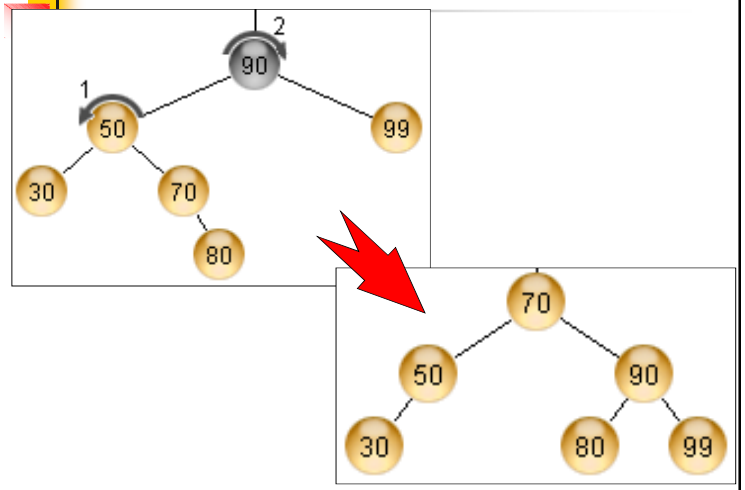
9

Yet more rotations



10

And now … how about more rotations?!



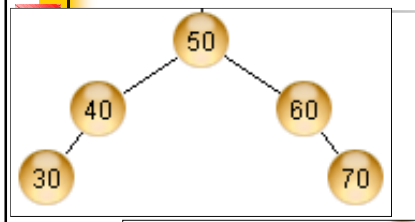
11

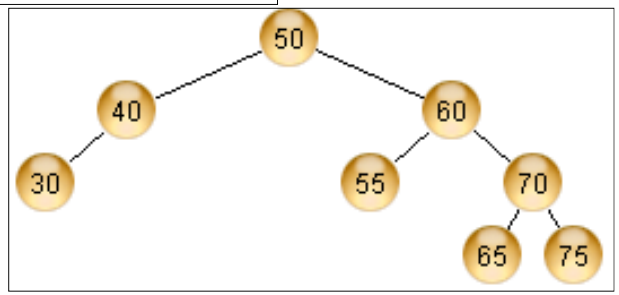
And the point of this would be?????

Now that we have rotated every which way, you were saying something about a balance!?!?!

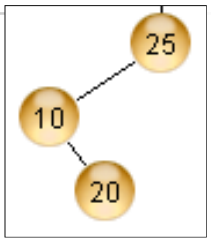
12

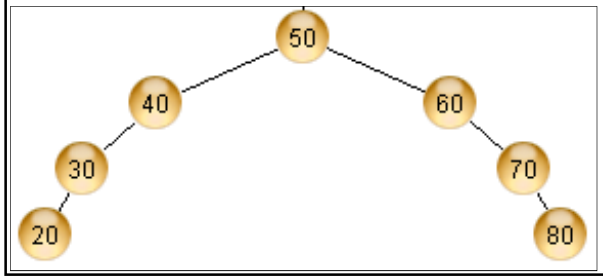
Well, on to balanced stuff – AVL trees





Are these AVL as well???





14

AVL Tree Rotations

Reconstruction procedure: **rotating** tree

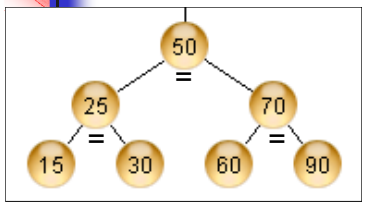
- ***left rotation*** and ***right rotation***

Suppose that the *rotation* occurs at node ***x***

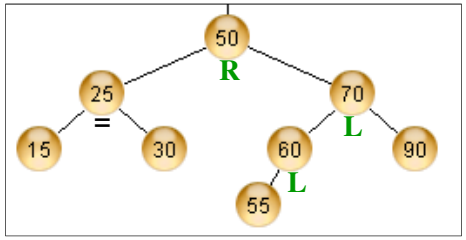
- ***Left rotation:*** certain *nodes* from the *right subtree* of *x move* to its *left subtree*; the *root* of the *right subtree* of ***x*** becomes the *new root* of the *reconstructed subtree*
- ***Right rotation at x:*** certain *nodes* from the *left subtree* of ***x*** *move* to its *right subtree*; the *root* of the *left subtree* of ***x*** becomes the *new root* of the *reconstructed subtree*

15

AVL tree cases – Case 1

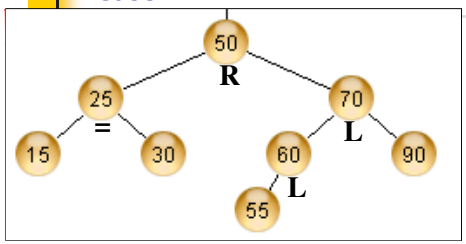


Insert 55.

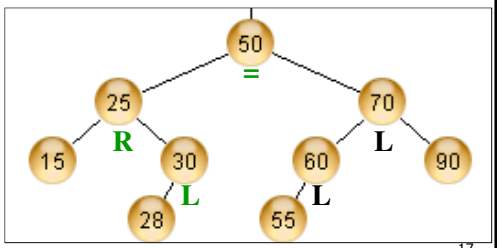


16

Case 2



Insert 28.



17

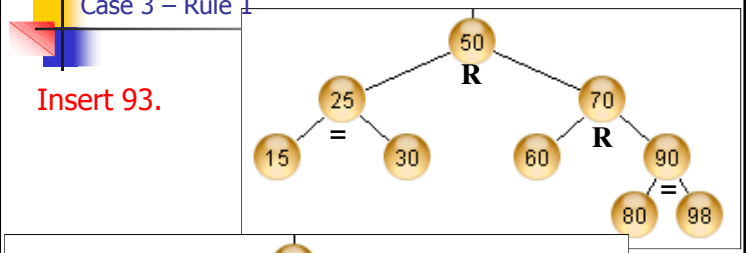
Rotation Rules

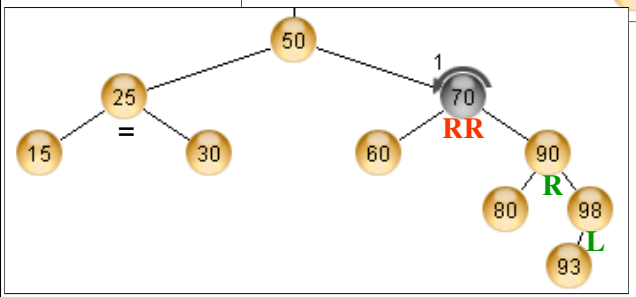
1. Ancestor balance is ***R***. Insertion is made in the *right* subtree of the *right* subtree of the *ancestor*. *Left* rotation around *ancestor*.
2. Ancestor balance is ***L***. Insertion is made in the *left* subtree of the *left* subtree of the ancestor. *Right* rotation around *ancestor*.
3. Ancestor balance is ***L***. Insertion is made in the *right* subtree of the *left* subtree of the ancestor. *Left* rotation around ancestor's *left child*. *Right* rotation around *ancestor*.
4. Ancestor balance is ***R***. Insertion is made in the *left* subtree of the *right* subtree of the ancestor. *Right* rotation around ancestor's *right child*. *Left* rotation around *ancestor*.

18

Case 3 – Rule 1

Insert 93.





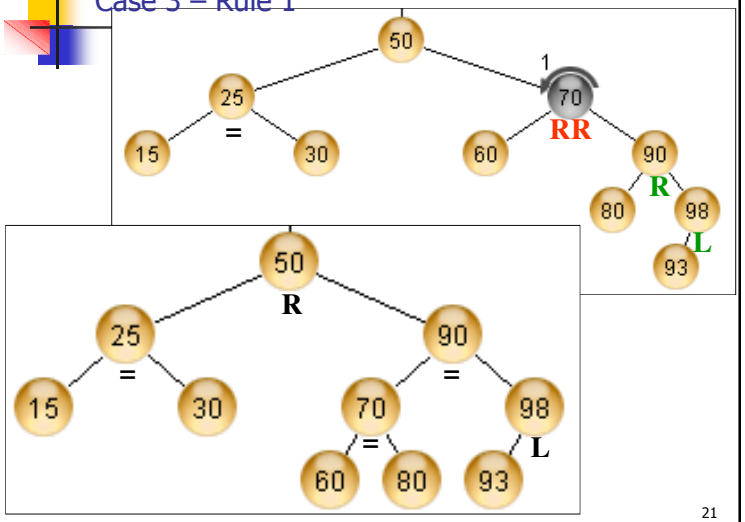
19

Rotation Rules

1. Ancestor balance is *R*. Insertion is made in the *right* subtree of the *right* subtree of the ancestor. *Left* rotation around *ancestor*.
2. Ancestor balance is *L*. Insertion is made in the *left* subtree of the *left* subtree of the ancestor. *Right* rotation around *ancestor*.
3. Ancestor balance is *L*. Insertion is made in the *right* subtree of the *left* subtree of the ancestor. *Left* rotation around ancestor's *left child*. *Right* rotation around *ancestor*.
4. Ancestor balance is *R*. Insertion is made in the *left* subtree of the *right* subtree of the ancestor. *Right* rotation around ancestor's *right child*. *Left* rotation around *ancestor*.

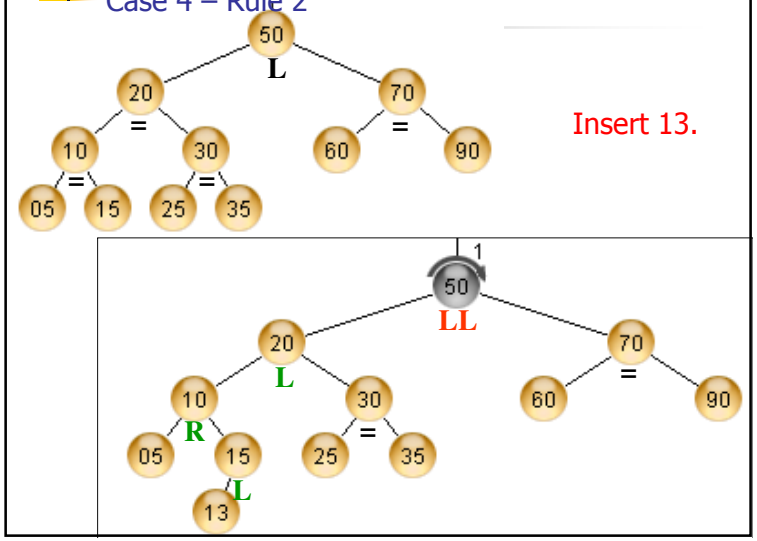
20

Case 3 – Rule 1



21

Case 4 – Rule 2



Rotation Rules

1. Ancestor balance is *R*. Insertion is made in the *right* subtree of the *right* subtree of the ancestor. *Left* rotation around *ancestor*.

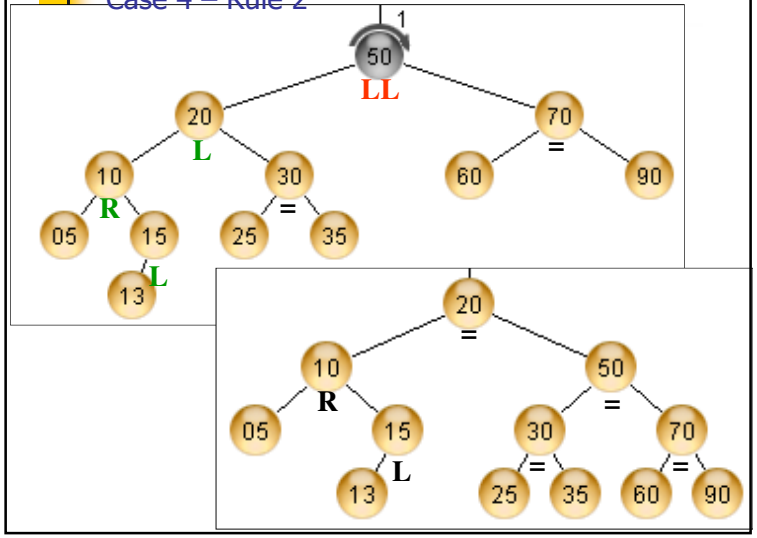
2. Ancestor balance is *L*. Insertion is made in the *left* subtree of the *left* subtree of the ancestor. *Right* rotation around *ancestor*.

3. Ancestor balance is *L*. Insertion is made in the *right* subtree of the *left* subtree of the ancestor. *Left* rotation around ancestor's *left child*. *Right* rotation around *ancestor*.

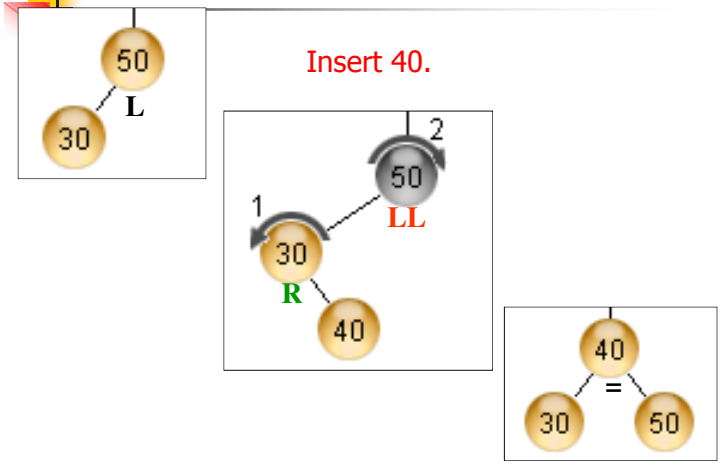
4. Ancestor balance is *R*. Insertion is made in the *left* subtree of the *right* subtree of the ancestor. *Right* rotation around ancestor's *right child*. *Left* rotation around *ancestor*.

23

Case 4 – Rule 2

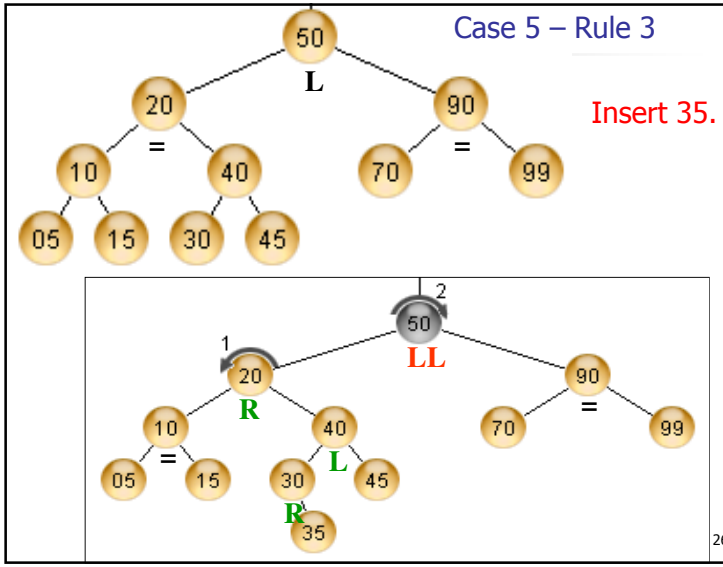


Case 5 – Rule 3



25

Case 5 – Rule 3



26

Rotation Rules

1. Ancestor balance is *R*. Insertion is made in the *right* subtree of the *right* subtree of the ancestor. *Left* rotation around *ancestor*.

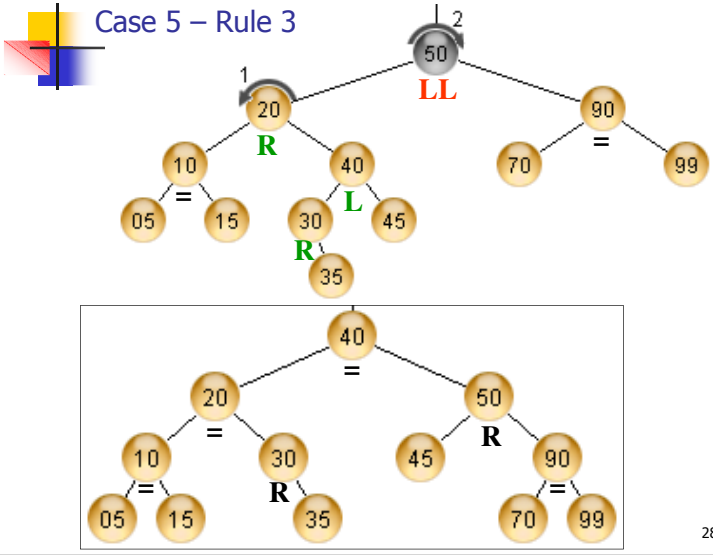
2. Ancestor balance is *L*. Insertion is made in the *left* subtree of the *left* subtree of the ancestor. *Right* rotation around *ancestor*.

3. Ancestor balance is *L*. Insertion is made in the *right* subtree of the *left* subtree of the ancestor. *Left* rotation around ancestor's *left child*. *Right* rotation around *ancestor*.

4. Ancestor balance is *R*. Insertion is made in the *left* subtree of the *right* subtree of the ancestor. *Right* rotation around ancestor's *right child*. *Left* rotation around *ancestor*.

27

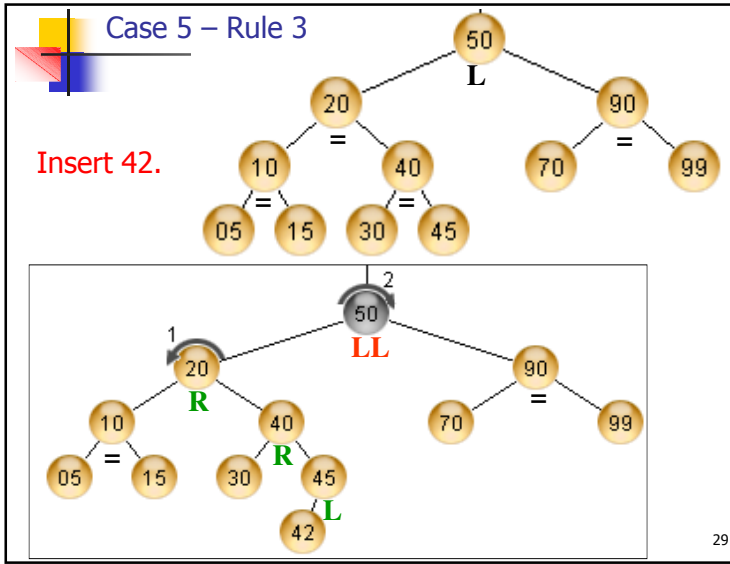
Case 5 – Rule 3



28

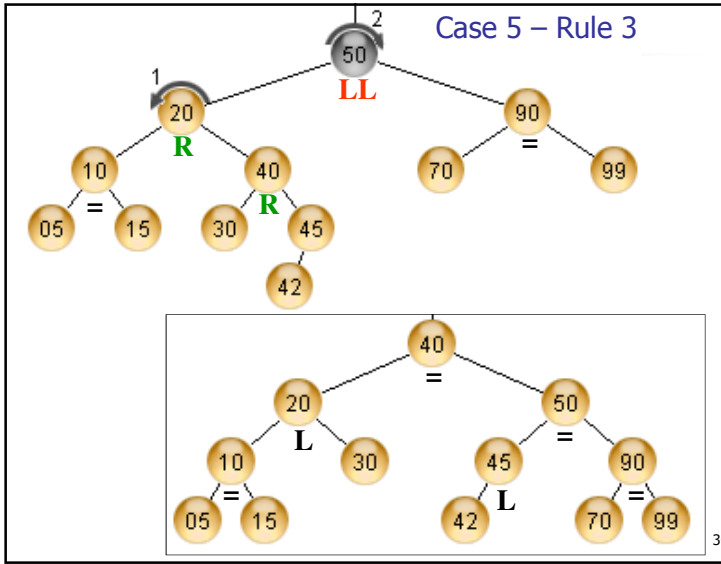
Case 5 – Rule 3

Insert 42.



29

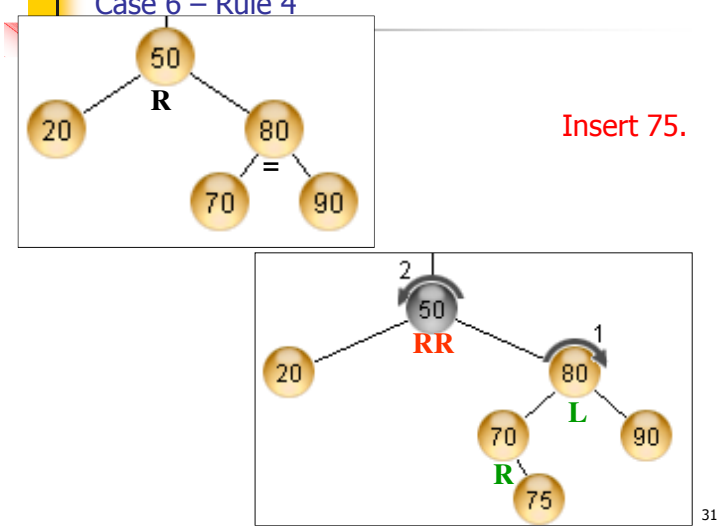
Case 5 – Rule 3



30

Case 6 – Rule 4

Insert 75.



31

Rotation Rules

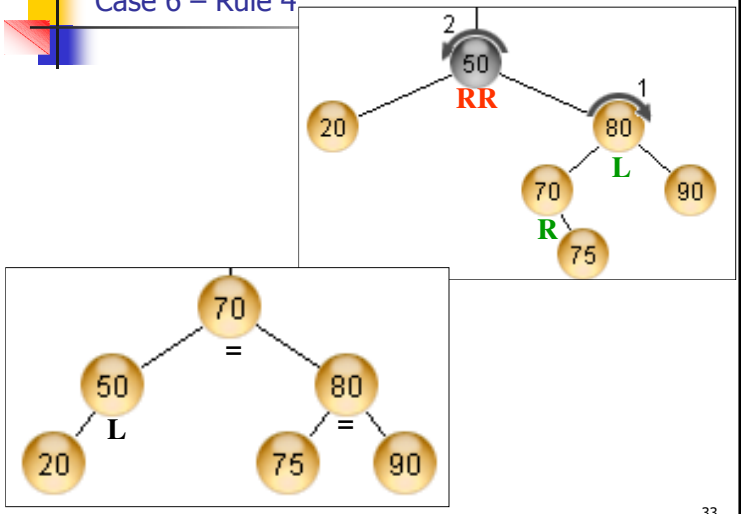
1. Ancestor balance is *R*. Insertion is made in the *right* subtree of the *right* subtree of the ancestor. *Left* rotation around *ancestor*.

2. Ancestor balance is *L*. Insertion is made in the *left* subtree of the *left* subtree of the ancestor. *Right* rotation around *ancestor*.

3. Ancestor balance is *L*. Insertion is made in the *right* subtree of the *left* subtree of the ancestor. *Left* rotation around ancestor's *left child*. *Right* rotation around *ancestor*.

4. Ancestor balance is *R*. Insertion is made in the *left* subtree of the *right* subtree of the ancestor. *Right* rotation around ancestor's *right child*. *Left* rotation around *ancestor*.

Case 6 – Rule 4



33

Deletion From AVL Trees

Case 1: the node to be deleted is a leaf

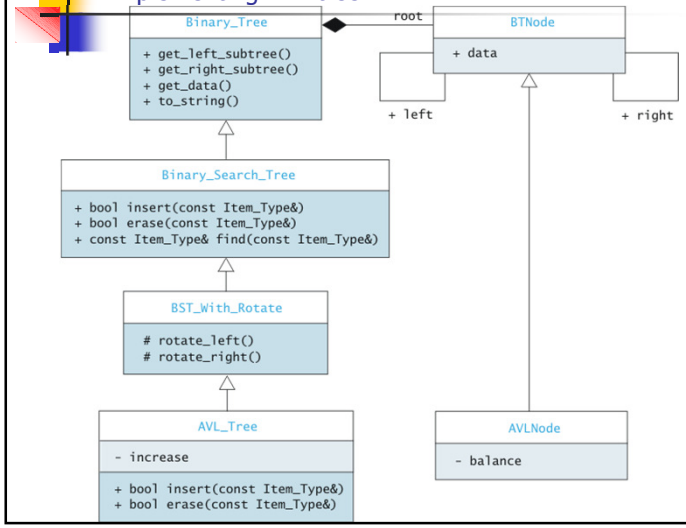
Case 2: the node to be deleted has no right child, that is, its right subtree is empty

Case 3: the node to be deleted has no left child, that is, its left subtree is empty

Case 4: the node to be deleted has a left child and a right child

34

Implementing AVL tree



35

Examples

```
//to define AVL trees as ADT

template<class elemType>
class AVLNode
{
public:
    elemType                    info;
    int                         bfactor;        // balance factor
    AVLNode<elemType> *         llink;
    AVLNode<elemType> *         rlink;
};
```

36

```
template<class elemType>
void rotateToLeft(AVLNode<elemType>* &root)
{
    AVLNode<elemType> *p;          //pointer to the root of the
                                   //right subtree of root
    if(root == NULL)
      cerr<<"Error in the tree."<<endl;
    else
      if(root->rlink == NULL)
        cerr<<"Error in the tree:"
            <<" No right subtree to rotate."<<endl;
      else
      {
        p = root->rlink;
        root->rlink = p->llink;        //the left subtree of p
                                       //becomes the right subtree of root
        p->llink = root;
        root = p;                      //make p the new root node
      }
}       //end rotateLeft
```

```
template<class elemType>
void rotateToRight(AVLNode<elemType>* &root)
{
    AVLNode<elemType> *p;  //pointer to the root of
                           //the left subtree of root
    if(root == NULL)
      cerr<<"Error in the tree."<<endl;
    else
      if(root->llink == NULL)
        cerr<<"Error in the tree:"
            <<" No left subtree to rotate."<<endl;
      else
      {
        p = root->llink;
        root->llink = p->rlink;        //the right subtree of p
                                       //becomes the left subtree of root
        p->rlink = root;
        root = p;                      //make p the new root node
      }
}       //end rotateRight
```



```
template<class elemType>
void balanceFromLeft(AVLNode<elemType>* &root)
{   AVLNode<elemType>   *p,   *w;

    p = root->llink;       //p points to the left subtree of root
    switch(p->bfactor)
    {    case -1: root->bfactor = 0; p->bfactor = 0;
                  rotateToRight(root);
                  break;
         case 0:  cerr<<"Error: Cannot balance from the left."<<endl;
                  break;
         case 1:  w = p->rlink;
           switch(w->bfactor)     //adjust the balance factors
           {    case -1: root->bfactor = 1; p->bfactor = 0;
                         break;
                case 0:  root->bfactor = 0; p->bfactor = 0;
                         break;
                case 1:  root->bfactor = 0; p->bfactor = -1;
           }      //end switch
           w->bfactor = 0;
           rotateToLeft(p);
           root->llink = p;
           rotateToRight(root);
    }    //end switch;
}  //end balanceFromLeft
```



```
template<class elemType>
void balanceFromRight(AVLNode<elemType>* &root)
{   AVLNode<elemType> *p, *w ;
    p = root->rlink;               //p points to the right subtree of root

    switch(p->bfactor)
    { case -1: w = p->llink;
           switch(w->bfactor)              //adjust the balance factors
           { case -1: root->bfactor = 0; p->bfactor = 1;
                      break;
             case 0:  root->bfactor = 0; p->bfactor = 0;
                      break;
             case 1:  root->bfactor = -1; p->bfactor = 0;
           }       //end switch
           w->bfactor = 0;
           rotateToRight(p);
           root->rlink = p;
           rotateToLeft(root);
           break;
      case 0:  cerr<<"Error: Cannot balance from the right."<<endl;
                      break;
      case 1:  root->bfactor = 0; p->bfactor = 0;
                      rotateToLeft(root);
    }    //end switch
}       //end balanceFromRight
```

```
template<class elemType>
void insertIntoAVL(AVLNode<elemType>* &root, AVLNode<elemType>
    *newNode, bool& isTaller)
{  if(root == NULL)
   {  root = newNode;
      isTaller = true;
   }
   else if(root->info == newNode->info)
        cerr<<"No duplicates are allowed."<<endl;
   else if(root->info > newNode->info)  //newItem goes in the left subtree
        {  insertIntoAVL(root->llink, newNode, isTaller);
         if(isTaller)            //after insertion, the subtree grew in height
         switch(root->bfactor)
             {
             case -1: balanceFromLeft(root);
                      isTaller = false;
                      break;
             case 0:  root->bfactor = -1; isTaller = true;
                     break;
             case 1:  root->bfactor = 0; isTaller = false;
           }       //end switch
        }                    //end if
```

```
else
        {
          insertIntoAVL(root->rlink, newNode, isTaller);

          if(isTaller)            //after insertion, the
                                  //subtree grew in height
            switch(root->bfactor)
            {
            case -1: root->bfactor = 0;
                     isTaller = false;
                     break;
            case 0:  root->bfactor = 1;
                     isTaller = true;
                     break;
            case 1:  balanceFromRight(root);
                     isTaller = false;
            }     //end switch
        }          //end else
}        //end insertIntoAVL
```



```
template<class elemType>
void insert(const elemType &newItem)
{
    bool isTaller = false;
    AVLNode<elemType>  *newNode;

    newNode = new AVLNode<elemType>;
    newNode->info = newItem;
    newNode->bfactor = 0;
    newNode->llink = NULL;
    newNode->rlink = NULL;

    insertIntoAVL(root, newNode, isTaller);
}
```


Client program

```
#include <iostream>
#include "avlTree.h"

using namespace std;

int main()
{
   cout<<"Implement this part"<<endl;

   return 0;
}
```