Data Structures

Lecture 22: AVL Trees (cont.)

Nopadon Juneam
Department of Computer Science
Kasetsart university

Outlines

- AVL trees (review)
- Insertions on AVL trees
  - Rotation subroutines
  - Implementation in C

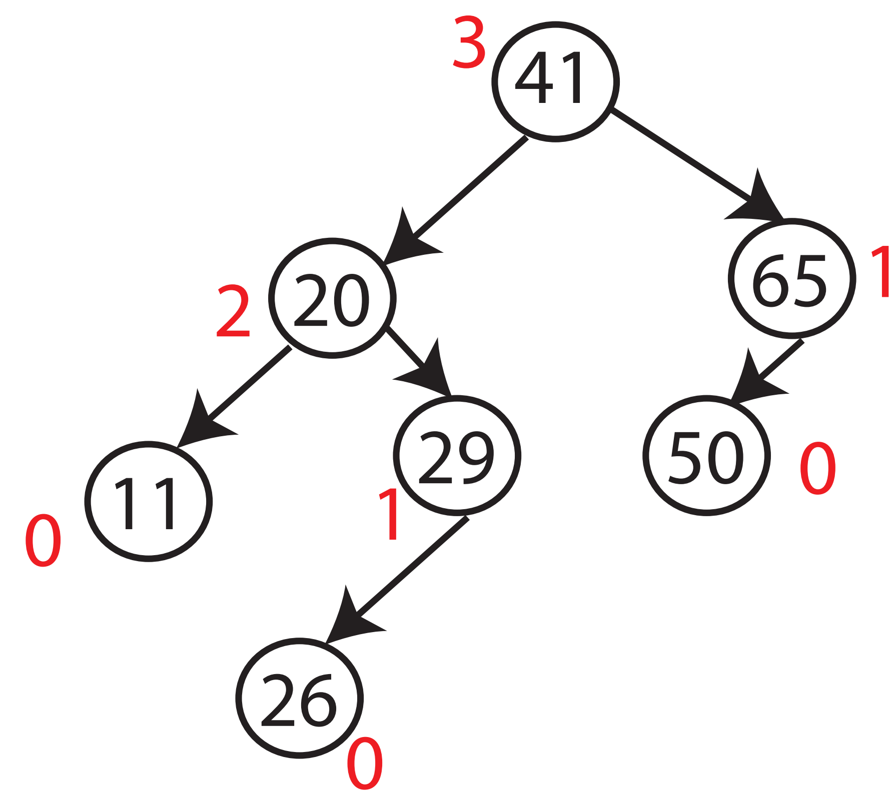
AVL Property

- **AVL property:** For every internal node *v,* the height of the subtrees rooted at children of *v* differs by at most 1.

- **Node's height (subtree's height):** The height of a node *u* in a tree is recursively defined by:

  - If *u* is external, then the height of *u* is zero.

  - Otherwise, the height of *u* is one plus the maximum height of a child of *u.*

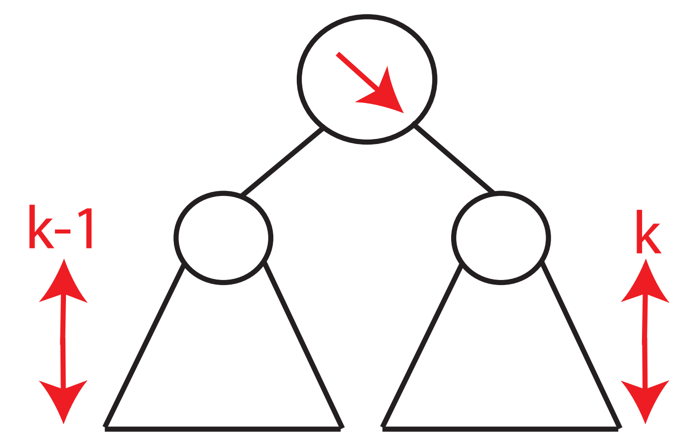


AVL Trees

- Any binary search tree that satisfies the AVL property is said to be an **AVL tree**.

    - An immediate consequence of the AVL property is that a subtree of an AVL tree itself is an AVL tree.

    - In other words, the difference between the heights of left and right subtrees cannot be more than one for all nodes.



- AVL Trees are named after the initials of its inventors (Adelson, Velskii, and Landis 1962).

Complexity of Operations on AVL Trees

Operations	Complexity
`search`	$O(\log n)$
`minimum`	$O(\log n)$
`maximum`	$O(\log n)$
`successor`	$O(\log n)$
`predecessor`	$O(\log n)$
`AVL-insert`	$\boldsymbol{O(\log n)}$

Insertions on AVL Trees

- The insert operations for an AVL tree *T are* similar to those for a binary search tree, but we must perform additional computations to fix the AVL property.

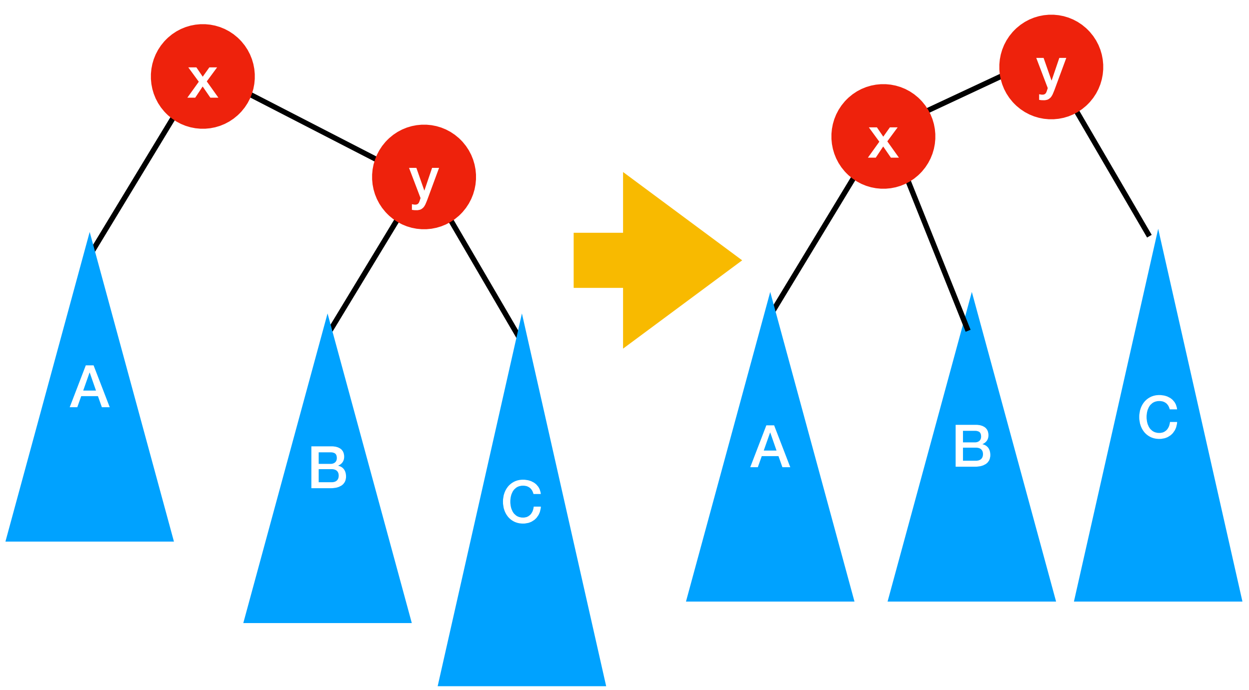
    - **Step 1:** Perform the insertion as in a general binary search tree.

    - **Step 2:** Restructure *T* to restore its height by performing rebalancing process through rotations.

Subroutine: Left-Rotate

- `Left-Rotate(x, T)`: To left rotate subtree rooted at *x* so that *x* becomes left child of *y* (the current right child of *x*) and the left child of *y* becomes the new right child of x.

```
Left-Rotate(x, T):
   y = x.right
   B = y.left
   y.left = x
   x.parent = y
   x.right = B
   B.parent = x
   // Update heights of x & y
   x.height = 1 + max(height(x.left),
         height(x.right))
   y.height = 1 + max(height(y.left),
         height(y.right))
```



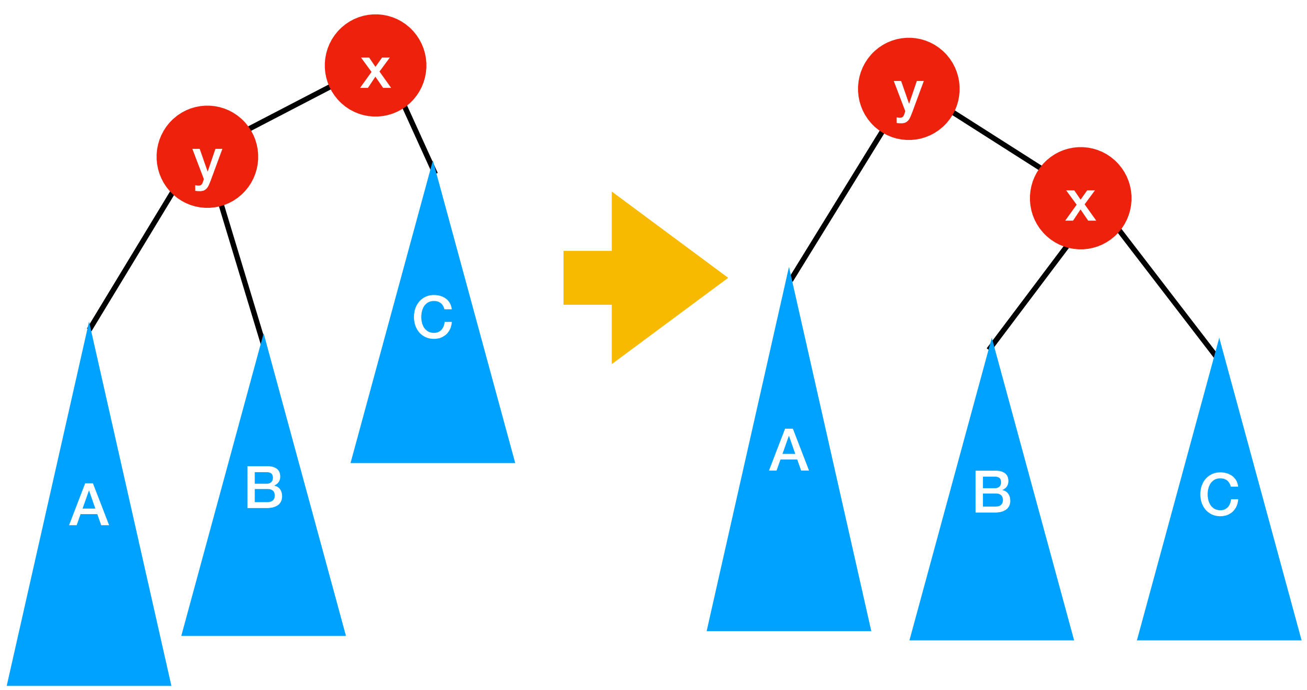

```
height(u):
  if (u == NULL):
    return -1
  return u.height
```

Subroutine: Right-Rotate

- `Right-Rotate(x, T):` To right rotate subtree rooted at *x* so that *x* becomes right child of *y* (the current left child of *x*) and the right child of *y* becomes the new left child of x.

```
Right-Rotate(x, T):
   y = x.left
   B = y.right
   y.right = x
   x.parent = y
   x.left = B
   B.parent = x
   // Update heights of x & y
   x.height = 1 + max(height(x.left),
         height(x.right))
   y.height = 1 + max(height(y.left),
         height(y.right))
```

```
height(u):
  if (u == NULL):
    return -1
  return u.height
```



Rebalancing Process

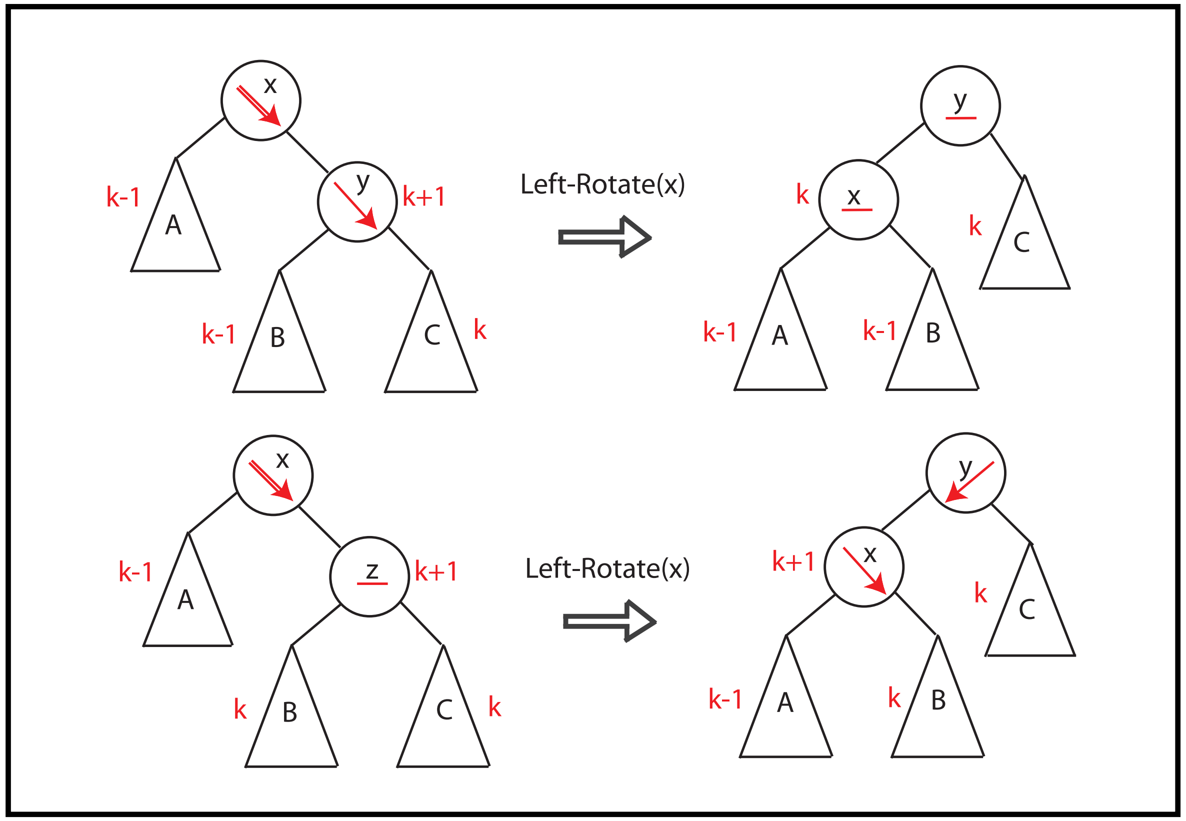
- Suppose *x* is lowest node violating the height-balance property.

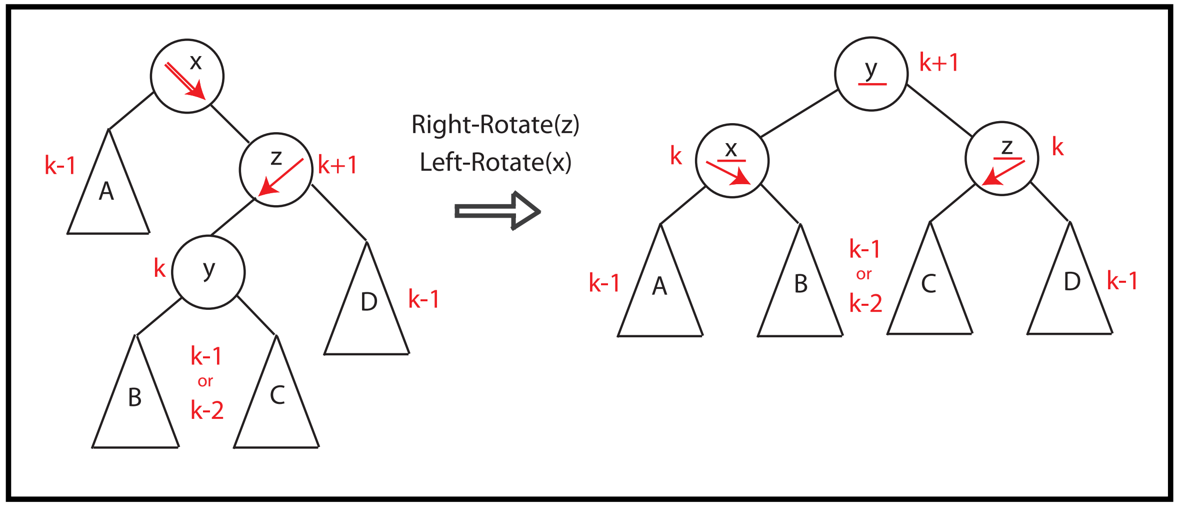
- Assume *x* is right-heavy (the left case is symmetric).

  - **Right-right case:** If x's right child is right-heavy or balanced, perform left rotation.

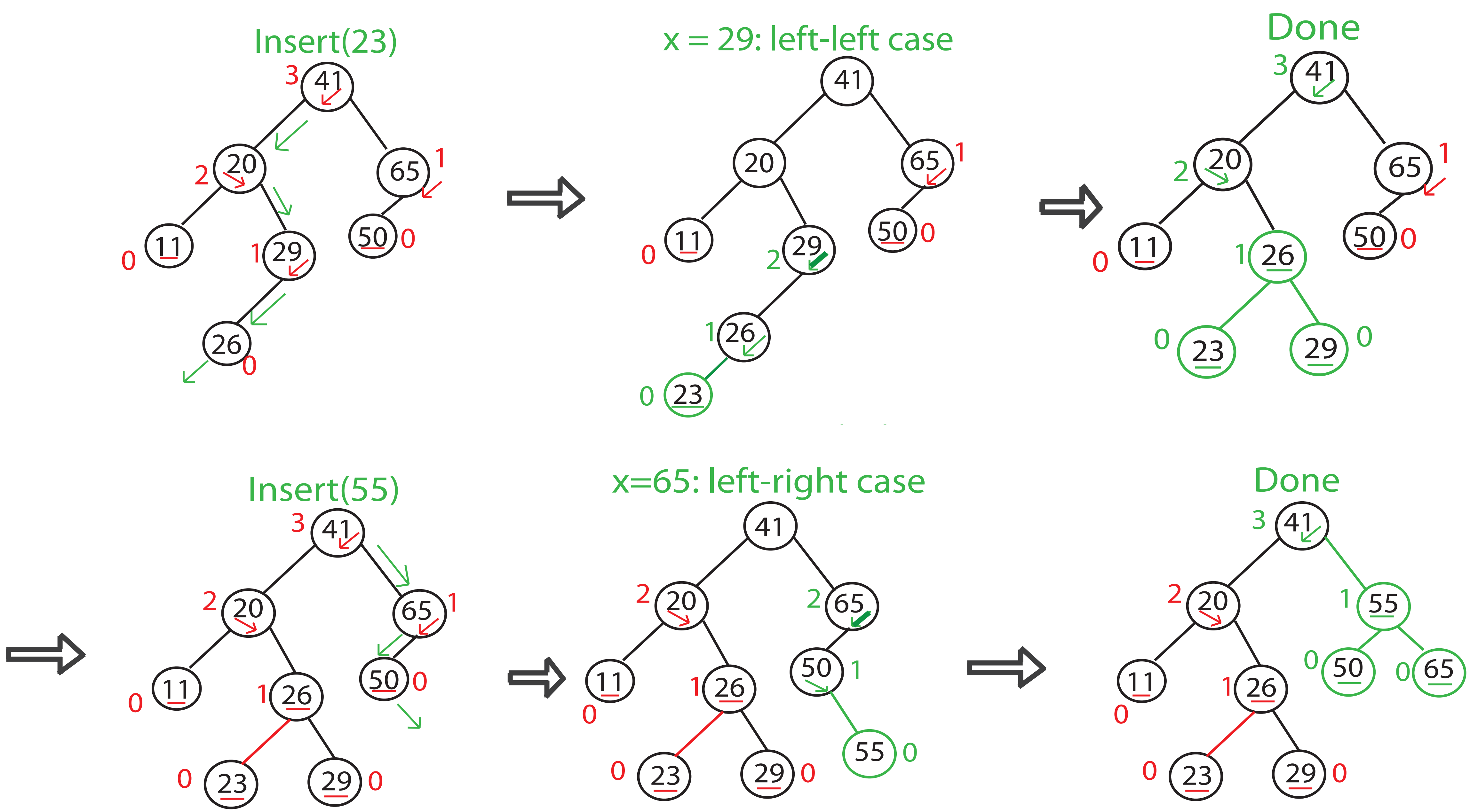
  - **Right-left case:** Else, perform two rotations (right rotation, and left rotation).

- Then, continue up to *x*'s grandparent, great-grandparent, and so on.





Examples of Insertions on AVL Trees



Operation: AVL-Insert

- `AVL-Insert(z, u, T)`: insert a new node *z* for which *z.key* = *v*, z.*parent* = NULL, *z.left* = NULL, and *z.right* = NULL into an appropriate position in the AVL subtree of *T* rooted at *u*;

```
AVL-Insert(z, u, T):
   // Step 1: Perform an insertion as in general BST
   if (r == NULL):
      return z
   if (z.key < u.key):
      u.left = AVL-Insert(z, u.left, T)
   else:
      u.right = AVL-Insert(z, u.right, T)
   // Step 2: Update height of node u
   u.height = 1 + max(height(u.left), height(u.right))
   b = checkBalanced(u)
   // Step 3: If u becomes unbalanced, then there are 4 cases
   // Right-right case
   if (b < -1 and z.key > u.right.key):
       Left-Rotate(u, T)
   …
   // Right-left case
   if (b < -1 and z.key > u.left.key):
       Right-Rotate(u.right, T)
       Left-Rotate(u, T)
   return u
```

```
height(u):
  if (u == NULL):
    return -1
  return u.height
```

```
node:
  node* left
  node* right
  node* parent
  int   key
  int   height
```

```
checkBalanced(u):
  if (u == NULL):
    return -1
  return height(u.left)-height(u.right)
```

Complexity of Operations on AVL Trees

Operations	Complexity
`search`	$O(\log n)$
`minimum`	$O(\log n)$
`maximum`	$O(\log n)$
`successor`	$O(\log n)$
`predecessor`	$O(\log n)$
`AVL-insert`	$\boldsymbol{O(\log n)}$

Implementation of AVL-Insert in C (1)

```c
// C program to insert a node in AVL tree
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct node
{
    int key;
    struct node *left;
    struct node *right;
    struct node *parent;
    int height;
};

// A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}

// A utility function to get the height of the tree
int height(struct node* node)
{
    if (node == NULL)
        return -1;
    return node->height;
}
```

Implementation of AVL-Insert in C (2)

```
// Helper function that allocates a new node with the given key
struct node* createNode(int key)
{
    struct node* node = (struct node*)
    malloc(sizeof(struct node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->parent = NULL;
    node->height = 0; // new node is initially added at leaf
    return(node);
}

// A utility function to left rotate subtree rooted with x
struct node* leftRotate(struct node* x)
{
    struct node* y = x->right;
    struct node* B = y->left;
    // Perform rotation
    y->left = x;
    x->parent = y;
    x->right = B;
    if(B != NULL)
       B->parent = x;
    // Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;
    // Return new root
    return y;
}
```

Implementation of AVL-Insert in C (3)

```c
// A utility function to right rotate subtree rooted at x
struct node* rightRotate(struct node* x)
{
    struct node* y = x->left;
    struct node* B = y->right;
    // Perform rotation
    y->right = x;
    x->parent = y;
    x->left = B;
    if(B != NULL)
        B->parent = x;
    // Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;
    // Return new root
    return y;
}

// Get Balance factor of a node
int getBalance(struct node* node)
{
    if (node == NULL)
        return -1;
    return height(node->left)-height(node->right);
}
```

Implementation of AVL-Insert in C (4)

```c
// Recursive function to insert a key in the subtree rooted with node and returns the new root of the subtree.
struct node* insert(struct node* node, int key)
{
    /* 1. Perform the normal BST insertion */
    if (node == NULL)
        return(createNode(key));
    if (key < node->key) {
        node->left = insert(node->left, key);
        node->left->parent = node;
    }
    else if (key > node->key) {
        node->right = insert(node->right, key);
        node->right->parent = node;
    }
    else // Equal keys are not allowed in BST
        return node;
    /* 2. Update height of this ancestor node */
    node->height = 1 + max(height(node->left), height(node->right));
    /* 3. Get the balance factor of this ancestor node to check whether this node became unbalanced */
    int balance = getBalance(node);
    // If this node becomes unbalanced, then there are 4 cases
    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);
    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);
    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
    /* return the (unchanged) node pointer */
    return node;
}
```

Implementation of AVL-Insert in C (5)

```
// A utility function to print preorder traversal of the tree. The function also prints height of
every node
void inorder(struct node* node)
{
    if(node != NULL)
    {
        inorder(node->left);
        printf("key: %d, height: %d\n", node->key, node->height);
        inorder(node->right);
    }
}

/* Drier program to test above function*/
int main()
{
    struct node *root = NULL;

    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    printf("Preorder traversal of the constructed AVL"
           " tree is \n");
    inorder(root);

    return 0;
}
```

```
nj@Nopadons-MacBook-Pro codes % ./
AVL_tree
Inorder traversal of the constructed AVL
tree is
key: 10, height: 0
key: 20, height: 1
key: 25, height: 0
key: 30, height: 2
key: 40, height: 1
key: 50, height: 0
nj@Nopadons-MacBook-Pro codes %
```