SCS1308 - Foundations of Algorithm

AVL Tree

AVL Tree
Data Structure

Binary search tree properties

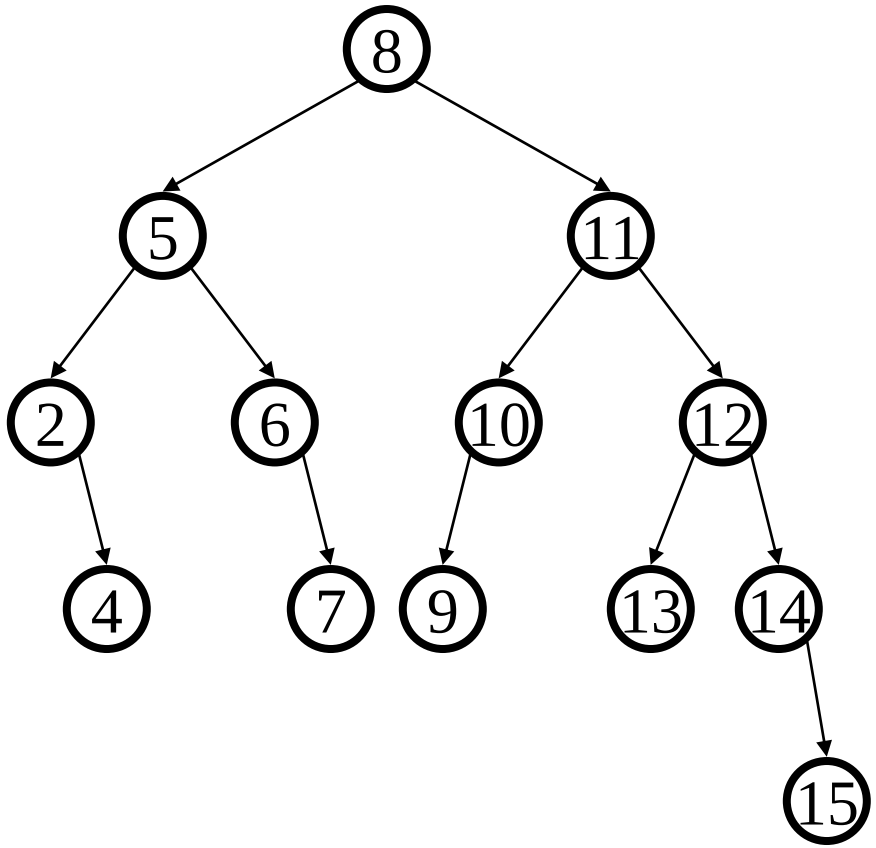
- binary tree property
- search tree property

Balance property

- balance of every node is:

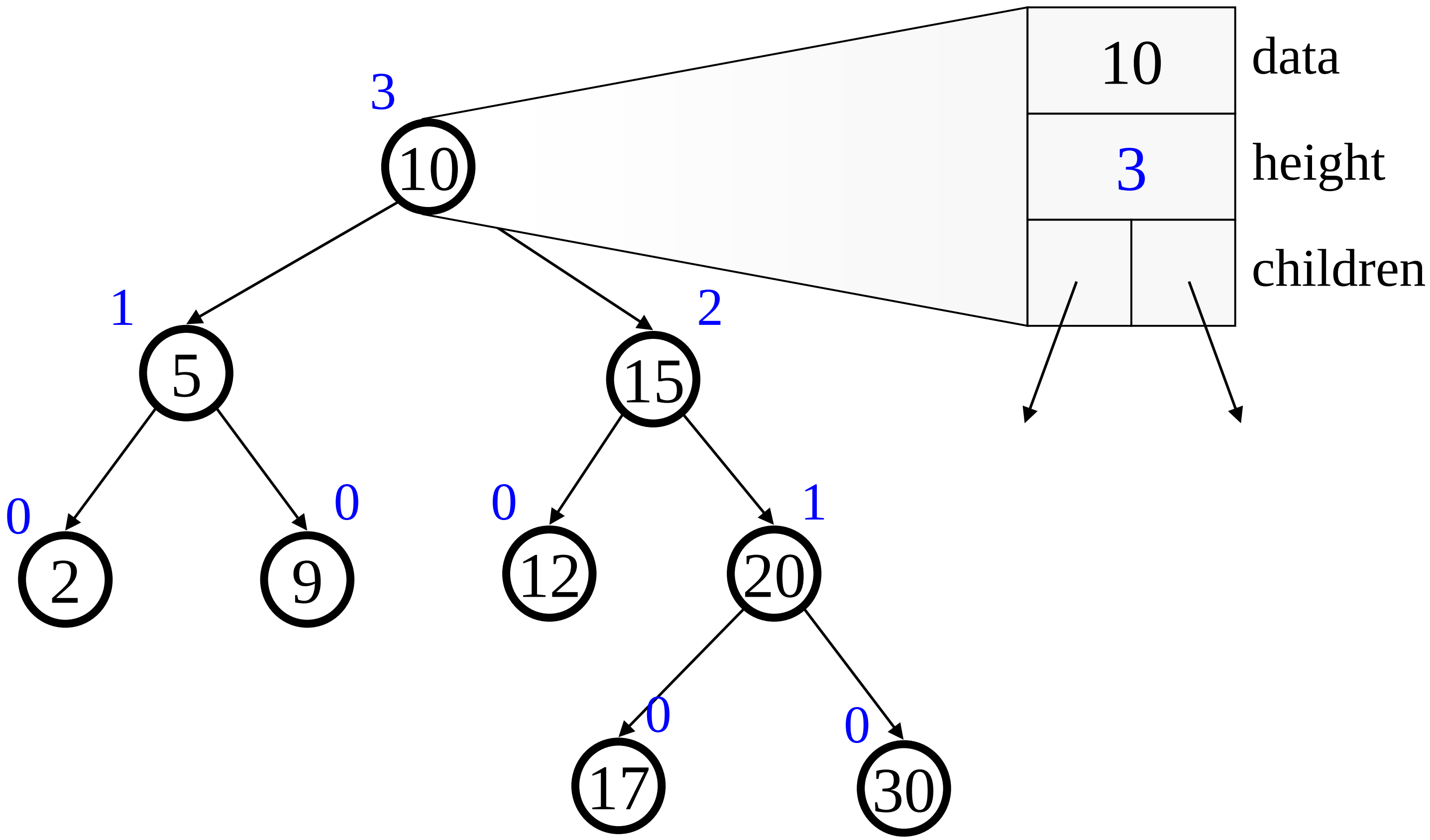
  $-1 \leq b \leq 1$

- result:
  - depth is Θ(log n)

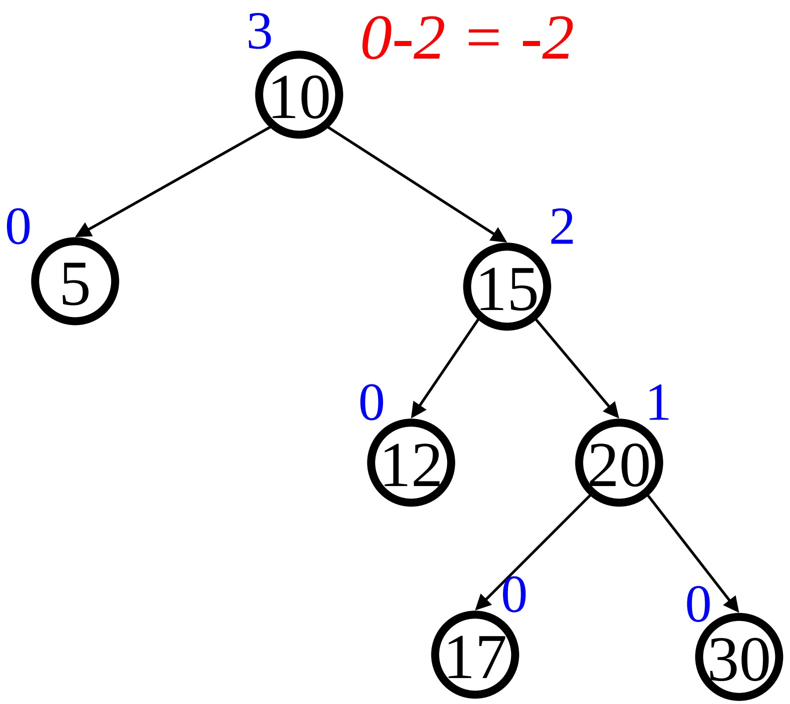


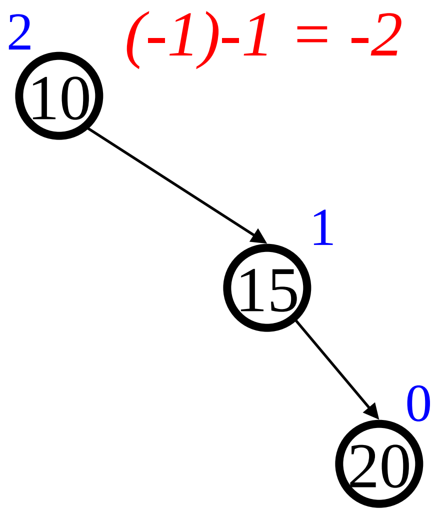
Balance == height(left subtree) - height(right subtree

An AVL Tree



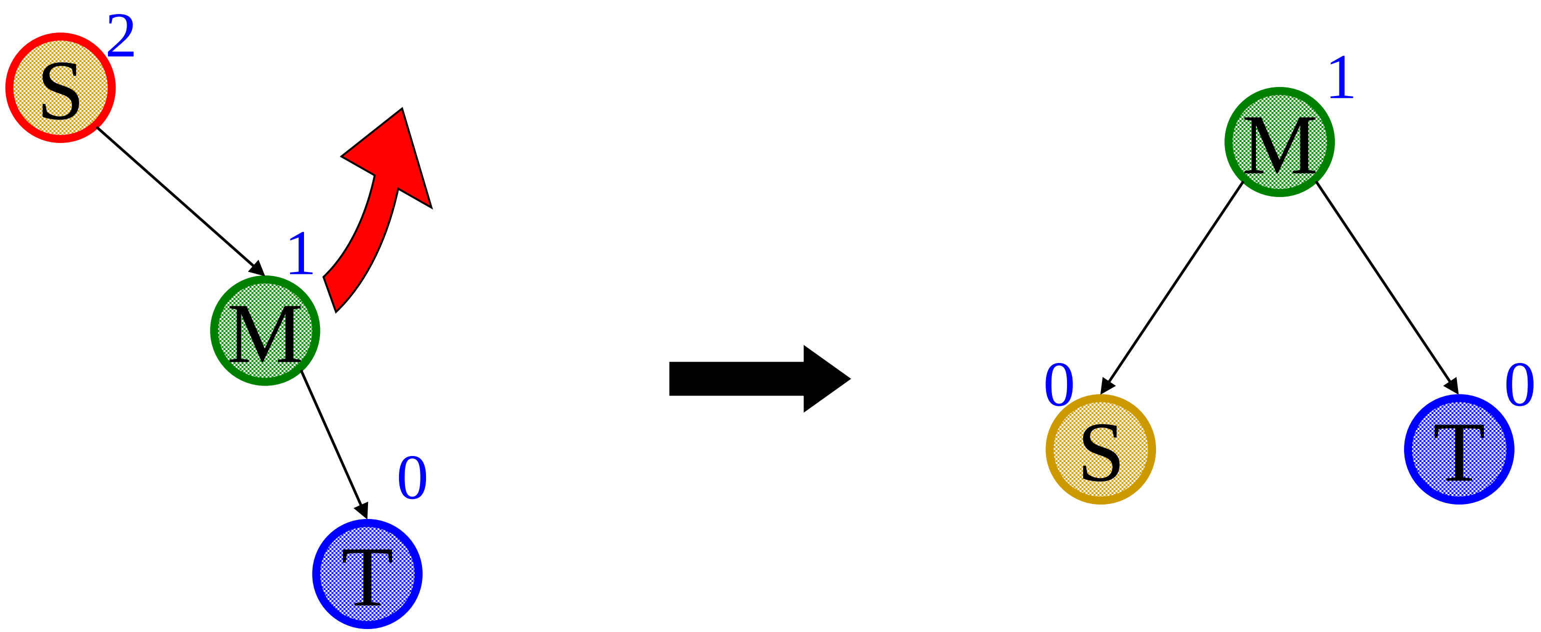
Not AVL Trees





Note: height(empty tree) == -1

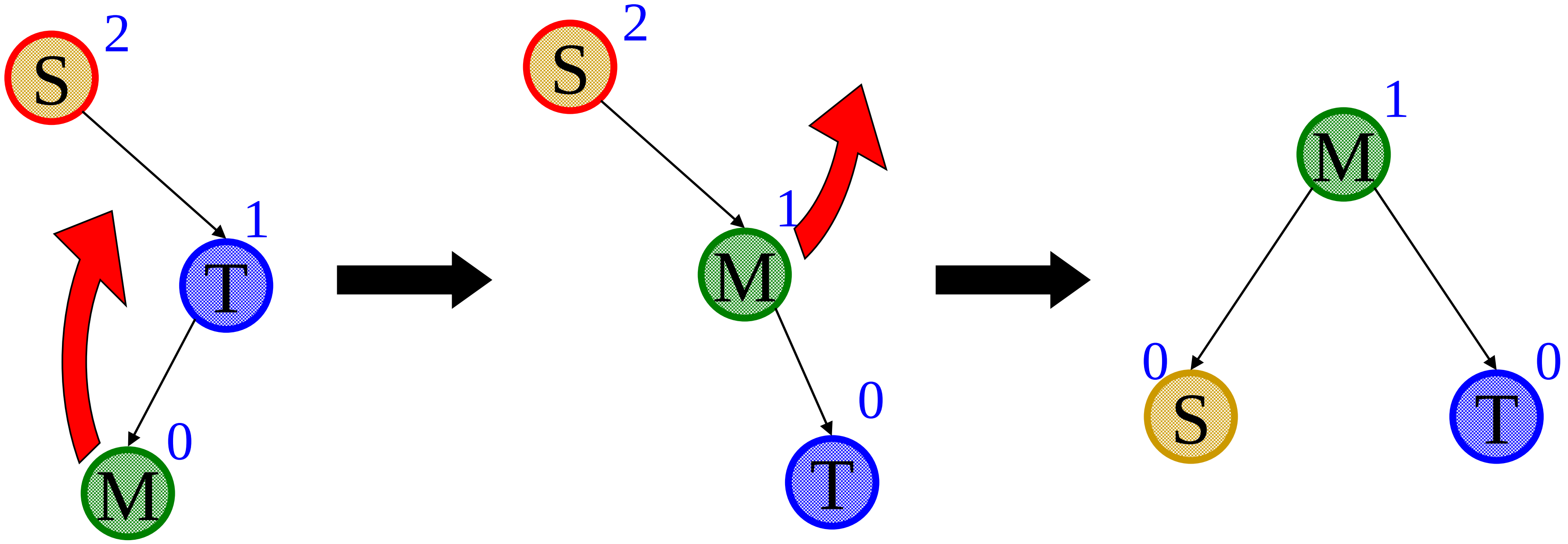
Single Rotation



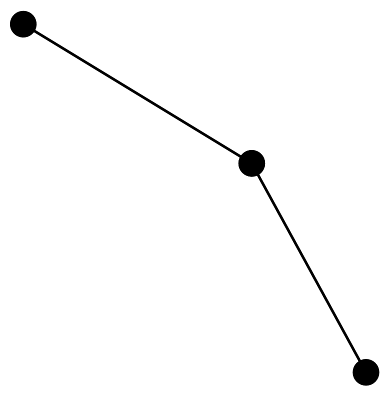
Basic operation used in AVL trees:

A right child could legally have its parent as its left child.

Double Rotation

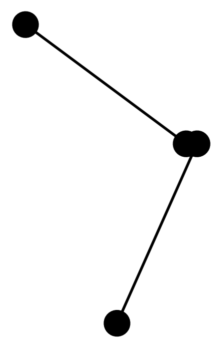


AVL Insert Algorithm



- Find spot for value
- Hang new node
- Search back up looking for imbalance
- If there is an imbalance:

  case #1: Perform single rotation

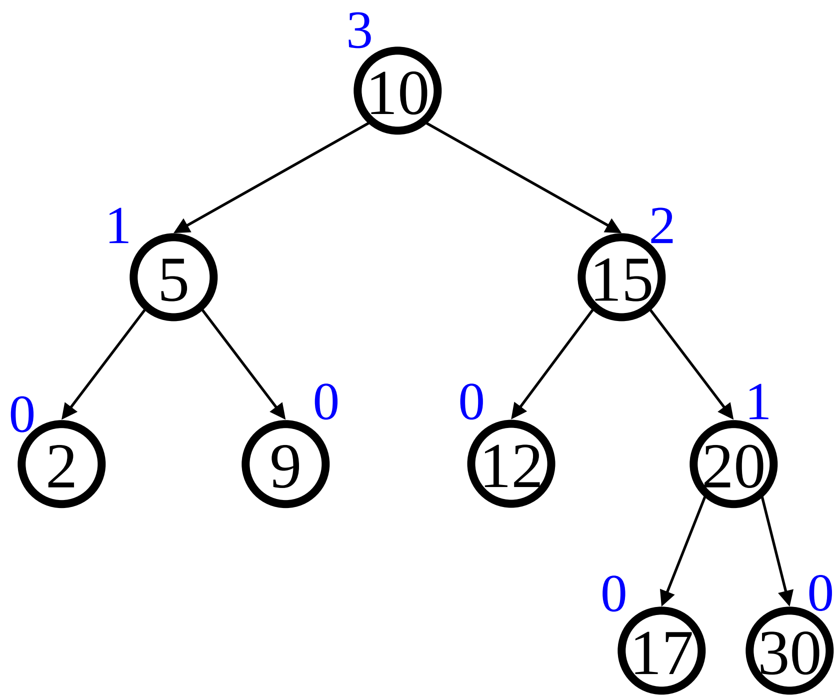
  case #2: Perform double rotation

- *Done!*

  (There can only be one imbalance!)

Easy Insert

Insert(3)



Node structure, Height, Balance Factor

```c
// A Node structure to store key,
// left child, right child, and height of each node
struct Node {
    int key;
    struct Node *left;
    struct Node *right;
    int height;
};

// Function to get the height of a node
int height(struct Node *node) {
    if (node == NULL) {
        return 0;
    }
    return node->height;
}

// Function to get the balance factor of a node
int getBalance(struct Node *node) {
    if (node == NULL) {
        return 0;
    }
    return height(node->left) - height(node->right);
}
```

Rotations

```c
// Function to perform a right rotation
struct Node* rightRotate(struct Node *y) {
    struct Node *x = y->left;
    struct Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right));
    x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));

    // Return new root
    return x;
}

// Function to perform a left rotation
struct Node* leftRotate(struct Node *x) {
    struct Node *y = x->right;
    struct Node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));
    y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right));

    // Return new root
    return y;
}
```

Insert function

```c
// Function to insert a node in the AVL tree
struct Node* insert(struct Node* node, int key) {
    // 1. Perform normal BST insert
    if (node == NULL) {
        struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->key = key;
        newNode->left = newNode->right = NULL;
        newNode->height = 1;
        return newNode;
    }

    if (key < node->key) {
        node->left = insert(node->left, key);
    } else if (key > node->key) {
        node->right = insert(node->right, key);
    } else {
        return node; // Duplicate keys are not allowed
    }

    // 2. Update height of this ancestor node
    node->height = 1 + (height(node->left) > height(node->right) ? 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);
```

Balancing Case

```
    // If the 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 node;
}
```

Print in sorted order

```c
// Function to do an inorder traversal of the tree
void inorder(struct Node *root) {
    if (root != NULL) {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

// Main function to test the AVL tree
int main() {
    struct Node* root = NULL;

    // Insert nodes into the AVL tree
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 15);
    root = insert(root, 25);

    // Print the inorder traversal of the AVL tree
    printf("Inorder traversal of the AVL tree: ");
    inorder(root);
    printf("\n");

    return 0;
}
```