AVL Trees

Lesson 6.3

Learning Objectives

LO 6.3.1 **Insert** correctly an element into an AVL Tree

LO 6.3.2 **Delete** an element successfully in an AVL Tree

LO 6.3.3 **Ensure** that all instances of AVL Trees are always balanced

AVL Trees

- An **AVL tree** is a binary tree that either is empty or is a *balanced binary tree* (*see Lesson 6.1*) that operates on these four cases that require rebalancing:

    1. **Left of left** (*L of L*) — A subtree of a tree that is left high has also become left high.

    2. **Right of right** (*R of R*) — A subtree of a tree that is right high has also become right high.

    3. **Right of left** (*R of L*) — A subtree of a tree that is left high has become right high.

    4. **Left of right** (*L of R*) — A subtree of a tree that is right high has become left high.

AVL Tree Operations

- *Inserting an element into the AVL Tree*

```
algorithm insertAVL (root, newData)
    Using recursion, insert a node into an AVL tree.
    Pre:    root is pointer to first node in AVL tree/subtree
            newData is pointer to new node to be inserted
    Post:   new node has been inserted
    Return: root returned recursively up the tree
    if (empty subtree)
        insert newData at root
        return root
    end if
    if (data of newData < data of root)
        insertAVL(left subtree, newData)
        if (left subtree is taller)
            leftBalance (root)
        end if
    ……
        else
            insertAVL(right subtree, newData)
            if (right subtree is taller)
                rightBalance (root)
            end if
        end if
        return root
end insertAVL
```

AVL Tree Operations

- *Balance a left high root of a subtree*

```
algorithm leftBalance (root)
    This algorithm is entered when the root is left high (the left subtree is
      higher than the right subtree).
    Pre:  root is a pointer to the root of the [sub]tree
    Post: root has been updated (if necessary)
    if (left subtree high)
        rotateRight (root)
    else
        rotateLeft (left subtree)
        rotateRight (root)
    end if
end leftBalance
```

AVL Tree Operations

- *Balance a right high root of a subtree*

```
algorithm rightBalance (root)
    This algorithm is entered when the root is right high (the right
      subtree is higher than the left subtree).
    Pre:  root is a pointer to the root of the [sub]tree
    Post: root has been updated (if necessary)
    if (right subtree high)
        rotateLeft (root)
    else
        rotateRight (right subtree)
        rotateLeft (root)
    end if
end rightBalance
```

AVL Tree Operations

- *Rotating a subtree to the right/left*

algorithm *rotateRight* (*root*)
This algorithm exchanges pointers to rotate the tree right.
Pre: root points to tree to be rotated
Post: node rotated and root updated
exchange left subtree with right subtree of left subtree
make left subtree new root
end *rotateRight*

algorithm *rotateLeft* (*root*)
This algorithm exchanges pointers to rotate the tree left.
Pre: root points to tree to be rotated
Post: node rotated and root updated
exchange right subtree with left subtree of right subtree
make right subtree new root
end *rotateLeft*

LO 6.3.1 Insert correctly an element into an AVL Tree

LO 6.3.3 Ensure that all instances of AVL Trees are always balanced

Insert the following respectively into an initially empty AVL BST:

14, 23, 7, 10, 56, 70, 80, 66, 33, 100

AVL Tree Operations

- *Deleting an element from the AVL Tree*

```
algorithm deleteAVL (root, dltKey, success)
    This algorithm deletes a node from an AVL tree and rebalances if necessary..
    Pre:        root is a pointer to a [sub]tree
                dltKey is the key of node to be deleted
                success is reference to boolean variable
    Post:       node deleted if found, tree unchanged if not, success set true (key found and deleted) or false (key not found)
    Return:     pointer to root of [potential] new subtree
    if (empty subtree)
        set success to false
        return null
    end if
    if (dltKey < data of root)
        set left subtree to deleteAVL(left subtree, dltKey, success)
        if (right subtree is taller)
            rightBalance(root)
        end if
    else if (dltKey > data of root)
        set right subtree to deleteAVL(right subtree, dltKey, success)
        if (left subtree is taller)
            leftBalance(root)
        end if
```

……

AVL Tree Operations

- *Deleting an element from the AVL Tree*

```
……
        else
                if (no right subtree)
                        set success to true
                        return left subtree
                else if (no left subtree)
                        set success to true
                        return right subtree
                else
                        set root data to the largest data in the left subtree
                        set left subtree to deleteAVL(left subtree, data of root, success)
                        if (right subtree is taller)
                                rightBalance(root)
                        end if
                end if
        end if
        return root
end deleteAVL
```

LO 6.3.2 Delete an element successfully in an AVL Tree
LO 6.3.3 Ensure that all instances of AVL Trees are always balanced

Delete the following respectively into the AVL BST you have just previously finished inserting:

14, 7, 33, 23