Scuola universitaria professionale
della Svizzera italiana
SUPSI

Università
della
Svizzera
italiana

Istituto Dalle Molle di studi sull'intelligenza artificiale

Algorithms and Data Structures

AVL Trees

Matteo Salani
matteo.salani@idsia.ch

AVL Tree

In Bynary Search Trees (BST) *search, insert and delete* procedures exibit $O(\mathit{height})$ complexity. Unfortunately BST height can degenerate to n.

- **AVL Trees** are BSTs in which height is controlled and maintained within $O(\log n)$.

AVL Tree Invariant

AVL Trees maintain at each node a property called **Balance factor (BF)**

Balance Factor

$$BF(Node) = H(node.right) - H(node.left)$$

where $H(node)$ is the height of the node (i.e. the length of the path from the node to the farther leaf).

- The balance factor must be maintained in {-1, 0, 1} in order for an AVL Trees to be balanced.
- This guarantees that an AVL tree with n elements has at most $O(\log n)$ levels.

Computing height and balance factor

```
def update(node):

    lh = -1
    rh = -1
    if node.left is not null: lh = node.left.height
    if node.right is not null: rh = node.right.height

    #Update this node's height.
    node.height = 1 + max(rh, lh)

    #Update balance factor.
    node.bf = rh - lh

    return
```

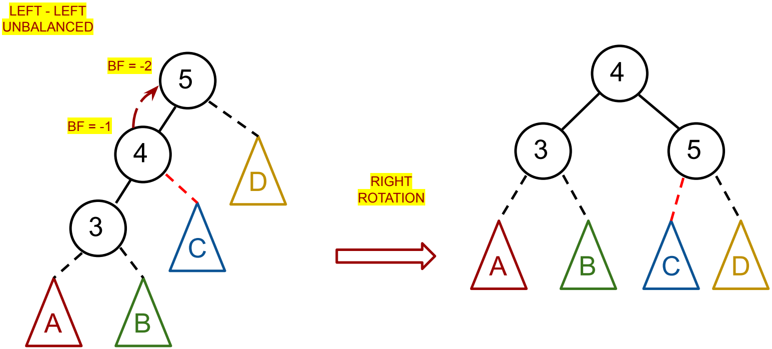
Listing 1: Update method

Re-balancing AVL trees

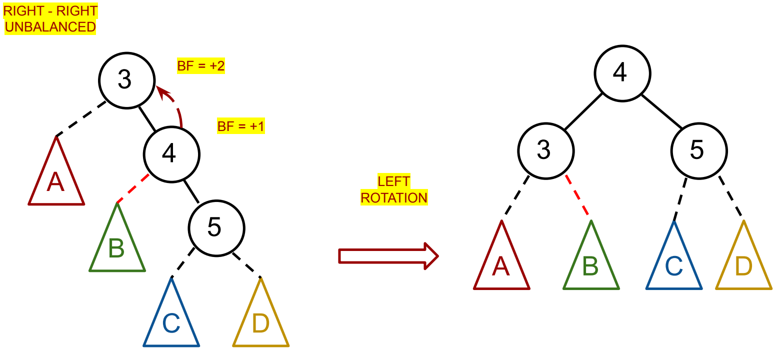
Inserting or deleting nodes from an AVL tree may break the tree's balancement.
Tree rotations are the procedures necessary to rebalance the AVL tree.
There are four types of tree rotation and they are applied according to the type of unbalancement.

- **Single right rotation** necessary to rebalance the tree when the tree is **left left unbalanced**
- **Single left rotation** necessary to rebalance the tree when the tree is **right right unbalanced**
- **Double right left rotation** necessary to rebalance the tree when the tree is **right left unbalanced**
- **Double left right rotation** necessary to rebalance the tree when the tree is **left right unbalanced**

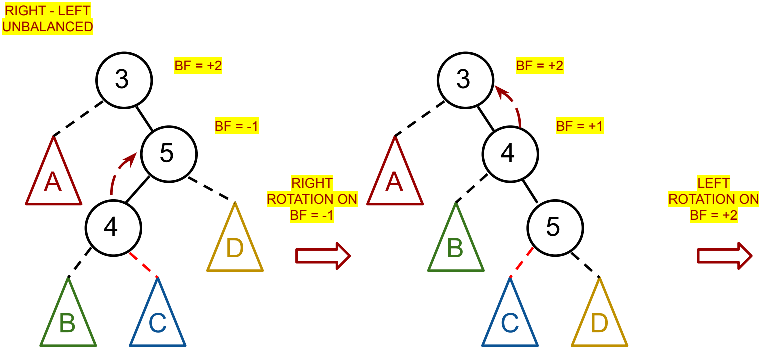
Single Right Rotation - Left Left unbalaced



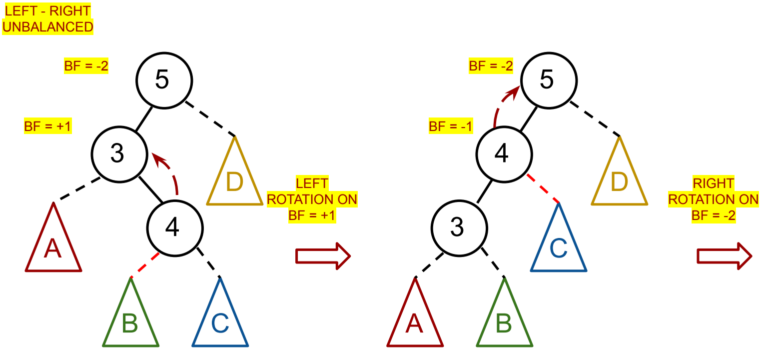
Single Left Rotation - Right Right unbalaced



Double Rotation - Right Left unbalaced



Double Rotation - Left Right unbalaced



Re-balancing AVL trees

In order to detect the type of unbalancement we observe the balance factor of the first unbalanced node, i.e. that with value -2 or $+2$ and consequently the balance factor of the *left* or the *right* child, respectively.

```
def detect(node):

    if node.bf == -2:
        if node.left.bf < 0:
            res = 'left left unbalanced'
        else
            res = 'left right unbalanced'
    else if node.bf == 2:
        if node.right.bf > 0:
            res = 'right right unbalanced'
        else
            res = 'right left unbalanced'
    else:
        res = 'balanced'

    return res
```

Listing 2: Detect method

Re-balancing AVL trees

```
def balance(node):

    res = detect(node)

    if res == 'left left unbalanced':
        return right_rotate(node)
    else if res == 'left right unbalanced'
        node.left = left_rotate(node.left)
        return right_rotate(node)
    else if res == 'right right unbalanced':
        return left_rotate(node)
    else if res == 'right left unbalanced'
        node.right = right_rotate(node.right)
        return left_rotate(node)

    return node
```

Listing 3: Balance method

Re-balancing AVL trees

Note that nodes' height and balance factor may change during rotation and they must be updated

```python
def right_rotate(node):

  tmp = node.left
  node.left = tmp.right
  tmp.right = node

  update(tmp)
  update(node)
  return tmp

def left_rotate(node):

  tmp = node.right
  node.right = tmp.left
  tmp.left = node

  update(tmp)
  update(node)
  return tmp

```

Listing 4: Right and Left rotations

Inserting in AVL trees

```
def insert(node, key):

    # Create a new leaf
    if node is null:
        return Node(key)

    if key < node.key:
        node.left = insert(node.left, key)
    else
        node.right = insert(node.right, key)

    # Update height and balance factor
    update(node)
    #Rebalance tree if necessary
    return balance(node)
```

Listing 5: Insert method

Deleting from AVL trees

```
def delete(node, key):

    # Same as BST
    if node is null: return null
    if key<node.key:   node.left = delete(node.left, key)
    elif key>node.key: node.right = delete(node.right, key)
    # Element to be deleted found
    else
      # One child cases
      if node.left is null return node.right
      elif node.right is null return node.left
      # Two children case
      else
        T = findMax(node.left)
        node.key = T
        node.left = delete(node.left, T)

    # Update height and balance factor
    update(node)
    #Rebalance tree if necessary
    return balance(node)
```

Listing 6: Delete method