Day 17 - 12th July 2025

AVL Tree

Adelson-Velsky and Evgenii Landis who invented the AVL Tree in 1962.

AVL trees are self-balancing, which means that the tree height is kept to a minimum so that a very fast runtime is guaranteed for searching, inserting and deleting nodes, with time complexity

O(log n).

The Balance Factor (BF) node (X) = difference in height of Right - Left subtrees.

Bf = 3 - 1 = 2

Balance factor values

0: The node is in balance.

more than 0: The node is "right heavy".

less than 0: The node is "left heavy".

Need clarification on this left - right or right - left

Task 1:

Write Algo for AVL tree

Avl insertion Algorithem

1. create a node

2. check if tree is empty or not

3. if tree is empty the inserted node will be the root node.

4. if tree is not empty , do a binary search tree insertion op and also check the balance factor of the node.

5. if the balance factor exceeds 1, we should do rotations on the heavy weighted tree and repeat the insertion from step 4 onwards.

AVL search Algo

1 − Create a node

2 − Check if tree is empty

3 − If tree is empty, new node is root node.

4 − not empty, perform Binary Search Tree insertion operation and check balancing factor of the node in the tree.

5 − Suppose balancing factor > apply rotations on node and resume insertion from Step 4.

Task 2:

import java.util.\*;

class Node {

   int key, height;

   Node left, right;

   Node (int d) {

      key = d;

      height = 1;

   }

}

public class AVLTree {

   Node root;

   int height (Node N) {

      if (N == null)

         return 0;

      return N.height;

   }

   int max (int a, int b) {

      return (a > b) ? a : b;

   }

   Node rightRotate (Node y) {

      Node x = y.left;

      Node T2 = x.right;

      x.right = y;

      y.left = T2;

      y.height = max (height (y.left), height (y.right)) + 1;

      x.height = max (height (x.left), height (x.right)) + 1;

      return x;

   }

   Node leftRotate (Node x) {  x =5 5,15,44

      Node y = x.right; x.right = 15 = y 15

5 44

      Node T2 = y.left;  //y.left = null = T2     null

      y.left = x; = //y.left = 5= x

      x.right = T2;  //x.right = null = t2;

      x.height = max (height (x.left), height (x.right)) + 1;

      y.height = max (height (y.left), height (y.right)) + 1;

      return y;

   }

   int getBalance (Node N) {

      if (N == null)

         return 0;

      return height (N.left) - height (N.right);

   }

   Node insert (Node node, int key) {

      if (node == null)

         return (new Node (key));

      if (key < node.key)

         node.left = insert (node.left, key);

      else if (key > node.key)

         node.right = insert (node.right, key);

      else

         return node;

      node.height = 1 + max (height (node.left), height (node.right));

      int balance = getBalance (node);

      if (balance > 1 && key < node.left.key)

         return rightRotate (node);

      if (balance < -1 && key > node.right.key)

         return leftRotate (node);

      if (balance > 1 && key > node.left.key) {

         node.left = leftRotate (node.left);

         return rightRotate (node);

      }

      if (balance < -1 && key < node.right.key) {

         node.right = rightRotate (node.right);

         return leftRotate (node);

      }

      return node;

   }

   void printTree(Node root){

   if (root == null)

      return;

   if (root != null) {

      printTree(root.left);

      System.out.print(root.key + " ");

      printTree(root.left);

   }

}

   public static void main(String args[]) {

      AVLTree tree = new AVLTree();

      tree.root = tree.insert(tree.root, 5);

      tree.root = tree.insert(tree.root, 15);

      tree.root = tree.insert(tree.root, 44);

      tree.root = tree.insert(tree.root, 35);

      tree.root = tree.insert(tree.root, 65);

      tree.root = tree.insert(tree.root, 78);

      System.out.println("AVL Tree: ");

      tree.printTree(tree.root);

   }

}

Write code for AVL tree

Hint: try to insert nodes

While inserting get the balance of the tree

Create 2 methods for left rotate and right rotate

Try to insert

Finally display

class AVLNode {

int key, height;

AVLNode left, right;

AVLNode(int key) {

this.key = key;

this.height = 1;

}

}

class AVLTree {

AVLNode root;

// Get height of the node

int height(AVLNode node) {

if (node == null)

return 0;

return node.height;

}

// Get balance factor of node

int getBalance(AVLNode node) {

if (node == null)

return 0;

return height(node.right) - height(node.left);

}

// Update height of the node

void updateHeight(AVLNode node) {

if (node == null)

return;

node.height = Math.*max*(height(node.left), height(node.right)) + 1;

}

// Right rotate

AVLNode rightRotate(AVLNode y) {

AVLNode x = y.left;

AVLNode T2 = x.right;

x.right = y;

y.left = T2;

updateHeight(y);

updateHeight(x);

return x;

}

// Left rotate

AVLNode leftRotate(AVLNode x) {

AVLNode y = x.right;

AVLNode T2 = y.left;

y.left = x;

x.right = T2;

updateHeight(x);

updateHeight(y);

return y;

}

// Insert a node

AVLNode insert(AVLNode node, int key) {

// 1. Perform normal BST insertion

if (node == null)

return new AVLNode(key);

if (key < node.key)

node.left = insert(node.left, key);

else if (key > node.key)

node.right = insert(node.right, key);

else // Duplicate keys not allowed

return node;

// 2. Update height of current node

updateHeight(node);

// 3. Get the balance factor

int balance = getBalance(node);

// 4. Balance the tree if needed

// 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;

}

// Search for a key

AVLNode search(AVLNode root, int key) {

// Base case: root is null or key is present at root

if (root == null || root.key == key)

return root;

// Key is greater than root's key

if (key > root.key)

return search(root.right, key);

// Key is smaller than root's key

return search(root.left, key);

}

// Wrapper method for insertion

void insert(int key) {

root = insert(root, key);

}

// Wrapper method for search

boolean search(int key) {

return search(root, key) != null;

}

// Inorder traversal

void inorder(AVLNode node) {

if (node != null) {

inorder(node.left);

System.*out*.print(node.key + " ");

inorder(node.right);

}

}

// Print inorder traversal

void printInorder() {

inorder(root);

System.*out*.println();

}

public static void main(String[] args) {

AVLTree tree = new AVLTree();

// Insert some keys

tree.insert(10);

tree.insert(20);

tree.insert(30);

tree.insert(40);

tree.insert(50);

tree.insert(25);

System.*out*.println("Inorder traversal of the AVL tree:");

tree.printInorder();

// Search for keys

int keyToSearch = 30;

if (tree.search(keyToSearch)) {

System.*out*.println("Key " + keyToSearch + " found in the tree");

} else {

System.*out*.println("Key " + keyToSearch + " not found in the tree");

}

keyToSearch = 60;

if (tree.search(keyToSearch)) {

System.*out*.println("Key " + keyToSearch + " found in the tree");

} else {

System.*out*.println("Key " + keyToSearch + " not found in the tree");

}

}

}

Output:

In order traversal of the AVL tree:

10 20 25 30 40 50

Key 30 found in the tree

Key 60 not found in the tree

**Task 3:**

Write algo for Read Black tree insertion

**Algorithm for Red-Black Tree Insertion:**

Algorithm: RedBlackInsert(T, z)

Input: Tree T, Node z to insert

Output: Modified Red-Black Tree

1. // Normal BST insertion

y = NIL

x = T.root

2. while x ≠ NIL do

y = x

if z.key < x.key then

x = x.left

else

x = x.right

3. z.parent = y

if y = NIL then

T.root = z // Tree was empty

else if z.key < y.key then

y.left = z

else

y.right = z

4. z.left = NIL

z.right = NIL

z.color = RED // New node is always red initially

5. RB-Insert-Fixup(T, z) // Fix Red-Black properties

Algorithm: RB-Insert-Fixup(T, z)

1. while z.parent.color = RED do

if z.parent = z.parent.parent.left then

y = z.parent.parent.right // Uncle node

if y.color = RED then // Case 1: Uncle is red

z.parent.color = BLACK

y.color = BLACK

z.parent.parent.color = RED

z = z.parent.parent

else

if z = z.parent.right then // Case 2: Uncle black, z is right child

z = z.parent

Left-Rotate(T, z)

z.parent.color = BLACK // Case 3: Uncle black, z is left child

z.parent.parent.color = RED

Right-Rotate(T, z.parent.parent)

else

// Symmetric cases (right subtree)

y = z.parent.parent.left

if y.color = RED then

z.parent.color = BLACK

y.color = BLACK

z.parent.parent.color = RED

z = z.parent.parent

else

if z = z.parent.left then

z = z.parent

Right-Rotate(T, z)

z.parent.color = BLACK

z.parent.parent.color = RED

Left-Rotate(T, z.parent.parent)

2. T.root.color = BLACK // Root is always black

**Task 4:**

Wap to insert an element in red black tree

**Program to Insert an Element in Red-Black Tree:**

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.color = "RED" # New nodes are red by default

self.parent = None

self.left = None

self.right = None

class RedBlackTree:

def \_\_init\_\_(self):

self.NIL = Node(None)

self.NIL.color = "BLACK"

self.root = self.NIL

def insert(self, data):

new\_node = Node(data)

new\_node.left = self.NIL

new\_node.right = self.NIL

# Standard BST insertion

parent = None

current = self.root

while current != self.NIL:

parent = current

if new\_node.data < current.data:

current = current.left

else:

current = current.right

new\_node.parent = parent

if parent is None:

self.root = new\_node

elif new\_node.data < parent.data:

parent.left = new\_node

else:

parent.right = new\_node

# Fix Red-Black Tree properties

self.fix\_insert(new\_node)

def fix\_insert(self, node):

while node.parent and node.parent.color == "RED":

if node.parent == node.parent.parent.left:

uncle = node.parent.parent.right

# Case 1: Uncle is red

if uncle.color == "RED":

node.parent.color = "BLACK"

uncle.color = "BLACK"

node.parent.parent.color = "RED"

node = node.parent.parent

else:

# Case 2: Uncle is black, node is right child

if node == node.parent.right:

node = node.parent

self.left\_rotate(node)

# Case 3: Uncle is black, node is left child

node.parent.color = "BLACK"

node.parent.parent.color = "RED"

self.right\_rotate(node.parent.parent)

else:

uncle = node.parent.parent.left

# Case 1: Uncle is red

if uncle.color == "RED":

node.parent.color = "BLACK"

uncle.color = "BLACK"

node.parent.parent.color = "RED"

node = node.parent.parent

else:

# Case 2: Uncle is black, node is left child

if node == node.parent.left:

node = node.parent

self.right\_rotate(node)

# Case 3: Uncle is black, node is right child

node.parent.color = "BLACK"

node.parent.parent.color = "RED"

self.left\_rotate(node.parent.parent)

self.root.color = "BLACK"

def left\_rotate(self, x):

y = x.right

x.right = y.left

if y.left != self.NIL:

y.left.parent = x

y.parent = x.parent

if x.parent is None:

self.root = y

elif x == x.parent.left:

x.parent.left = y

else:

x.parent.right = y

y.left = x

x.parent = y

def right\_rotate(self, y):

x = y.left

y.left = x.right

if x.right != self.NIL:

x.right.parent = y

x.parent = y.parent

if y.parent is None:

self.root = x

elif y == y.parent.right:

y.parent.right = x

else:

y.parent.left = x

x.right = y

y.parent = x

def inorder(self, node):

if node != self.NIL:

self.inorder(node.left)

print(f"{node.data}({node.color})", end=" ")

self.inorder(node.right)

def display(self):

print("Inorder traversal:")

self.inorder(self.root)

print()

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

rb\_tree = RedBlackTree()

# Insert elements

elements = [10, 20, 30, 15, 25, 5, 1]

print("Inserting elements:", elements)

for element in elements:

print(f"Inserting {element}")

rb\_tree.insert(element)

rb\_tree.display()

print()

**Output:**

Inserting elements: [10, 20, 30, 15, 25, 5, 1]

Inserting 10

Inorder traversal:

10(BLACK)

Inserting 20

Inorder traversal:

10(BLACK) 20(RED)

Inserting 30

Inorder traversal:

10(RED) 20(BLACK) 30(RED)

Inserting 15

Inorder traversal:

10(RED) 15(RED) 20(BLACK) 30(RED)

Inserting 25

Inorder traversal:

10(RED) 15(RED) 20(BLACK) 25(RED) 30(RED)