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".

**task 1:**

**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.

class Node {

int key, height;

Node left, right;

Node (int d) {

key = d;

height = 1;

}

}

public class task001 {

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) {

Node y = x.right;

Node T2 = y.left;

y.left = x;

x.right = 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[]) {

task001 tree = new task001();

tree.root = tree.insert(tree.root, 20);

tree.root = tree.insert(tree.root, 10);

tree.root = tree.insert(tree.root, 30);

tree.root = tree.insert(tree.root, 40);

tree.root = tree.insert(tree.root, 60);

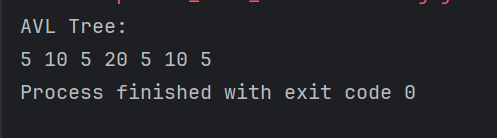
tree.root = tree.insert(tree.root, 5);

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

tree.printTree(tree.root);

}

}



**Task 3:**

**Write algo for Read Black tree insertion**

1. **Create a new node N** with the value to insert.
2. Set the **color of N as RED**.
3. **Insert N using Binary Search Tree (BST)** insertion rules.
4. If the inserted node is the **root**, set its color to **BLACK** and finish.

### **🔄 Fix Violations (if N has a RED parent):**

1. While N is not the root **and** its parent P is **RED**:  
    6. Let G = grandparent of N.  
    7. Let U = uncle of N (sibling of P).

### **🟥 Case 1: Uncle U is RED**

1. Set P.color = BLACK.
2. Set U.color = BLACK.
3. Set G.color = RED.
4. Move N up to G and continue loop.

### **⚫ Case 2: Uncle U is BLACK or NULL**

#### **Case 2a: Left-Left (N is left child of left parent)**

1. Perform **Right Rotation** at G.
2. Swap colors of P and G.

#### **Case 2b: Right-Right (N is right child of right parent)**

1. Perform **Left Rotation** at G.
2. Swap colors of P and G.

#### **Case 2c: Left-Right (N is right child of left parent)**

1. Perform **Left Rotation** at P.
2. Set N = P, then follow **Left-Left case**.

#### **Case 2d: Right-Left (N is left child of right parent)**

1. Perform **Right Rotation** at P.
2. Set N = P, then follow **Right-Right case**.

**✅ Final Step:**

1. Set the **root color to BLACK** to ensure tree property is maintained.

Task 4:

Wap to insert an element in red black tree

// Task 4: WAP to insert an element in Red-Black Tree

enum Color {

*RED*, *BLACK*

}

class Node2 {

int data;

Color color;

Node2 left, right, parent;

public Node2(int data) {

this.data = data;

this.color = Color.*RED*; // New nodes are red initially

this.left = this.right = this.parent = null;

}

}

public class RedBlackTreeInsert {

private Node2 root;

// Public insert method

public void insert(int data) {

Node2 newNode = new Node2(data);

root = bstInsert(root, newNode);

fixViolation(newNode);

}

// BST insert

private Node2 bstInsert(Node2 root, Node2 node) {

if (root == null)

return node;

if (node.data < root.data) {

root.left = bstInsert(root.left, node);

root.left.parent = root;

} else if (node.data > root.data) {

root.right = bstInsert(root.right, node);

root.right.parent = root;

}

return root;

}

// Fix Red-Black Tree violations

private void fixViolation(Node2 node) {

Node2 parent = null;

Node2 grandparent = null;

while (node != root && node.parent.color == Color.*RED*) {

parent = node.parent;

grandparent = parent.parent;

if (parent == grandparent.left) {

Node2 uncle = grandparent.right;

if (uncle != null && uncle.color == Color.*RED*) {

// Case 1 - Recoloring

parent.color = Color.*BLACK*;

uncle.color = Color.*BLACK*;

grandparent.color = Color.*RED*;

node = grandparent;

} else {

// Case 2 - Left-Right or Left-Left

if (node == parent.right) {

node = parent;

leftRotate(node);

}

// Left-Left case

parent.color = Color.*BLACK*;

grandparent.color = Color.*RED*;

rightRotate(grandparent);

}

} else {

Node2 uncle = grandparent.left;

if (uncle != null && uncle.color == Color.*RED*) {

// Case 1 - Recoloring

parent.color = Color.*BLACK*;

uncle.color = Color.*BLACK*;

grandparent.color = Color.*RED*;

node = grandparent;

} else {

// Case 2 - Right-Left or Right-Right

if (node == parent.left) {

node = parent;

rightRotate(node);

}

// Right-Right case

parent.color = Color.*BLACK*;

grandparent.color = Color.*RED*;

leftRotate(grandparent);

}

}

}

root.color = Color.*BLACK*; // Ensure root is always black

}

// Left rotate

private void leftRotate(Node2 x) {

Node2 y = x.right;

x.right = y.left;

if (y.left != null)

y.left.parent = x;

y.parent = x.parent;

if (x.parent == null)

root = y;

else if (x == x.parent.left)

x.parent.left = y;

else

x.parent.right = y;

y.left = x;

x.parent = y;

}

// Right rotate

private void rightRotate(Node2 y) {

Node2 x = y.left;

y.left = x.right;

if (x.right != null)

x.right.parent = y;

x.parent = y.parent;

if (y.parent == null)

root = x;

else if (y == y.parent.left)

y.parent.left = x;

else

y.parent.right = x;

x.right = y;

y.parent = x;

}

// In-order traversal for testing

public void inorder() {

System.*out*.print("In-order traversal with color: ");

inorderHelper(root);

System.*out*.println();

}

private void inorderHelper(Node2 root) {

if (root != null) {

inorderHelper(root.left);

System.*out*.print(root.data + "(" + root.color + ") ");

inorderHelper(root.right);

}

}

// Main method for testing

public static void main(String[] args) {

RedBlackTreeInsert tree = new RedBlackTreeInsert();

// Example insertion

int[] values = {10, 20, 30, 15, 25, 5};

for (int val : values) {

System.*out*.println("Inserting: " + val);

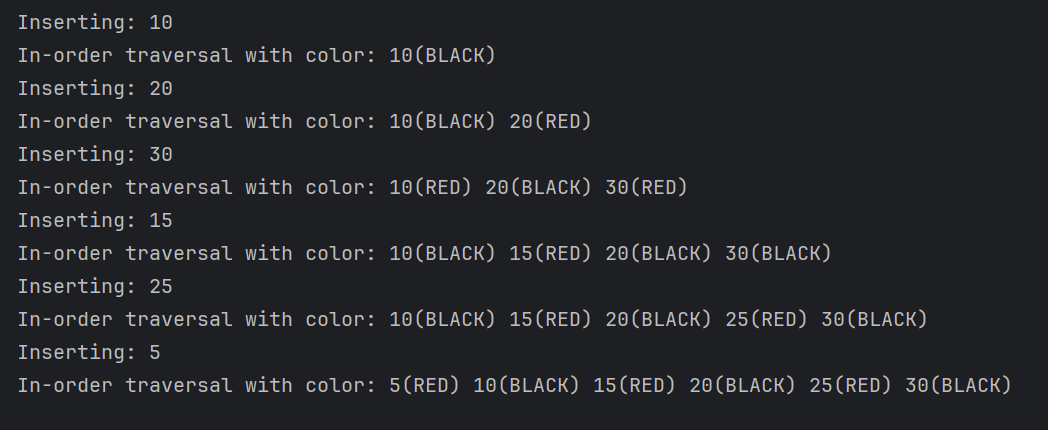
tree.insert(val);

tree.inorder();

}

}

}



**Insert an Element - Red Black Tree −**

1. Check tree is empty. If empty, then insert new node - color Black. (Because Root Node - Black in color)

2. else if Tree - not empty then insert new node as leaf node to the end and color - Red.

3. If parent of new node is Red and its neighbours(parent’s) node is also Red,

then Flip the color of the both neighbour and Parent and Grandparents (If it is not Root Node Otherwise Flip the color of the Parent and neighbour only) i.e., Black.

4. If parent of new node is Red and its neighbours(parent’s) node is empty or NULL,

then Rotate (either Left-Left or Left-Right rotation) the new node and parent.

5. we have two types of rotation

- Left Left Rotation and

- Left Right Rotation.

6. we apply Rotation in some conditions only.

The conditions are −

- If parent of new node is Red and neighbour node is empty or NULL, then rotate left or right rotation.

- In Left-Left Rotation flip the color of the parent and grandparent.

Make the parent as Grandparent and grandparent as child

====================================================================

Quiz qns

technique improves the binary search tree balance here?

class AVLTreeNode {

int val, height;

AVLTreeNode left, right;

AVLTreeNode(int v) {

val = v;

height = 1;

}

}

public class AVLTree {

public int getHeight(AVLTreeNode n) {

return (n == null) ? 0 : n.height;

}

public int getBalance(AVLTreeNode n) {

return (n == null) ? 0 : getHeight(n.left) - getHeight(n.right);

}

}

Qn:

What is the reason for stack over flow

class RecursiveLoop {

public int calculate(int n) {

if (n == 0) return 0;

return n + calculate(n);

}

public static void main(String[] args) {

System.out.println(new RecursiveLoop().calculate(5));

}

}

Qn

===========================================================================================================

AVL qn:

Why are AVL trees considered balanced, and how does this impact performance?

1. They rearrange nodes to ensure maximum depth for fast access.
2. They allow unbalanced growth in left subtrees for quick insertion.
3. They maintain a balance factor (height difference) of at most 1 between subtrees to ensure O(log n) operations.
4. They replicate nodes for redundancy, enabling constant-time deletion.

What is the maximum height of an AVL tree with p nodes?  
a) p  
b) log(p)  
c) log(p)/2  
d) *p*⁄*2*

What maximum difference in heights between the leafs of a AVL tree is possible?  
a) log(n) where n is the number of nodes  
b) n where n is the number of nodes  
c) 0 or 1  
d) atmost 1 (as avl is self balanced)

Why to prefer red-black trees over AVL trees?  
a) Because red-black is more rigidly balanced  
b) AVL tree store balance factor in every node which costs space  
c) AVL tree fails at scale  
d) Red black is more efficient

in binary search tree balance what is the technique used?

class AVLTreeNode {

int val, height;

AVLTreeNode left, right;

AVLTreeNode(int v) {

val = v;

height = 1;

}

}

public class AVLTree {

public int getHeight(AVLTreeNode n) {

return (n == null) ? 0 : n.height;

}

public int getBalance(AVLTreeNode n) {

return (n == null) ? 0 : getHeight(n.left) - getHeight(n.right);

}

}

| 1. Using balance factor to determine level-order traversal | 1. Adding random nodes at different depths to flatten the tree | 1. Checking balance factor to perform rotations when needed | 1. Reversing subtree links on imbalance |
| --- | --- | --- | --- |

DSA quiz 1:

| property of a priority queue differentiates it most from a regular queue implementation | It maintains a strict hierarchical structure using a self-balancing BST to enforce priority. | It allows insertion and removal only from one end, similar to a stack. | Elements are dequeued based on their priority, not their insertion order, often implemented using a binary heap. | Elements are removed based on their order of insertion rather than priority. |
| --- | --- | --- | --- | --- |

Task 3:

Insert an Element - Red Black Tree −

1. Check tree is empty. If empty, then insert new node - color Black. (Because Root Node - Black in color)

2. else if Tree - not empty then insert new node as leaf node to the end and color - Red.

3. If parent of new node is Red and its neighbours(parent’s) node is also Red,

then Flip the color of the both neighbour and Parent and Grandparents (If it is not Root Node Otherwise Flip the color of the Parent and neighbour only) i.e., Black.

4. If parent of new node is Red and its neighbours(parent’s) node is empty or NULL,

then Rotate (either Left-Left or Left-Right rotation) the new node and parent.

5. we have two types of rotation

- Left Left Rotation and

- Left Right Rotation.

6. we apply Rotation in some conditions only.

The conditions are −

- If parent of new node is Red and neighbour node is empty or NULL, then rotate left or right rotation.

- In Left-Left Rotation flip the color of the parent and grandparent.

Make the parent as Grandparent and grandparent as child

tell the main advantage of using a doubly linked list as shown here?

class Node {

int data;

Node prev, next;

Node(int data) {

this.data = data;

}

}

public class DoublyList {

Node head, tail;

public void insertEnd(int data) {

Node newNode = new Node(data);

if (tail == null) {

head = tail = newNode;

} else {

tail.next = newNode;

newNode.prev = tail;

tail = newNode;

}

}

}

| has simpler node deletion than singly list | allows forward traversal only using one pointer | avoids null pointers at boundaries | uses more space but enables two-way traversal |
| --- | --- | --- | --- |

output of the reverse string function and its complexity is

public class ReverseString {

public String reverse(String input) {

char[] chars = input.toCharArray();

int left = 0, right = chars.length - 1;

while (left < right) {

char temp = chars[left];

chars[left] = chars[right];

chars[right] = temp;

left++;

right--;

}

return new String(chars);

}

public static void main(String[] args) {

ReverseString rs = new ReverseString();

System.out.println(rs.reverse("java"));

}

}

| Output: avaj, Time: O(n) due to single pass swap from ends | Output: vaaj, Time: O(n) with off-by-one index swap | Output: java, Time: O(n^2) due to nested loop |
| --- | --- | --- |

| primary purpose of using recursion in solving problems like tree traversal or factorial calculation is? | Recursion allows breaking a large problem into smaller sub-problems by calling the same function repeatedly with modified input. | Recursion introduces randomness which helps simulate complex decisions in algorithms. | Recursion eliminates the need for any stack or memory during runtime. | Recursion forces the problem to run in parallel threads, increasing computation speed. |
| --- | --- | --- | --- | --- |

| merge sort considered a stable sorting algorithm reason? | requires fewer recursive calls than unstable sorts like quicksort. | maintains the relative order of equal elements during the sorting process. | reduces time complexity by ignoring duplicate values. | rearranges all elements randomly to achieve faster execution. |
| --- | --- | --- | --- | --- |

| comparing hash tables and arrays, what is a significant functional difference regarding key access | Arrays dynamically resize to avoid collisions, while hash tables use fixed buckets. | Arrays support hashed key access, while hash tables require index-based retrieval. | Hash tables support binary searching, but arrays do not. | Hash tables map arbitrary keys to values using a hash function, whereas arrays only support integer index-based access. |
| --- | --- | --- | --- | --- |

| technique is commonly used to reduce repeated recursive calls in dynamic programming problems like Fibonacci calculation | Employing memoization to store and reuse previously computed results. | Applying nested recursion to simplify base case evaluation. | Transforming recursion into stack-based iteration for space optimization. | Using a binary search strategy to narrow down recursive branches. |
| --- | --- | --- | --- | --- |

| What is good hash function in a hash table implementation? | distribute input keys uniformly across the hash table to minimize clustering and reduce collision probability. | be complex enough to prevent reverse-engineering of keys. | produce sequential hash codes for predictable indexing and faster traversal. | generate a unique hash code for every possible key to avoid hash collisions completely. |
| --- | --- | --- | --- | --- |

dynamic programming optimize the Fibonacci computation. Explain

public class DPFibonacci {

public int fib(int n) {

if (n == 0) return 0;

if (n == 1) return 1;

int[] dp = new int[n + 1];

dp[0] = 0;

dp[1] = 1;

for (int i = 2; i <= n; i++) {

dp[i] = dp[i - 1] + dp[i - 2];

}

return dp[n];

}

}

| computes each Fibonacci number independently | reduces time from O(2^n) to O(n) by caching | uses recursion for faster computation | increases space complexity for no gain |
| --- | --- | --- | --- |

which sort and time complexity is applied in the below insertion sort code

public class Insertiont {

public void sort(int[] arr) {

for (int i = 1; i < arr.length; i++) {

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

}

| Stable sort, w  orst-case O(n^2) | Stable sort, O(n log n) in all cases | Unstable sort with O(n^2) time for best case  14 correct out of 20 | Unstable sort with average O(n) time |
| --- | --- | --- | --- |