**DSA Project – 1**

**Binary Search Tree (BST) and Time Complexity**

A **Binary Search Tree (BST)** is a hierarchical data structure in which each node has at most two children, commonly referred to as the left and right child. The left subtree contains nodes with values less than the parent, while the right subtree contains values greater than the parent.

**Time Complexity of BST Operations:**

* **Search:** O(h)O(h)O(h), where hhh is the height of the tree. In a balanced BST, h=O(log⁡n)h = O(\log n)h=O(logn), but in a skewed tree, it can go up to O(n)O(n)O(n).
* **Insertion:** O(h)O(h)O(h)
* **Deletion:** O(h)O(h)O(h)

**Need for Self-Balancing Trees**

A standard BST may become skewed, resulting in degraded performance (i.e., O(n)O(n)O(n) instead of O(log⁡n)O(\log n)O(logn)). **Self-balancing trees** ensure that the height remains approximately O(log⁡n)O(\log n)O(logn), providing better efficiency for operations.

**AVL Trees: Introduction and Detection of Imbalance**

An **AVL Tree** is a self-balancing BST where the height difference (balance factor) between left and right subtrees of any node is at most 1. The **balance factor** of a node is calculated as:

Balance Factor=Height of Left Subtree−Height of Right Subtree\text{Balance Factor} = \text{Height of Left Subtree} - \text{Height of Right Subtree}Balance Factor=Height of Left Subtree−Height of Right Subtree

An AVL tree becomes unbalanced when the balance factor goes beyond −1,0,1-1, 0, 1−1,0,1.

**Sub-Cases of Imbalance in AVL Trees**

1. **LL (Left-Left) Rotation**: Occurs when nodes are inserted in strictly increasing order in the left subtree.
2. **LR (Left-Right) Rotation**: Occurs when a node is inserted into the right subtree of the left child.
3. **RL (Right-Left) Rotation**: Happens when a node is inserted into the left subtree of the right child.
4. **RR (Right-Right) Rotation**: Happens when nodes are inserted in strictly increasing order in the right subtree.

**How to Balance Each Case**

* **LL Rotation (Right Rotation):** Perform a **single right rotation**.
* **RR Rotation (Left Rotation):** Perform a **single left rotation**.
* **LR Rotation (Left-Right Rotation):** Perform a **left rotation** on the left child, followed by a **right rotation** on the node.
* **RL Rotation (Right-Left Rotation):** Perform a **right rotation** on the right child, followed by a **left rotation** on the node.

**AVL Trees vs. Red-Black Trees: When to Use Which?**

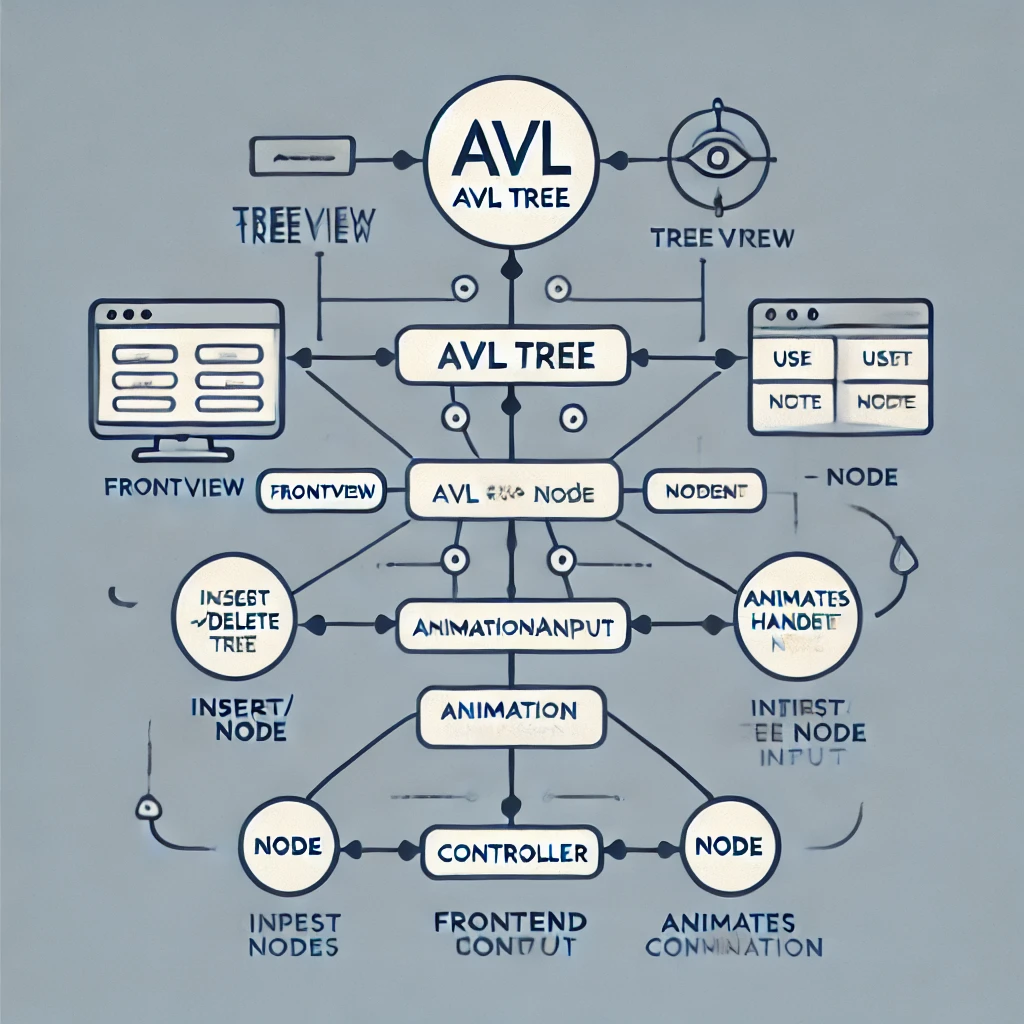
| **Feature** | **AVL Tree** | **Red-Black Tree** |
| --- | --- | --- |
| **Balance** | Strictly balanced | Loosely balanced |
| **Search Complexity** | O(log⁡n)O(\log n)O(logn) (better for lookups) | O(log⁡n)O(\log n)O(logn) (but slightly slower) |
| **Insertion Complexity** | More rotations (costlier) | Fewer rotations (efficient for insertions) |
| **Deletion Complexity** | Costlier compared to Red-Black Tree | Faster due to fewer rotations |
| **Use Case** | Best for search-intensive applications | Best for frequent insertions/deletions |

**When to use:**

* **AVL Trees** are preferred when searches are more frequent.
* **Red-Black Trees** are better when insertions and deletions are more frequent.

**UML diagram for Web UI project showing AVL tree visualization.**

* **Frontend Components:**
  + **Tree Visualization** (Displays AVL tree structure)
  + **User Input Panel** (Allows node insertion/deletion)
  + **Balance Factor Display** (Shows imbalance in nodes)
  + **Animation Controller** (Handles rotation animations)
* **Backend Components:**
  + **AVL Tree Class** (Handles tree operations)
  + **Node Class** (Defines individual tree nodes)
  + **Controller** (Manages interactions between frontend and backend)



**Code**

import java.util.\*;

import java.io.PrintStream;

import java.util.function.Function;

class BSTNode{

int data;

int h;

BSTNode left, right;

public BSTNode(int data){

this.data = data; this.left = this.right = null;

this.h = 1;

}

}

class Solution {

void inOrder(BSTNode root){

if(root == null) return;

inOrder(root.left); System.out.print(root.data + " "); inOrder(root.right);

}

BSTNode insert(BSTNode root, int x){

if(root == null) return new BSTNode(x);

if(x < root.data)

root.left = insert(root.left, x);

else

root.right = insert(root.right, x);

updateHeight(root);

if(getBF(root) == +2){ // LEFT Heavy

if(getBF(root.left) == +1){ // LL

root = rightRotation(root);

} else { // LR

root.left = leftRotation(root.left);

root = rightRotation(root);

}

} else if(getBF(root) == -2) { // RIGHT Heavy

if(getBF(root.right) == -1){ // RR

root = leftRotation(root);

} else { // RL

root.right = rightRotation(root.right);

root = leftRotation(root);

}

}

return root;

}

int getBF(BSTNode root){

return (root.left!=null ? root.left.h : 0) -

(root.right==null ? 0 : root.right.h);

}

void updateHeight(BSTNode root){

root.h = 1 + Math.max(root.left!=null ? root.left.h : 0,

root.right==null ? 0 : root.right.h);

}

BSTNode leftRotation(BSTNode root){

BSTNode nr = root.right;

BSTNode t2 = root.right.left;

root.right = t2;

updateHeight(root);

nr.left = root;

updateHeight(nr);

return nr;

}

BSTNode rightRotation(BSTNode root){

BSTNode nr = root.left;

BSTNode t2 = root.left.right;

root.left = t2;

updateHeight(root);

nr.right = root;

updateHeight(nr);

return nr;

}

}

class Main {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

Solution sol = new Solution();

TreePrinter<BSTNode> tp = new TreePrinter<>(n->n.data+"", n->n.left, n->n.right);

BSTNode root = null; // Empty tree

while(sc.hasNextInt()){

int x = sc.nextInt();

root = sol.insert(root, x);

tp.printTree(root);

}

}

}

//////////////////////////////////// TREE PRINTER /////////////////

class TreePrinter<T> {

private Function<T, String> getLabel;

private Function<T, T> getLeft;

private Function<T, T> getRight;

private PrintStream outStream = System.out;

private boolean squareBranches = false;

private boolean lrAgnostic = false;

private int hspace = 2;

private int tspace = 1;

public TreePrinter(Function<T, String> getLabel, Function<T, T> getLeft, Function<T, T> getRight) {

this.getLabel = getLabel;

this.getLeft = getLeft;

this.getRight = getRight;

}

public void setPrintStream(PrintStream outStream) {

this.outStream = outStream;

}

public void setSquareBranches(boolean squareBranches) { this.squareBranches = squareBranches; }

public void setLrAgnostic(boolean lrAgnostic) { this.lrAgnostic = lrAgnostic; }

public void setHspace(int hspace) { this.hspace = hspace; }

public void setTspace(int tspace) { this.hspace = tspace; }

/\*

Prints ascii representation of binary tree.

Parameter hspace is minimum number of spaces between adjacent node labels.

Parameter squareBranches, when set to true, results in branches being printed with ASCII box

drawing characters.

\*/

public void printTree(T root) {

List<TreeLine> treeLines = buildTreeLines(root);

printTreeLines(treeLines);

}

/\*

Prints ascii representations of multiple trees across page.

Parameter hspace is minimum number of spaces between adjacent node labels in a tree.

Parameter tspace is horizontal distance between trees, as well as number of blank lines

between rows of trees.

Parameter lineWidth is maximum width of output

Parameter squareBranches, when set to true, results in branches being printed with ASCII box

drawing characters.

\*/

public void printTrees(List<T> trees, int lineWidth) {

List<List<TreeLine>> allTreeLines = new ArrayList<>();

int[] treeWidths = new int[trees.size()];

int[] minLeftOffsets = new int[trees.size()];

int[] maxRightOffsets = new int[trees.size()];

for (int i = 0; i < trees.size(); i++) {

T treeNode = trees.get(i);

List<TreeLine> treeLines = buildTreeLines(treeNode);

allTreeLines.add(treeLines);

minLeftOffsets[i] = minLeftOffset(treeLines);

maxRightOffsets[i] = maxRightOffset(treeLines);

treeWidths[i] = maxRightOffsets[i] - minLeftOffsets[i] + 1;

}

int nextTreeIndex = 0;

while (nextTreeIndex < trees.size()) {

// print a row of trees starting at nextTreeIndex

// first figure range of trees we can print for next row

int sumOfWidths = treeWidths[nextTreeIndex];

int endTreeIndex = nextTreeIndex + 1;

while (endTreeIndex < trees.size() && sumOfWidths + tspace + treeWidths[endTreeIndex] < lineWidth) {

sumOfWidths += (tspace + treeWidths[endTreeIndex]);

endTreeIndex++;

}

endTreeIndex--;

// find max number of lines for tallest tree

int maxLines = allTreeLines.stream().mapToInt(list -> list.size()).max().orElse(0);

// print trees line by line

for (int i = 0; i < maxLines; i++) {

for (int j = nextTreeIndex; j <= endTreeIndex; j++) {

List<TreeLine> treeLines = allTreeLines.get(j);

if (i >= treeLines.size()) {

System.out.print(spaces(treeWidths[j]));

} else {

int leftSpaces = -(minLeftOffsets[j] - treeLines.get(i).leftOffset);

int rightSpaces = maxRightOffsets[j] - treeLines.get(i).rightOffset;

System.out.print(spaces(leftSpaces) + treeLines.get(i).line + spaces(rightSpaces));

}

if (j < endTreeIndex) System.out.print(spaces(tspace));

}

System.out.println();

}

for (int i = 0; i < tspace; i++) {

System.out.println();

}

nextTreeIndex = endTreeIndex + 1;

}

}

private void printTreeLines(List<TreeLine> treeLines) {

if (treeLines.size() > 0) {

int minLeftOffset = minLeftOffset(treeLines);

int maxRightOffset = maxRightOffset(treeLines);

for (TreeLine treeLine : treeLines) {

int leftSpaces = -(minLeftOffset - treeLine.leftOffset);

int rightSpaces = maxRightOffset - treeLine.rightOffset;

outStream.println(spaces(leftSpaces) + treeLine.line + spaces(rightSpaces));

}

}

}

private List<TreeLine> buildTreeLines(T root) {

if (root == null) return Collections.emptyList();

else {

String rootLabel = getLabel.apply(root);

List<TreeLine> leftTreeLines = buildTreeLines(getLeft.apply(root));

List<TreeLine> rightTreeLines = buildTreeLines(getRight.apply(root));

int leftCount = leftTreeLines.size();

int rightCount = rightTreeLines.size();

int minCount = Math.min(leftCount, rightCount);

int maxCount = Math.max(leftCount, rightCount);

// The left and right subtree print representations have jagged edges, and we essentially we have to

// figure out how close together we can bring the left and right roots so that the edges just meet on

// some line. Then we add hspace, and round up to next odd number.

int maxRootSpacing = 0;

for (int i = 0; i < minCount; i++) {

int spacing = leftTreeLines.get(i).rightOffset - rightTreeLines.get(i).leftOffset;

if (spacing > maxRootSpacing) maxRootSpacing = spacing;

}

int rootSpacing = maxRootSpacing + hspace;

if (rootSpacing % 2 == 0) rootSpacing++;

// rootSpacing is now the number of spaces between the roots of the two subtrees

List<TreeLine> allTreeLines = new ArrayList<>();

// strip ANSI escape codes to get length of rendered string. Fixes wrong padding when labels use ANSI escapes for colored nodes.

String renderedRootLabel = rootLabel.replaceAll("\\e\\[[\\d;]\*[^\\d;]", "");

// add the root and the two branches leading to the subtrees

allTreeLines.add(new TreeLine(rootLabel, -(renderedRootLabel.length() - 1) / 2, renderedRootLabel.length() / 2));

// also calculate offset adjustments for left and right subtrees

int leftTreeAdjust = 0;

int rightTreeAdjust = 0;

if (leftTreeLines.isEmpty()) {

if (!rightTreeLines.isEmpty()) {

// there's a right subtree only

if (squareBranches) {

if (lrAgnostic) {

allTreeLines.add(new TreeLine("\u2502", 0, 0));

} else {

allTreeLines.add(new TreeLine("\u2514\u2510", 0, 1));

rightTreeAdjust = 1;

}

} else {

allTreeLines.add(new TreeLine("\\", 1, 1));

rightTreeAdjust = 2;

}

}

} else if (rightTreeLines.isEmpty()) {

// there's a left subtree only

if (squareBranches) {

if (lrAgnostic) {

allTreeLines.add(new TreeLine("\u2502", 0, 0));

} else {

allTreeLines.add(new TreeLine("\u250C\u2518", -1, 0));

leftTreeAdjust = -1;

}

} else {

allTreeLines.add(new TreeLine("/", -1, -1));

leftTreeAdjust = -2;

}

} else {

// there's a left and right subtree

if (squareBranches) {

int adjust = (rootSpacing / 2) + 1;

String horizontal = String.join("", Collections.nCopies(rootSpacing / 2, "\u2500"));

String branch = "\u250C" + horizontal + "\u2534" + horizontal + "\u2510";

allTreeLines.add(new TreeLine(branch, -adjust, adjust));

rightTreeAdjust = adjust;

leftTreeAdjust = -adjust;

} else {

if (rootSpacing == 1) {

allTreeLines.add(new TreeLine("/ \\", -1, 1));

rightTreeAdjust = 2;

leftTreeAdjust = -2;

} else {

for (int i = 1; i < rootSpacing; i += 2) {

String branches = "/" + spaces(i) + "\\";

allTreeLines.add(new TreeLine(branches, -((i + 1) / 2), (i + 1) / 2));

}

rightTreeAdjust = (rootSpacing / 2) + 1;

leftTreeAdjust = -((rootSpacing / 2) + 1);

}

}

}

// now add joined lines of subtrees, with appropriate number of separating spaces, and adjusting offsets

for (int i = 0; i < maxCount; i++) {

TreeLine leftLine, rightLine;

if (i >= leftTreeLines.size()) {

// nothing remaining on left subtree

rightLine = rightTreeLines.get(i);

rightLine.leftOffset += rightTreeAdjust;

rightLine.rightOffset += rightTreeAdjust;

allTreeLines.add(rightLine);

} else if (i >= rightTreeLines.size()) {

// nothing remaining on right subtree

leftLine = leftTreeLines.get(i);

leftLine.leftOffset += leftTreeAdjust;

leftLine.rightOffset += leftTreeAdjust;

allTreeLines.add(leftLine);

} else {

leftLine = leftTreeLines.get(i);

rightLine = rightTreeLines.get(i);

int adjustedRootSpacing = (rootSpacing == 1 ? (squareBranches ? 1 : 3) : rootSpacing);

TreeLine combined = new TreeLine(leftLine.line + spaces(adjustedRootSpacing - leftLine.rightOffset + rightLine.leftOffset) + rightLine.line,

leftLine.leftOffset + leftTreeAdjust, rightLine.rightOffset + rightTreeAdjust);

allTreeLines.add(combined);

}

}

return allTreeLines;

}

}

private static int minLeftOffset(List<TreeLine> treeLines) {

return treeLines.stream().mapToInt(l -> l.leftOffset).min().orElse(0);

}

private static int maxRightOffset(List<TreeLine> treeLines) {

return treeLines.stream().mapToInt(l -> l.rightOffset).max().orElse(0);

}

private static String spaces(int n) {

return String.join("", Collections.nCopies(n, " "));

}

private static class TreeLine {

String line;

int leftOffset;

int rightOffset;

TreeLine(String line, int leftOffset, int rightOffset) {

this.line = line;

this.leftOffset = leftOffset;

this.rightOffset = rightOffset;

}

}

}

**Input:**

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

**Output:**

1

1

\

2

2

/ \

1 3

2

/ \

1 3

\

4

2

/ \

1 4

/ \

3 5

4

/ \

2 5

/ \ \

1 3 6

4

/ \

/ \

/ \

2 6

/ \ / \

1 3 5 7

4

/ \

/ \

/ \

2 6

/ \ / \

1 3 5 7

\

8

4

/ \

/ \

/ \

2 6

/ \ / \

1 3 5 8

/ \

7 9

4

/ \

/ \

/ \

2 8

/ \ / \

1 3 6 9

/ \ \

5 7 10

4

/ \

/ \

/ \

2 8

/ \ / \

1 3 / \

/ \

6 10

/ \ / \

5 7 9 11

8

/ \

/ \

/ \

4 10

/ \ / \

/ \ 9 11

/ \ \

2 6 12

/ \ / \

1 3 5 7

8

/ \

/ \

/ \

4 10

/ \ / \

/ \ 9 12

/ \ / \

2 6 11 13

/ \ / \

1 3 5 7

8

/ \

/ \

/ \

/ \

/ \

4 12

/ \ / \

/ \ 10 13

/ \ / \ \

2 6 9 11 14

/ \ / \

1 3 5 7

8

/ \

/ \

/ \

/ \

/ \

/ \

/ \

4 12

/ \ / \

/ \ / \

/ \ / \

2 6 10 14

/ \ / \ / \ / \

1 3 5 7 9 11 13 15