1. Sort linked list using merge sort

class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

}

public ListNode(int key, ListNode head) {

this.val = key;

this.next = head;

}

}

public class MergeSortLinkedList {

public static ListNode mergeSort(ListNode head) {

if (head == null || head.next == null) {

return head;

}

ListNode middle = getMiddle(head);

System.out.println("Mid point "+middle.val);

ListNode nextOfMiddle = middle.next;

middle.next = null;

ListNode left = mergeSort(head);

ListNode right = mergeSort(nextOfMiddle);

return merge(left, right);

}

private static ListNode merge(ListNode left, ListNode right) {

ListNode merged = new ListNode(0);

ListNode current = merged;

while (left != null && right != null) {

if (left.val < right.val) {

current.next = left;

left = left.next;

} else {

current.next = right;

right = right.next;

}

current = current.next;

}

if (left != null) {

current.next = left;

}

if (right != null) {

current.next = right;

}

return merged.next;

}

private static ListNode getMiddle(ListNode head) {

if (head == null) {

return null;

}

ListNode slow = head;

ListNode fast = head;

while (fast.next != null && fast.next.next != null) {

slow = slow.next;

fast = fast.next.next;

}

return slow;

}

public static void printList(ListNode head) {

ListNode current = head;

while (current != null) {

System.out.print(current.val + " ");

current = current.next;

}

System.out.println();

}

public static void main(String[] args) {

int[] keys = { 8, 6, 4, 9, 3, 1 };

ListNode head = null;

for (int key: keys) {

head = new ListNode(key, head);

}

System.out.println("Original List:");

printList(head);

head = mergeSort(head);

System.out.println("Sorted List:");

printList(head);

}

}

2. Reverse Nodes in k-Group

Given the head of a linked list, reverse the nodes of the list k at a time, and return the modified list.

k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of k then left-out nodes, in the end, should remain as it is.

You may not alter the values in the list's nodes, only nodes themselves may be changed.

Example 1:

Input: head = [1,2,3,4,5], k = 2

Output: [2,1,4,3,5]

Example 2:

Input: head = [1,2,3,4,5], k = 3

Output: [3,2,1,4,5]

public static ListNode reverseKGroup(ListNode head, int k) {

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode prevGroupTail = dummy;

ListNode current = head;

ListNode groupPrev = null;

int length = getListLength(head);

while (length > k) {

ListNode groupStart = current;

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

ListNode nextNode = current.next;

current.next = groupPrev;

groupPrev = current;

current = nextNode;

}

prevGroupTail.next = groupPrev;

groupStart.next = current;

prevGroupTail = groupStart;

length -= k;

}

return dummy.next;

}

private static int getListLength(ListNode head) {

int length = 0;

while (head != null) {

length++;

head = head.next;

}

return length;

}

public static void printList(ListNode head) {

ListNode current = head;

while (current != null) {

System.out.print(current.val + " ");

current = current.next;

}

System.out.println();

}

public static ListNode reverse(ListNode head, ListNode headRef)

{

ListNode first, rest;

// empty list base case

if (head == null) {

return headRef;

}

first = head; // suppose first = {1, 2, 3}

rest = first.next; // rest = {2, 3}

// base case: the list has only one node

if (rest == null)

{

// fix the head pointer here

headRef = first;

return headRef;

}

// recursively reverse the smaller {2, 3} case

// after: rest = {3, 2}

headRef = reverse(rest, headRef);

// put the first item at the end of the list

rest.next = first;

first.next = null; // (tricky step — make a drawing)

return headRef;

}

public static void main(String[] args) {

int[] keys = { 8, 6, 4, 9, 3, 1 };

ListNode head = null;

for (int key: keys) {

head = new ListNode(key, head);

}

System.out.println("Original List:");

printList(head);

//head=reverse(head,head);// total reverse

head=reverseKGroup(head,2);

printList(head);

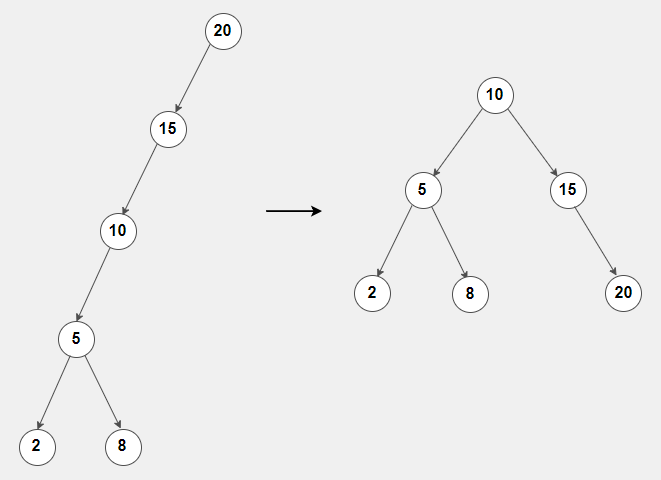
}

3. Construct a height-balanced BST from an unbalanced BST

Given a binary search tree (BST), convert it into a height-balanced binary search tree.

For a height-balanced binary search tree, the difference between the height of the left and right subtree of every node is never more than 1. The height of a binary search tree with n nodes is never more than log2(n) + 1.

For example, convert BST on the left into a BST on the right:



class TreeNode

{

int data;

TreeNode left, right;

// Constructor

TreeNode(int data) {

this.data = data;

}

}

public class BalancedHeightBST {

public static void preorder (TreeNode root)

{

if (root == null) {

return;

}

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

preorder(root.left);

preorder(root.right);

}

// Recursive function to push nodes of a given binary search tree into a

// list in an inorder fashion

public static void pushTreeNodes(TreeNode root, List<TreeNode> nodes)

{

// base case

if (root == null) {

return;

}

pushTreeNodes(root.left, nodes);

nodes.add(root);

pushTreeNodes(root.right, nodes);

}

// Recursive function to construct a height-balanced BST from

// given nodes in sorted order

public static TreeNode buildBalancedBST(List<TreeNode> nodes, int start, int end)

{

// base case

if (start > end) {

return null;

}

// find the middle index

int mid = (start + end) / 2;

// The root TreeNode will be a TreeNode present at the mid-index

TreeNode root = nodes.get(mid);

// recursively construct left and right subtree

root.left = buildBalancedBST(nodes, start, mid - 1);

root.right = buildBalancedBST(nodes, mid + 1, end);

// return root TreeNode

return root;

}

// Function to construct a height-balanced BST from an unbalanced BST

public static TreeNode constructBalancedBST(TreeNode root)

{

// Push nodes of a given binary search tree into a list in sorted order

List<TreeNode> nodes = new ArrayList<>();

pushTreeNodes(root, nodes);

// Construct a height-balanced BST from sorted BST nodes

return buildBalancedBST(nodes, 0, nodes.size() - 1);

}

public static void main(String[] args) {

TreeNode root = new TreeNode(20);

root.left = new TreeNode(15);

root.left.left = new TreeNode(10);

root.left.left.left = new TreeNode(5);

root.left.left.left.left = new TreeNode(2);

root.left.left.left.right = new TreeNode(8);

root = constructBalancedBST(root);

System.out.print("Preorder traversal of the constructed BST is ");

preorder(root);

}

}

4. Merge two BSTs into a doubly-linked list in sorted order

Given two binary search trees, merge them into a doubly-linked list in sorted order.

For example,

Input: Below BSTs

20

/ \

10 30

/ \

25 100

50

/ \

5 70

Output: Below DDL

5 —> 10 —> 20 —> 25 —> 30 —> 50 —> 70 —> 100 —> null

public static void printDoublyLinkedList(TreeNode head)

{

while (head != null)

{

System.out.print(head.data + " —> ");

head = head.right;

}

System.out.println("null");

}

// Function to insert a BST TreeNode at the front of a doubly linked list

public static TreeNode push(TreeNode root, TreeNode head)

{

// insert the given TreeNode at the front of a DDL

root.right = head;

// update the left child of the existing head TreeNode of the DDL

// to point to the BST TreeNode

if (head != null) {

head.left = root;

}

// update the head pointer of DDL

head = root;

return head;

}

// Recursive function to convert a BST into a doubly-linked list. It takes

// the BST's root TreeNode and the head TreeNode of the doubly linked list as an argument

public static TreeNode convertBSTtoDLL(TreeNode root, TreeNode head)

{

// Base case

if (root == null) {

return head;

}

// recursively convert the right subtree

head = convertBSTtoDLL(root.right, head);

// push the current TreeNode at the front of the doubly linked list

head = push(root, head);

// recursively convert the left subtree

head = convertBSTtoDLL(root.left, head);

return head;

}

// Recursive function to merge two doubly-linked lists into a

// single doubly linked list in sorted order

public static TreeNode mergeDDLs(TreeNode a, TreeNode b)

{

// if the first list is empty, return the second list

if (a == null) {

return b;

}

// if the second list is empty, return the first list

if (b == null) {

return a;

}

// if the head TreeNode of the first list is smaller

if (a.data < b.data)

{

a.right = mergeDDLs(a.right, b);

a.right.left = a;

return a;

}

// if the head TreeNode of the second list is smaller

else {

b.right = mergeDDLs(a, b.right);

b.right.left = b;

return b;

}

}

// Function to merge two binary search trees into a doubly-linked list

// in sorted order

public static TreeNode merge(TreeNode a, TreeNode b)

{

// convert the first binary search tree into a doubly-linked list

TreeNode first = null;

first = convertBSTtoDLL(a, first);

// convert the second binary search tree into a doubly-linked list

TreeNode second = null;

second = convertBSTtoDLL(b, second);

// merge both doubly-linked lists

return mergeDDLs(first, second);

}

public static void main(String[] args)

{

/\*

Construct the first BST

20

/ \

10 30

/ \

25 100

\*/

TreeNode a = new TreeNode(20);

a.left = new TreeNode(10);

a.right = new TreeNode(30);

a.right.left = new TreeNode(25);

a.right.right = new TreeNode(100);

/\*

Construct the second BST

50

/ \

5 70

\*/

TreeNode b = new TreeNode(50);

b.left = new TreeNode(5);

b.right = new TreeNode(70);

// merge both BSTs into a doubly-linked list

TreeNode root = merge(a, b);

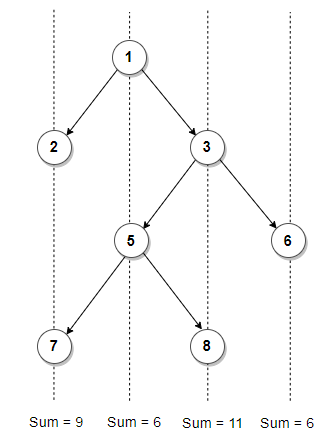
printDoublyLinkedList(root);

}

5. **Find the vertical sum of a binary tree**

Given a binary tree, the print vertical sum of it. Assume the left and right child of a node makes a 45–degree angle with the parent.

For example, the vertical sum is shown in the following binary tree:



public class VerticalSumNLogN {

public static void printVerticalSum(TreeNode root, int dist, Map<Integer, Integer> map)

{

// base case: empty tree

if (root == null) {

return;

}

// update the map

map.put(dist, map.getOrDefault(dist, 0) + root.data);

// recur for the left subtree by decreasing horizontal distance by 1

printVerticalSum(root.left, dist - 1, map);

// recur for the right subtree by increasing horizontal distance by 1

printVerticalSum(root.right, dist + 1, map);

}

// Function to print the vertical sum of a given binary tree

public static void printVerticalSum(TreeNode root)

{

// create an empty `TreeMap` where

// key —> relative horizontal distance of the TreeNode from the root TreeNode, and

// value —> sum of all nodes present at the same horizontal distance

Map<Integer, Integer> map = new TreeMap<>();

// perform preorder traversal on the tree and fill the map

printVerticalSum(root, 0, map);

// print vertical sum

System.out.println(map.values());

}

public static void main(String[] args)

{

/\* Construct the following tree

1

/ \

/ \

2 3

/ \

/ \

5 6

/ \

/ \

7 8

\*/

TreeNode root = new TreeNode(1);

root.left = new TreeNode(2);

root.right = new TreeNode(3);

root.right.left = new TreeNode(5);

root.right.right = new TreeNode(6);

root.right.left.left = new TreeNode(7);

root.right.left.right = new TreeNode(8);

printVerticalSum(root);

}

}

6. Swim in the raising water

You are given an n x n integer matrix grid where each value grid[i][j] represents the elevation at that point (i, j).

The rain starts to fall. At time t, the depth of the water everywhere is t. You can swim from a square to another 4-directionally adjacent square if and only if the elevation of both squares individually are at most t. You can swim infinite distances in zero time. Of course, you must stay within the boundaries of the grid during your swim.

Return the least time until you can reach the bottom right square (n - 1, n - 1) if you start at the top left square (0, 0)

Input: grid = [[0,2],[1,3]]

Output: 3

Input: grid = [[0,1,2,3,4],[24,23,22,21,5],[12,13,14,15,16],[11,17,18,19,20],[10,9,8,7,6]]

Output: 16

public int swimInWater(int[][] grid) {

int n = grid.length;

int left = 0, right = n \* n - 1;

while (left < right) {

int mid = left + (right - left) / 2;

if (canReachTarget(grid, mid)) {

right = mid;

} else {

left = mid + 1;

}

}

return left;

}

public boolean canReachTarget(int[][] grid, int time) {

int n = grid.length;

boolean[][] visited = new boolean[n][n];

return dfs(grid, 0, 0, time, visited);

}

public boolean dfs(int[][] grid, int x, int y, int time, boolean[][] visited) {

int n = grid.length;

if (x < 0 || x >= n || y < 0 || y >= n || visited[x][y] || grid[x][y] > time) {

return false;

}

if (x == n - 1 && y == n - 1) {

return true;

}

visited[x][y] = true;

return dfs(grid, x + 1, y, time, visited) ||

dfs(grid, x - 1, y, time, visited) ||

dfs(grid, x, y + 1, time, visited) ||

dfs(grid, x, y - 1, time, visited);

}

public static void main(String[] args) {

SwinRainWater solution = new SwinRainWater();

int[][] grid1 = {{0, 2}, {1, 3}};

System.out.println(solution.swimInWater(grid1)); // Output: 3

int[][] grid2 = {{0, 1, 2, 3, 4}, {24, 23, 22, 21, 5}, {12, 13, 14, 15, 16}, {11, 17, 18, 19, 20}, {10, 9, 8, 7, 6}};

System.out.println(solution.swimInWater(grid2)); // Output: 16

}

7. Serialize and Deserialize Binary Tree:

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

Serialization:

For serialization, you can perform a preorder traversal of the binary tree and append the values of each node to a string, using a delimiter to separate the values. You should also include a special marker to represent null nodes.

Algorithm:

Traverse the tree in preorder.

For each node, append its value to the string, followed by the delimiter.

If the node is null, append a special marker (e.g., "X") instead of the value, followed by the delimiter.

Deserialization:

For deserialization, you can split the serialized string using the delimiter to obtain the values of nodes in preorder traversal order. You can then recursively reconstruct the binary tree using these values and markers for null nodes.

Algorithm:

Split the serialized string using the delimiter.

Use a queue to keep track of the values.

Pop the first value from the queue.

If the value is the null marker, return None.

Create a new node with the popped value.

Recursively call the deserialization function for the left and right children of the node.

Return the node.

Serialized tree: 1,2,X,X,3,4,X,X,5,X,X

Deserialized tree: 1,2,X,X,3,4,X,X,5,X,X

1

/ \

2 3

/ \

4 5

1,2,X,X,3,4,X,X,5,X,X

The serialization follows a preorder traversal.

"1" is the root value.

"2" is the value of the left child of the root.

"X" represents a null node.

"3" is the value of the right child of the root.

"4" is the value of the left child of the right child.

"5" is the value of the right child of the right child.

1

/ \

2 3

/ \

4 5

public class BinaryTreeSerialization {

// Serialization

public String serialize(TreeNode root) {

if (root == null) {

return "X"; // Null marker

}

String left = serialize(root.left);

String right = serialize(root.right);

return root.val + "," + left + "," + right;

}

// Deserialization

public TreeNode deserialize(String data) {

Queue<String> queue = new LinkedList<>(Arrays.asList(data.split(",")));

return helper(queue);

}

private TreeNode helper(Queue<String> queue) {

String val = queue.poll();

if (val.equals("X")) {

return null;

}

TreeNode node = new TreeNode(Integer.parseInt(val));

node.left = helper(queue);

node.right = helper(queue);

return node;

}

public static void main(String[] args) {

// Example usage

TreeNode root = new TreeNode(1);

root.left = new TreeNode(2);

root.right = new TreeNode(3);

root.right.left = new TreeNode(4);

root.right.right = new TreeNode(5);

BinaryTreeSerialization serializer = new BinaryTreeSerialization();

String serialized = serializer.serialize(root);

System.out.println("Serialized tree: " + serialized);

TreeNode deserializedRoot = serializer.deserialize(serialized);

System.out.println("Deserialized tree: " + serializer.serialize(deserializedRoot));

}

}

8. Heap Sort:

3.4 9.2 1.12 12.09 4.5 7.2

1.12 3.4 4.5 7.2 9.2 12.09

/\*

Heap sort:

0 1 2 3 4 5

3.4 9.2 1.12 12.09 4.5 7.2

3.4 >> max | parent

9.2 1.12

12.09 4.5 7.2

0

3.4 12.09 1.12 9.2 4.5 7.2

3.4 >> max | parent

12.09 1.12

9.2 4.5 7.2

0

12.09 3.4 1.12 9.2 4.5 7.2

12.09 >> max | parent

3.4 1.12

9.2 4.5 7.2

7.2 3.4 1.12 9.2 4.5 12.09

7.2 >> max | parent

3.4 1.12

9.2 4.5

7.2 9.2 1.12 3.4 4.5 12.09

7.2 >> max | parent

9.2 1.12

3.4 4.5

9.2 7.2 1.12 3.4 4.5 12.09

9.2 >> max | parent

7.2 1.12

3.4 4.5

4.5 7.2 1.12 3.4 9.2 12.09

4.5 >> max | parent

7.2 1.12

3.4

7.2 4.5 1.12 3.4 9.2 12.09

7.2 >> max | parent

4.5 1.12

3.4

3.4 4.5 1.12 7.2 9.2 12.09

3.4 >> max | parent

4.5 1.12

4.5 3.4 1.12 7.2 9.2 12.09

4.5 >> max | parent

3.4 1.12

1.12 3.4 4.5 7.2 9.2 12.09

1.12 >> max | parent

3.4

3.4 1.12 4.5 7.2 9.2 12.09

3.4 >> max | parent

1.12

1.12 3.4 4.5 7.2 9.2 12.09

1.12 >> max | parent

\*/

public class Heap {

public static void heapify(double[] arr,int length, int parent){

int maximum=parent;

int left=parent\*2+1;

int right=parent\*2+2;

if(left<length&&arr[left]>arr[maximum]){

maximum=left;

}

if(right<length&&arr[right]>arr[maximum]){

maximum=right;

}

if(maximum!=parent){

double third=arr[maximum];

arr[maximum]=arr[parent];

arr[parent]=third;

// arr[maximum]\*=arr[parent];

// arr[parent]=arr[maximum]/arr[parent];

// arr[maximum]/=arr[parent];

heapify(arr,length,maximum);

}

}

public static void sortHeap(double[] arr){

for(int index=arr.length/2-1;index>=0;index--){

heapify(arr, arr.length, index);

}

for(int position=arr.length-1; position>=0;position-- ){

double third=arr[0];

arr[0]=arr[position];

arr[position]=third;

// arr[0]\*=arr[position];

// arr[position]=arr[0]/arr[position];

// arr[0]/=arr[position];

heapify(arr,position,0);

}

}

public static void show(double[] arr, int index){

if(index<arr.length) {

System.out.print(arr[index] + " ");

index++;

show(arr, index);

}

}

public static void main(String[] args) {

double[] myData={3.4,9.2,1.12,12.09,4.5,7.2};

show(myData,0);

System.out.println();

sortHeap(myData);

show(myData,0);

}

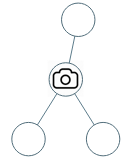
}

9. Binary tree camera

You are given the root of a binary tree. We install cameras on the tree nodes where each camera at a node can monitor its parent, itself, and its immediate children.

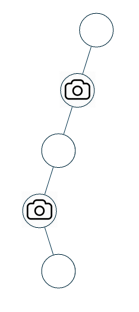
Return the minimum number of cameras needed to monitor all nodes of the tree. with samples

Input: root = [0,0,null,0,0]



Output: 1

Input: root = [0,0,null,0,null,0,null,null,0]



Output: 2