|  |  |
| --- | --- |
| **Binary Tree 2 LL in C++** | |
| #include <iostream>  using namespace std;  class Node {  public:  int key;  Node\* left;  Node\* right;    Node(int value) {  key = value;  left = nullptr;  right = nullptr;  }  };  class BinTree2LL {  private:  static Node\* prev;  public:  static void flatten(Node\* root) {  if (root == nullptr) return;  flatten(root->right);  flatten(root->left);  root->right = prev;  root->left = nullptr;  prev = root;  }  static void printList(Node\* root) {  while (root->right != nullptr) {  cout << root->key << "->";  root = root->right;  }  cout << root->key;  }  };  Node\* BinTree2LL::prev = nullptr;  int main() {  Node\* root = new Node(1);  root->left = new Node(2);  root->left->left = new Node(3);  root->left->right = new Node(4);  root->right = new Node(5);  root->right->right = new Node(6);  root->right->right->left = new Node(7);  BinTree2LL::flatten(root);  BinTree2LL::printList(root);  // Clean up allocated memory (not present in Java version)  while (root != nullptr) {  Node\* temp = root;  root = root->right;  delete temp;  }  return 0;  } | **Original Binary Tree Structure**  1  / \  2 5  / \ \  3 4 6  /  7  **🛠️ Flattening Logic: Reverse Postorder (Right → Left → Node)**  The algorithm works like this:   * Traverse the tree in **reverse postorder**. * Use a static prev pointer to keep track of the previously processed node. * Set the current node’s right to prev, and its left to nullptr.   **🧮 Step-by-Step Tabular Dry Run**  We will track:   * The current node being visited * The state of prev * Links updated  | **Step** | **Node Visited** | **Previous (prev)** | **Action** | **Updated Links** | | --- | --- | --- | --- | --- | | 1 | 7 | nullptr | Set 7.right = nullptr, 7.left = nullptr, prev = 7 | 7 → nullptr | | 2 | 6 | 7 | Set 6.right = 7, 6.left = nullptr, prev = 6 | 6 → 7 | | 3 | 5 | 6 | Set 5.right = 6, 5.left = nullptr, prev = 5 | 5 → 6 → 7 | | 4 | 4 | 5 | Set 4.right = 5, 4.left = nullptr, prev = 4 | 4 → 5 → 6 → 7 | | 5 | 3 | 4 | Set 3.right = 4, 3.left = nullptr, prev = 3 | 3 → 4 → ... | | 6 | 2 | 3 | Set 2.right = 3, 2.left = nullptr, prev = 2 | 2 → 3 → ... | | 7 | 1 | 2 | Set 1.right = 2, 1.left = nullptr, prev = 1 | 1 → 2 → 3 → ... |   **📈 Final Flattened Linked List (Right Pointers)**  1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7  All left pointers are nullptr, forming a **single right-skewed list**.  **✅ Output**  1->2->3->4->5->6->7 |
| 1->2->3->4->5->6->7 | |