|  |  |
| --- | --- |
| **Binary Tree to CDLL in C++** | |
| #include <iostream>  using namespace std;  class Node {  public:  int data;  Node\* left;  Node\* right;  Node(int data) {  this->data = data;  this->left = nullptr;  this->right = nullptr;  }  };  class BinartTree2CDLL {  public:  // Function to concatenate two circular doubly linked lists  Node\* concatenate(Node\* H1, Node\* H2) {  if (H1 == nullptr) return H2;  if (H2 == nullptr) return H1;  Node\* T1 = H1->left;  Node\* T2 = H2->left;  T1->right = H2;  H2->left = T1;  T2->right = H1;  H1->left = T2;  return H1;  }  // Function to convert binary tree into circular doubly linked list  Node\* bTreeToClist(Node\* root) {  if (root == nullptr) return nullptr;  Node\* l = bTreeToClist(root->left);  Node\* r = bTreeToClist(root->right);  root->left = root->right = root;  Node\* result = concatenate(concatenate(l, root), r);  return result;  }  // Function to print the circular doubly linked list  void printCList(Node\* head) {  if (head == nullptr) return;  Node\* temp = head;  do {  cout << temp->data << " ";  temp = temp->right;  } while (temp != head);  cout << endl;  }  };  // Main method to test the bTreeToClist function  int main() {  BinartTree2CDLL solution;  // Creating a sample binary tree:  // 1  // / \  // 2 3  // / \ \  // 4 5 6  Node\* root = new Node(1);  root->left = new Node(2);  root->right = new Node(3);  root->left->left = new Node(4);  root->left->right = new Node(5);  root->right->right = new Node(6);  Node\* head = solution.bTreeToClist(root);  cout << "Circular Doubly Linked List:" << endl;  solution.printCList(head);  // Clean up memory  // In a real-world scenario, you would implement a function to delete the tree nodes.  // For brevity, memory cleanup is not shown in this example.  return 0;  } | Your code to convert a **Binary Tree to a Circular Doubly Linked List (CDLL)** is **elegant and correct**. You're using **in-order traversal** with recursive linking, which is the standard and efficient approach. Let's break it down with a **dry run + visual table** using the tree:  1  / \  2 3  / \ \  4 5 6 🔁 Step-by-Step Dry Run (In-order traversal) Traversal order: **4 → 2 → 5 → 1 → 3 → 6**   | **Call Stack Depth** | **Node Visited** | **Left CDLL** | **Right CDLL** | **Resulting CDLL** | | --- | --- | --- | --- | --- | | 1 | 4 | null | null | 4 | | 1 | 5 | null | null | 5 | | 2 | 2 | 4 | 5 | 4 ⇄ 2 ⇄ 5 | | 1 | 6 | null | null | 6 | | 2 | 3 | null | 6 | 3 ⇄ 6 | | 3 (root) | 1 | 4 ⇄ 2 ⇄ 5 | 3 ⇄ 6 | 4 ⇄ 2 ⇄ 5 ⇄ 1 ⇄ 3 ⇄ 6 |  * ⇄ means CDLL bidirectional links. * At each recursive return, you concatenate left CDLL, root (self-circular), and right CDLL.  ✅ Output Circular Doubly Linked List:  4 2 5 1 3 6 |
| Output:-  Circular Doubly Linked List:  4 2 5 1 3 6 | |