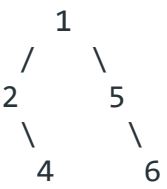
Flatten a binary tree into linked list

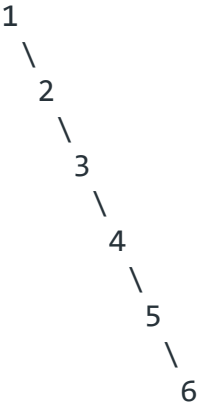
Given a binary tree, flatten it into linked list in-place. Usage of auxiliary data structure is not allowed. After flattening, left of each node should point to NULL and right should contain next node in preorder.

Examples:

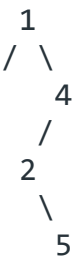
Input :



Output :



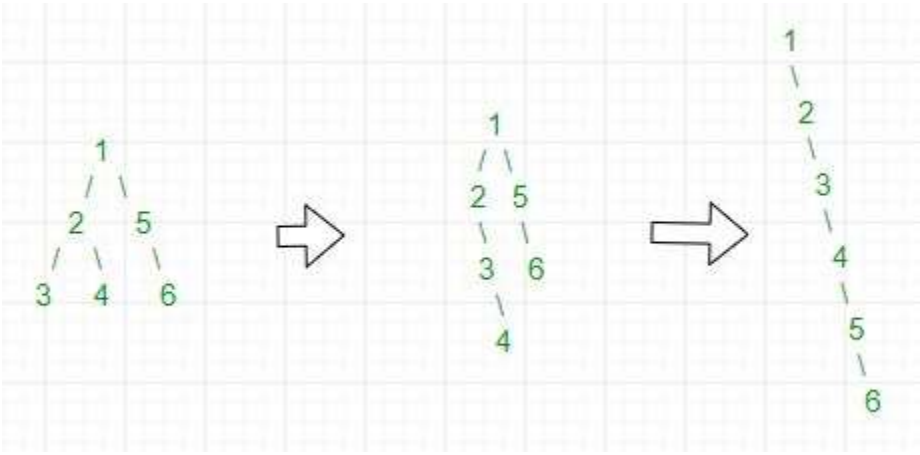
Input :



Output :

Simple Approach: A simple solution is to use Level Order Traversal using Queue. In level order traversal, keep track of previous node. Make current node

as right child of previous and left of previous node as NULL. This solution requires queue, but question asks to solve without additional data structure.
Efficient Without Additional Data Structure Recursively look for the node with no grandchildren and both left and right child in the left sub-tree. Then store node->right in temp and make node->right=node->left. Insert temp in first node NULL on right of node by node=node->right. Repeat until it is converted to linked list. Even though this approach does not require additional data structure , but still it takes space for Recursion stack.
For Example,



Implementation:
C++

```
// C++ Program to flatten a given Binary Tree into linked

// list by using Morris Traversal concept

#include <bits/stdc++.h>

using namespace std;


struct Node {

    int key;

    Node *left, *right;

};


// utility that allocates a new Node with the given key
```

```
Node* newNode(int key)
{
    Node* node = new Node;
    node->key = key;
    node->left = node->right = NULL;
    return (node);
}

// Function to convert binary tree into linked list by
// altering the right node and making left node point to
// NULL
void flatten(struct Node* root)
{
    // base condition- return if root is NULL or if it is a
    // leaf node
    if (root == NULL || root->left == NULL && root->right == NULL)
        return;
    // if root->left exists then we have to make it
    // root->right
    if (root->left != NULL) {
        // move left recursively
        flatten(root->left);
        // store the node root->right
        struct Node* tmpRight = root->right;
        root->right = root->left;
        root->left = NULL;
        // find the position to insert the stored value
```

```
        struct Node* t = root->right;
        while (t->right != NULL)
            t = t->right;
        // insert the stored value
        t->right = tmpRight;
    }
    // now call the same function for root->right
    flatten(root->right);
}

// To find the inorder traversal
void inorder(struct Node* root)
{
    // base condition
    if (root == NULL)
        return;
    inorder(root->left);
    cout << root->key << " ";
    inorder(root->right);
}

/* Driver program to test above functions*/
int main()
{
    /*    1
        /   \
       2     5
```

```
      / \     \
     3   4     6 */
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(5);
    root->left->left = newNode(3);
    root->left->right = newNode(4);
    root->right->right = newNode(6);
    flatten(root);
    cout << "The Inorder traversal after flattening binary tree ";
    inorder(root);
    return 0;
}
```

Output

```
The Inorder traversal after flattening binary tree 1 2 3 4 5 6
```

Complexity Analysis:

- **Time Complexity: O(n),** traverse the whole tree
- **Space Complexity: O(n),** Extra space used for recursion call.

Another Approach:

We will use the intuition behind Morris's traversal. In Morris Traversal we use the concept of a threaded binary tree.

- At a node(say cur) if there exists a left child, we will find the rightmost node in the left subtree(say prev).
- We will set prev's right child to cur's right child,
- We will then set cur's right child to it's left child.
- We will then move cur to the next node by assigning cur it to its right child
- We will stop the execution when cur points to NULL.

Implementation:

C++

```
// C++ Program to flatten a given Binary Tree into linked
// list
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int key;
    Node *left, *right;
};

// utility that allocates a new Node with the given key
Node* newNode(int key)
{
    Node* node = new Node;
    node->key = key;
    node->left = node->right = NULL;
    return (node);
}

// Function to convert binary tree into linked list by
```

```
// altering the right node and making left node point to
// NULL
void flatten(Node* root)
{
    // traverse till root is not NULL
    while (root) {
        // if root->left is not NULL
        if (root->left != NULL) {
            // set curr node as root->left;
            Node* curr = root->left;
            // traverse to the extreme right of curr
            while (curr->right) {
                curr = curr->right;
            }
            // join curr->right to root->right
            curr->right = root->right;
            // put root->left to root->right
            root->right = root->left;
            // make root->left as NULL
            root->left = NULL;
        }
        // now go to the right of the root
        root = root->right;
    }
}

// To find the inorder traversal
```

```
void inorder(struct Node* root)
{
    // base condition
    if (root == NULL)
        return;
    inorder(root->left);
    cout << root->key << " ";
    inorder(root->right);
}

/* Driver program to test above functions*/
int main()
{
    /*    1
        /   \
       2     5
      / \     \
     3   4     6 */
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(5);
    root->left->left = newNode(3);
    root->left->right = newNode(4);
    root->right->right = newNode(6);
    flatten(root);
    cout << "The Inorder traversal after flattening binary "
            "tree ";
```

```
    inorder(root);
    return 0;
}
```

Output

```
The Inorder traversal after flattening binary tree 1 2 3 4 5 6
```

Time Complexity: O(N) Time complexity will be the same as that of a Morris's traversal
Auxiliary Space: O(1)

Another Approach Using Stack:

In this solution, we start by initializing a prev variable to None. This variable will keep track of the previously flattened node as we recursively flatten the binary tree.

- We then define a recursive function flatten that takes in the root node of the binary tree. This function does not return anything, but instead modifies the tree in-place.
- The first thing we do in the flatten function is to check if the root node is None. If it is, we simply return.
- Next, we recursively flatten the right subtree of the root node by calling self.flatten(root.right). This will flatten the right subtree and set self.prev to the rightmost node in the right subtree.
- We then recursively flatten the left subtree of the root node by calling self.flatten(root.left). This will flatten the left subtree and update self.prev to the rightmost node in the flattened left subtree.

- Once we have flattened both the left and right subtrees, we update the root.right pointer to be the previously flattened node (self.prev). We also set the root.left pointer to None to remove the left child.
- Finally, we update self.prev to be the current node (root). This is important because it allows us to keep track of the previously flattened node as we continue to recursively flatten the tree.

This algorithm flattens the binary tree in pre-order traversal, so the resulting "linked list" will be in the same order as a pre-order traversal of the tree.

Implementation:

C++

```
// C++ Program to flatten a given Binary Tree into linked
// list
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int key;
    Node *left, *right;
};

// utility that allocates a new Node with the given key
Node* newNode(int key)
{
    Node* node = new Node;
    node->key = key;
    node->left = node->right = NULL;
    return (node);
}
```

```
// Function to convert binary tree into linked list by
// altering the right node and making left node point to
// NULL

void flatten(Node* root)
    {
  // Base case: if the current node is null, return null
        stack<Node *>st;
        st.push(root);
  // If the left subtree was flattened, merge it with the current node
   // If the right subtree was flattened, return its tail node
   // If neither subtree was flattened, return the current node as the tail node
        while(st.empty()!=true)
        {
            Node *curr = st.top();
            st.pop();
            if(curr==NULL) return;
            if(curr->right!=NULL) st.push(curr->right);// Connect the right subtree of t
left tail to the right subtree of the current node
            if(curr->left!=NULL) st.push(curr->left);// Make the left subtree the new ri
subtree of the current node
            if(st.empty()!=true)// Set the left child of the current node to null
            {
                curr->right = st.top();
            }
            curr->left = NULL;
        }
        return;
```

```
    }

void inorder(struct Node* root)
{
    // base condition
    if (root == NULL)
        return;
    inorder(root->left);
    cout << root->key << " ";
    inorder(root->right);
}

int main()
{
    /*    1
        /   \
       2     5
      / \     \
     3   4     6 */
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(5);
    root->left->left = newNode(3);
    root->left->right = newNode(4);
    root->right->right = newNode(6);
  // Call the recursive helper function
    flatten(root);
```

```
    cout << "The Inorder traversal after flattening binary "
            "tree ";
    inorder(root);
    return 0;
}
```

Output

```
The Inorder traversal after flattening binary tree 1 2 3 4 5 6
```

Time Complexity: O(N), The loop will execute for every node once.
Space Complexity: O(N), Auxiliary Stack Space is needed.