Reverse alternate levels of a perfect binary tree

Given a Perfect Binary Tree, reverse the alternate level nodes of the binary tree.

```
Given tree:
               a
            /     \
           b       c
         /  \     /  \
        d    e    f    g
       / \  / \  / \  / \
       h  i j  k l  m  n  o

Modified tree:
          a
            /     \
           c       b
         /  \     /  \
        d    e    f    g
       / \  / \  / \  / \
      o  n m  l k  j  i  h
```

We strongly recommend to minimize the browser and try this yourself first.

Method 1 (Simple)

A **simple solution** is to do following steps.
1) Access nodes level by level.
2) If current level is odd, then store nodes of this level in an array.
3) Reverse the array and store elements back in tree.

Method 2 (Using Two Traversals)

Another is to do two inorder traversals. Following are steps to be followed.
1) Traverse the given tree in inorder fashion and store all odd level nodes in an auxiliary array. For the above example given tree, contents of array become {h, i, b, j, k, l, m, c, n, o}

2) Reverse the array. The array now becomes {o, n, c, m, l, k, j, b, i, h}

3) Traverse the tree again inorder fashion. While traversing the tree, one by one take elements from array and store elements from array to every odd level traversed node.
For the above example, we traverse 'h' first in above array and replace 'h' with 'o'. Then we traverse 'i' and replace it with n.

Following is the implementation of the above algorithm.

C++

```
// C++ program to reverse alternate levels of a binary tree
#include<iostream>
#define MAX 100
using namespace std;

// A Binary Tree node
struct Node
{
    char data;
    struct Node *left, *right;
};

// A utility function to create a new Binary Tree Node
```

```
struct Node *newNode(char item)
{
    struct Node *temp =  new Node;
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Function to store nodes of alternate levels in an array
void storeAlternate(Node *root, char arr[], int *index, int l)
{
    // Base case
    if (root == NULL) return;

    // Store elements of left subtree
    storeAlternate(root->left, arr, index, l+1);

    // Store this node only if this is a odd level node
    if (l%2 != 0)
    {
        arr[*index] = root->data;
        (*index)++;
    }

    // Store elements of right subtree
    storeAlternate(root->right, arr, index, l+1);
}

// Function to modify Binary Tree (All odd level nodes are
// updated by taking elements from array in inorder fashion)
void modifyTree(Node *root, char arr[], int *index, int l)
{
    // Base case
    if (root == NULL) return;

    // Update nodes in left subtree
    modifyTree(root->left, arr, index, l+1);

    // Update this node only if this is an odd level node
    if (l%2 != 0)
    {
        root->data = arr[*index];
        (*index)++;
    }

    // Update nodes in right subtree
    modifyTree(root->right, arr, index, l+1);
}

// A utility function to reverse an array from index
// 0 to n-1
void reverse(char arr[], int n)
{
    int l = 0, r = n-1;
    while (l < r)
    {
        int temp = arr[l];
        arr[l] = arr[r];
        arr[r] = temp;
        l++; r--;
    }
}

// The main function to reverse alternate nodes of a binary tree
void reverseAlternate(struct Node *root)
{
    // Create an auxiliary array to store nodes of alternate levels
    char *arr = new char[MAX];
    int index = 0;

    // First store nodes of alternate levels
    storeAlternate(root, arr, &index, 0);
```

```
    // Reverse the array
    reverse(arr, index);

    // Update tree by taking elements from array
    index = 0;
    modifyTree(root, arr, &index, 0);
}

// A utility function to print indorder traversal of a
// binary tree
void printInorder(struct Node *root)
{
    if (root == NULL) return;
    printInorder(root->left);
    cout << root->data << " ";
    printInorder(root->right);
}

// Driver Program to test above functions
int main()
{
    struct Node *root = newNode('a');
    root->left = newNode('b');
    root->right = newNode('c');
    root->left->left = newNode('d');
    root->left->right = newNode('e');
    root->right->left = newNode('f');
    root->right->right = newNode('g');
    root->left->left->left = newNode('h');
    root->left->left->right = newNode('i');
    root->left->right->left = newNode('j');
    root->left->right->right = newNode('k');
    root->right->left->left = newNode('l');
    root->right->left->right = newNode('m');
    root->right->right->left = newNode('n');
    root->right->right->right = newNode('o');

    cout << "Inorder Traversal of given tree\n";
    printInorder(root);

    reverseAlternate(root);

    cout << "\n\nInorder Traversal of modified tree\n";
    printInorder(root);

    return 0;
}
```

Java

```
// Java program to reverse alternate levels of  perfect binary tree
// A binary tree node
class Node {

    char data;
    Node left, right;

    Node(char item) {
        data = item;
        left = right = null;
    }
}

// class to access index value by reference
class Index {

    int index;
}
```

```
class BinaryTree {

    Node root;
    Index index_obj = new Index();

    // function to store alternate levels in a tree
    void storeAlternate(Node node, char arr[], Index index, int l) {
        // base case
        if (node == null) {
            return;
        }
        // store elements of left subtree
        storeAlternate(node.left, arr, index, l + 1);

        // store this node only if level is odd
        if (l % 2 != 0) {
            arr[index.index] = node.data;
            index.index++;
        }

        storeAlternate(node.right, arr, index, l + 1);
    }

    // Function to modify Binary Tree (All odd level nodes are
    // updated by taking elements from array in inorder fashion)
    void modifyTree(Node node, char arr[], Index index, int l) {

        // Base case
        if (node == null) {
            return;
        }

        // Update nodes in left subtree
        modifyTree(node.left, arr, index, l + 1);

        // Update this node only if this is an odd level node
        if (l % 2 != 0) {
            node.data = arr[index.index];
            (index.index)++;
        }

        // Update nodes in right subtree
        modifyTree(node.right, arr, index, l + 1);
    }

    // A utility function to reverse an array from index
    // 0 to n-1
    void reverse(char arr[], int n) {
        int l = 0, r = n - 1;
        while (l < r) {
            char temp = arr[l];
            arr[l] = arr[r];
            arr[r] = temp;
            l++;
            r--;
        }
    }

    void reverseAlternate() {
        reverseAlternate(root);
    }

    // The main function to reverse alternate nodes of a binary tree
    void reverseAlternate(Node node) {

        // Create an auxiliary array to store nodes of alternate levels
        char[] arr = new char[100];

        // First store nodes of alternate levels
        storeAlternate(node, arr, index_obj, 0);

        //index_obj.index = 0;
```

```
        // Reverse the array
        reverse(arr, index_obj.index);

        // Update tree by taking elements from array
        index_obj.index = 0;
        modifyTree(node, arr, index_obj, 0);
    }

    void printInorder() {
        printInorder(root);
    }

    // A utility function to print indorder traversal of a
    // binary tree
    void printInorder(Node node) {
        if (node == null) {
            return;
        }
        printInorder(node.left);
        System.out.print(node.data + " ");
        printInorder(node.right);
    }

    // Driver program to test the above functions
    public static void main(String args[]) {
        BinaryTree tree = new BinaryTree();
        tree.root = new Node('a');
        tree.root.left = new Node('b');
        tree.root.right = new Node('c');
        tree.root.left.left = new Node('d');
        tree.root.left.right = new Node('e');
        tree.root.right.left = new Node('f');
        tree.root.right.right = new Node('g');
        tree.root.left.left.left = new Node('h');
        tree.root.left.left.right = new Node('i');
        tree.root.left.right.left = new Node('j');
        tree.root.left.right.right = new Node('k');
        tree.root.right.left.left = new Node('l');
        tree.root.right.left.right = new Node('m');
        tree.root.right.right.left = new Node('n');
        tree.root.right.right.right = new Node('o');
        System.out.println("Inorder Traversal of given tree");
        tree.printInorder();

        tree.reverseAlternate();
        System.out.println("");
        System.out.println("");
        System.out.println("Inorder Traversal of modified tree");
        tree.printInorder();
    }
}

// This code has been contributed by Mayank Jaiswal
```

Output:

```
Inorder Traversal of given tree
h d i b j e k a l f m c n g o

Inorder Traversal of modified tree
o d n c m e l a k f j b i g h
```

Time complexity of the above solution is O(n) as it does two inorder traversals of binary tree.

Method 3 (Using One Traversal)

```
// C++ program to reverse alternate levels of a tree
#include <bits/stdc>
using namespace std;
```

```
struct Node
{
    char key;
    Node *left, *right;
};

void preorder(struct Node *root1, struct Node* root2, int lvl)
{
    // Base cases
    if (root1 == NULL || root2==NULL)
        return;

    // Swap subtrees if level is even
    if (lvl%2 == 0)
        swap(root1->key, root2->key);

    // Recur for left and right subtrees (Note : left of root1
    // is passed and right of root2 in first call and opposite
    // in second call.
    preorder(root1->left, root2->right, lvl+1);
    preorder(root1->right, root2->left, lvl+1);
}

// This function calls preorder() for left and right children
// of root
void reverseAlternate(struct Node *root)
{
   preorder(root->left, root->right, 0);
}

// Inorder traversal (used to print initial and
// modified trees)
void printInorder(struct Node *root)
{
    if (root == NULL)
       return;
    printInorder(root->left);
    cout << root->key << " ";
    printInorder(root->right);
}

// A utility function to create a new node
Node *newNode(int key)
{
    Node *temp = new Node;
    temp->left = temp->right = NULL;
    temp->key = key;
    return temp;
}

// Driver program to test above functions
int main()
{
    struct Node *root = newNode('a');
    root->left = newNode('b');
    root->right = newNode('c');
    root->left->left = newNode('d');
    root->left->right = newNode('e');
    root->right->left = newNode('f');
    root->right->right = newNode('g');
    root->left->left->left = newNode('h');
    root->left->left->right = newNode('i');
    root->left->right->left = newNode('j');
    root->left->right->right = newNode('k');
    root->right->left->left = newNode('l');
    root->right->left->right = newNode('m');
    root->right->right->left = newNode('n');
    root->right->right->right = newNode('o');

    cout << "Inorder Traversal of given tree\n";
    printInorder(root);

    cout << "\n\nInorder Traversal of modified tree\n";
```

```
    cout << "\n\nInorder Traversal of modified tree\n";
    printInorder(root);
    return 0;
}
```

Output :

```
Inorder Traversal of given tree
h d i b j e k a l f m c n g o

Inorder Traversal of modified tree
h d i b j e k a l f m c n g o
```