Maximum Sum of Non-adjacent Nodes :-

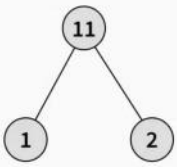
Maximum sum of Non-adjacent nodes

Difficulty: **Medium** Accuracy: **55.35%** Submissions: **83K+** Points: **4** Average Time: **45m**

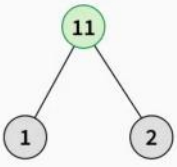
Given a **binary tree** with a value associated with each node. Your task is to select a **subset of nodes** such that the sum of their values is **maximized**, with the condition that no two selected nodes are **directly connected** that is, if a node is included in the subset, neither its **parent** nor its **children** can be included.

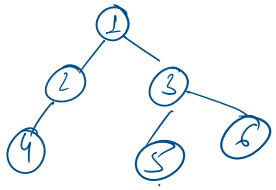
Examples:

Input: root[] = [11, 1, 2]



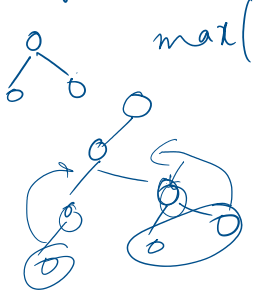
Output: 11
Explanation: The maximum sum is obtained by selecting the node 11.

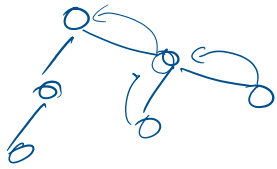




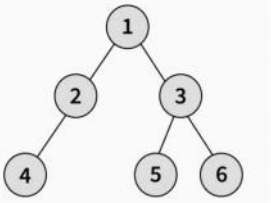
leaf node: return root->data

max(l+r, root->data)

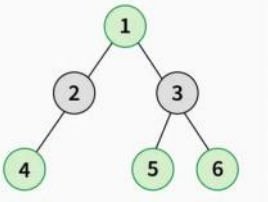




Input: root[] = [1, 2, 3, 4, N, 5, 6]



Output: 16
Explanation: The maximum sum is obtained by selecting the nodes 1, 4, 5, and 6, which are not directly connected to each other. Their total sum is 16.



Constraints:
$1 \leq$ no. of nodes in the tree $\leq 10^4$
$1 \leq$ Node.val $\leq 10^5$

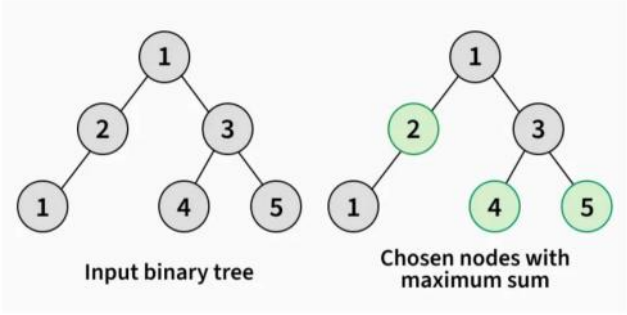
[Naive Approach] Using Recursion - O(2^n) Time and O(n) Space

We can solve this problem by considering the fact that **both node and its immediate children** can't be in **sum** at the same time.

- **Include the current node's value in the sum:** In this case, we cannot include the values of its immediate children in the sum. Therefore, we recursively call the function on the **grandchildren** of the current node.
- **Exclude the current node's value in the sum:** In this case, we are allowed to include the values of its **immediate children** in the sum. So, we recursively call the function on the immediate children of the current node.
- Finally we will choose **maximum** from both of the results.

```
class Solution {
  public:
    // Function to return the maximum sum of non-adjacent nodes.
   int getMaxSumUtil(Node* node) {
    if (node == nullptr) {

        // If the node is null, the sum is 0
        return 0;
    }

    // Calculate the maximum sum including the
    // current node
    int incl = node->data;

    // If the left child exists, include its contribution
    if (node->left) {
        incl += getMaxSumUtil(node->left->left) +
                getMaxSumUtil(node->left->right);
    }

    // If the right child exists, include its contribution
    if (node->right) {
        incl += getMaxSumUtil(node->right->left) +
                getMaxSumUtil(node->right->right);
    }

    // Calculate the maximum sum excluding
    // the current node
    int excl = 0;
    if (node->left) {
        excl += getMaxSumUtil(node->left);
    }
    if (node->right) {
        excl += getMaxSumUtil(node->right);
    }

    // The result for the current node is the
    // maximum of including or excluding it
    return max(incl, excl);
}

int getMaxSum(Node* root) {

    // If the tree is empty, the maximum sum is 0
    if (root == nullptr) {
        return 0;
    }

    // Call the utility function to compute the
    // maximum sum for the entire tree
    return getMaxSumUtil(root);
}
};
```

[Expected Approach 1] Using Top-Down DP (Memoization) - O(n) Time and O(n) Space



The naive approach leads to **recalculating** results for the same nodes multiple times. For example, if we **include** the root node, we **recursively** compute the sum for its **grandchildren (nodes 4 and 5).** But if we **exclude** the root, we compute the sum for its children, and **node 3** also computes the sum for its children **(4 and 5 again).**

To avoid this redundancy, we use **memoization**:

- We store the result of each node in a **hashmap.**
- When a node's value is needed again, we directly **return** it from the map instead of recalculating.

x

```
*/

class Solution {
  public:
    // Function to return the maximum sum of non-adjacent nodes.
    int getMaxSumUtil(Node* node, unordered_map<Node*, int>& memo) {
    if (node == nullptr) {

        // If the node is null, the sum is 0
        return 0;
    }

    // If the result is already computed, return it from memo
    if (memo.find(node) != memo.end()) {
        return memo[node];
    }

    // Calculate the maximum sum including the current node
    int incl = node->data;

    // If the left child exists, include its grandchildren
    if (node->left) {
        incl += getMaxSumUtil(node->left->left, memo) +
                getMaxSumUtil(node->left->right, memo);
    }

    // If the right child exists, include its grandchildren
    if (node->right) {
        incl += getMaxSumUtil(node->right->left, memo) +
                getMaxSumUtil(node->right->right, memo);
    }

    // Calculate the maximum sum excluding the current node
    int excl = getMaxSumUtil(node->left, memo) +
               getMaxSumUtil(node->right, memo);

    // Store the result in memo and return
    // the maximum of incl and excl
    memo[node] = max(incl, excl);
    return memo[node];
}

// Function to compute the maximum
// sum of non-adjacent nodes
int getMaxSum(Node* root) {
    unordered_map<Node*, int> memo;
    return getMaxSumUtil(root, memo);
}

};
```

[Expected Approach 2] Using Pairs - O(n) Time and O(h) Space

Return a **pair** for each node in the binary tree such that the **first** of the pair indicates the **maximum sum** when the data of a node is **included** and the second indicates the maximum sum when the data of a particular node is not included.

```
/

class Solution {
  public:
    pair<int, int> maxSumHelper(Node* root){
        if(root==nullptr){
            pair<int, int> sum(0, 0);
            return sum;
        }
        pair<int, int> sum1=maxSumHelper(root->left);
        pair<int, int> sum2=maxSumHelper(root->right);
        pair<int, int> sum;
        // This node is included(left and right children are not included)
        sum.first=sum1.second+sum2.second+root->data;
        // This node is excluded(either left or right child is included)
        sum.second=max(sum1.first, sum1.second)+max(sum2.first, sum2.second);
        return sum;
    }
    // Function to return the maximum sum of non-adjacent nodes.
    int getMaxSum(Node *root) {
        // code here
        pair<int, int> res=maxSumHelper(root);
        return max(res.first, res.second);
    }
};

```