Trees : Level 2
Lesson 2

height-using-parent-array
Count-number-of-subtrees-having-given-sum
Extreme-nodes-in-alternate-order
serialize-and-deserialize-a-binary-tree
duplicate-subtree-in-binary-tree
single-valued-subtree

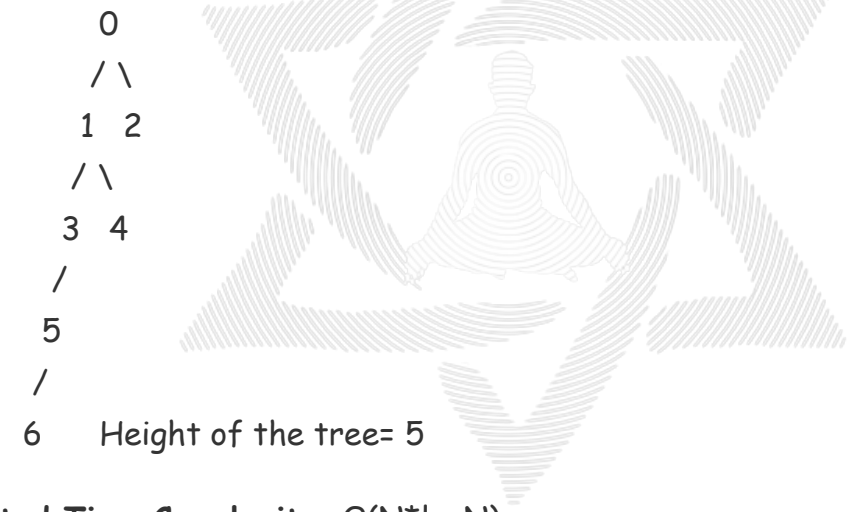
Height Using Parent Array

Given a parent array **arr**[] of a binary tree of **N** nodes. Element at index i in the array arr[] represents the parent of ith node, i.e, **arr[i]** = **parent(i)**. Find the height of this binary tree.

Note: There will be a node in the array arr[], where **arr[i]** = **-1**, which means this node is the root of the binary tree.

Example 1:

Input: N = 7
arr = {-1, 0, 0, 1, 1, 3, 5}
Output: 5
Explanation: Tree formed is:



Height of the tree= 5

Expected Time Complexity: *O*(N*logN)

Expected Auxiliary Space: *O*(1)

Constraints:

1 ≤ N ≤ 104

1 ≤ arr[i] ≤ 104

Now let's look at the approach where we don't need to construct the binary tree to find the height of it from its parent array representation. The steps of this algorithm are as following -

1. Create height array of size same as parent array. For all legal values of 'i' make height[i] = INVALID_HEIGHT. height[i] denotes the height of node with value 'i'.
2. To compute the height for all nodes of this tree, we call computeHeight(i, height, parent) for all values of legal values of 'i'.
3. In computeHeight(i, height, parent) recursive function,
3a. If height[i] is already computed, we return to the calling function.
3b. If parent[i] == -1, then that means this index corresponds to root node. Hence we make height[i] = 1 and return. Note that, as against the conventional notion of height here we have height of root node as minimum and height of leaf node as maximum.
3c. If conditions 3a and 3b are not true, then first we compute the height of the parent node by making a recursive call computeHeight(parent[i], height, parent). This parent height would be used to compute the height of the current node.
3d. Once we compute height of parent node, we compute height of current node with value 'i' as height[i] = 1 + height[parent[i]] and return.
4. After computeHeight() call sequence is complete, we have heights for all tree nodes. Now all we do is scan the height array to find and return the maximum height from this height array.

```
1.    class Solution{
2.    public:
3.
4.      void height(int arr[] , int i , vector<int>&v)
5.      {
6.          if(v[i])
7.          {
8.              return;
9.          }
10.         if(arr[i]==-1)
11.         {
12.             v[i]=1;
13.             return;
```

```
14.         }
15.
16.         if(v[arr[i]]==0)
17.         {
18.             height(arr,arr[i],v);
19.         }
20.
21.         v[i]=v[arr[i]]+1;
22.     }
23.     int findHeight(int N, int arr[])
24.     {
25.         vector<int>v(N,0);
26.         for(int i=0;i<N;i++)
27.             height(arr,i,v);
28.         sort(v.begin(),v.end());
29.         return v[v.size()-1];
30.         // code here
31.     }
32. };
```

Count Number of SubTrees having given Sum

Given a binary tree and an integer **X**. Your task is to complete the function **countSubtreesWithSumX()** that returns the count of the number of subtress having total node's data sum equal to the value **X**.

Example 1:

Input:

```
     5
   /   \
 -10    3
/  \   /  \
9   8 -4   7
```

X = 7

Output: 2

Example 2:

Input:

```
   1
 /  \
2    3
```

X = 5

Output: 0

Expected Time Complexity: O(N).

Expected Auxiliary Space: O(Height of the Tree).

Constraints:

1 <= N <= 103

-103 <= Node Value <= 103

Method 1:Convert to SumTree

```
1.  int sumtree(Node*root)
2.  {
3.    if(root==NULL)
4.       return 0;
5.    if(!root->left && !root->right)
6.       return root->data;
7.    int x=root->data;
8.    root->data=x+sumtree(root->left)+sumtree(root->right);
9.    return root->data;
10. }
11. void count(Node*root, int X,int &nodes)
12. {
13.   if(root==NULL)
14.      return;
15.   if(root->data==X)
16.      nodes++;
17.   count(root->left,X,nodes);
18.   count(root->right,X,nodes);
19. }
20. //Function to count number of subtrees having sum equal to given sum.
21. int countSubtreesWithSumX(Node* root, int X)
22. {
23.   int nodes=0;
24.   sumtree(root);
25.   count(root,X,nodes);
26.   return nodes;
27.       // Code here
28. }
```

Method 2: Traverse Recursively

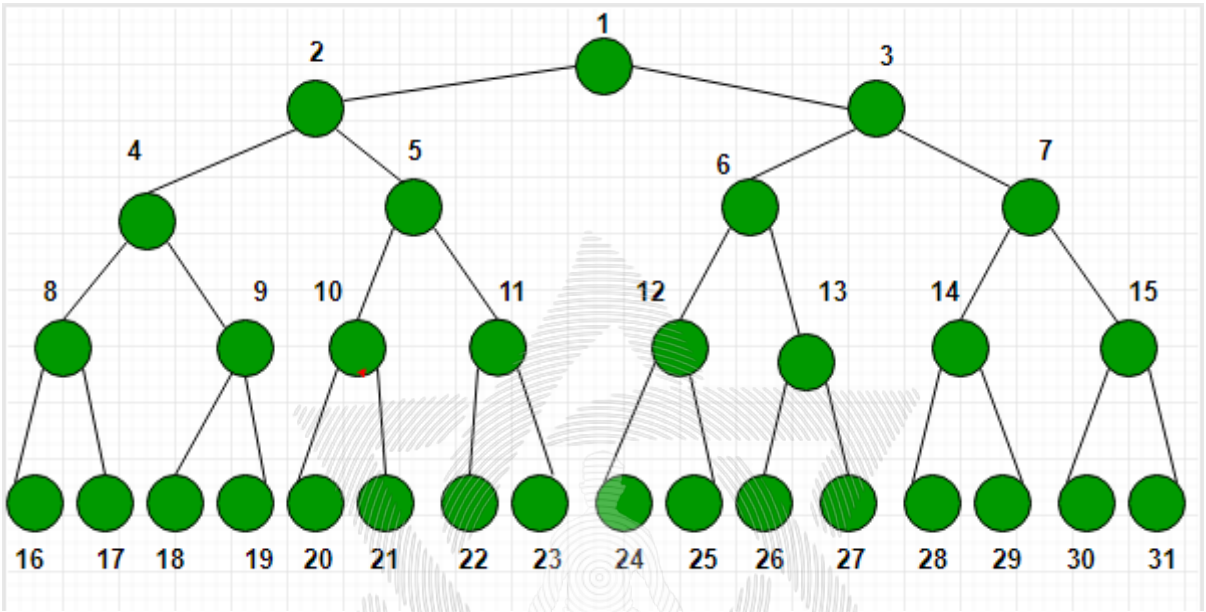
```
1.  int countSubtreesWithSumX(Node* root,
2.                            int& count, int x)
3.  {
4.    // if tree is empty
5.    if (!root)
6.      return 0;
7.
8.    // sum of nodes in the left subtree
9.    int ls = countSubtreesWithSumX(root->left, count, x);
10.
11.   // sum of nodes in the right subtree
12.   int rs = countSubtreesWithSumX(root->right, count, x);
13.
14.   // sum of nodes in the subtree rooted
15.   // with 'root->data'
16.   int sum = ls + rs + root->data;
17.
18.   // if true
19.   if (sum == x)
20.     count++;
21.
22.   // return subtree's nodes sum
23.   return sum;
24. }
25.
26. // utility function to count subtress that
27. // sum up to a given value x
28. int countSubtreesWithSumXUtil(Node* root, int x)
29. {
```

```
30.     // if tree is empty
31.     if (!root)
32.         return 0;
33.
34.     int count = 0;
35.
36.     // sum of nodes in the left subtree
37.     int ls = countSubtreesWithSumX(root->left, count, x);
38.
39.     // sum of nodes in the right subtree
40.     int rs = countSubtreesWithSumX(root->right, count, x);
41.
42.     // if tree's nodes sum == x
43.     if ((ls + rs + root->data) == x)
44.         count++;
45.
46.     // required count of subtrees
47.     return count;
48. }
```

Extreme nodes in alternate order

Given a binary tree, print nodes of extreme corners of each level but in alternate order.

Example:



For above tree, the output is

1 2 7 8 31

- print rightmost node of 1st level

- print leftmost node of 2nd level

- print rightmost node of 3rd level

- print leftmost node of 4th level

- print rightmost node of 5th level

Example 1:

Input:

```
     1
   / \
  2   3
```

Output: 1 2

Expected Time Complexity: O(N).

Expected Auxiliary Space: O(N).

Constraints:

1 <=Number of nodes<= 100

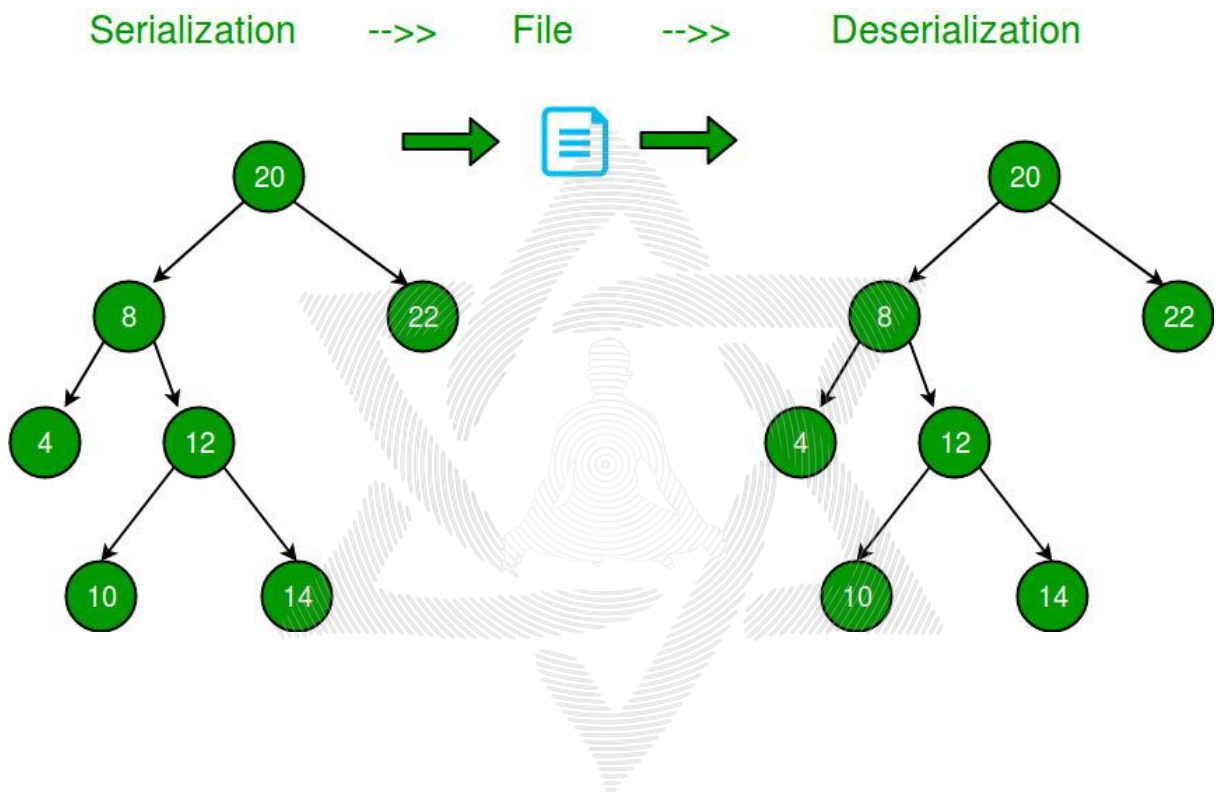
1 <=Data of a node<= 1000

```
class Solution{
 public:
   /* Function to print nodes of extreme corners
    of each level in alternate order */
   vector<int> ExtremeNodes(Node* root)
   {
       vector<int>v;
       queue<Node*>q;
       int level=1;
       q.push(root);
       while(!q.empty())
       {
          int n = q.size();
          for(int i=0;i<n;i++)
          {
             Node*temp=q.front();
             if(level==0 && i==0)
             {
                v.push_back(q.front()->data);
             }
             if(level==1 && i==n-1)
             {
```

```
23.                 v.push_back(q.front()->data);
24.             }
25.             if(temp->left)
26.                 q.push(temp->left);
27.             if(temp->right)
28.                 q.push(temp->right);
29.             q.pop();
30.         }
31.         if(level==1)
32.             level=0;
33.         else
34.             level=1;
35.     }
36.     return v;
37.     // Your code here
38.   }
39. };
```

Serialize and Deserialize a Binary Tree

Serialization is to store tree in a file so that it can be later restored. The structure of tree must be maintained. Deserialization is reading tree back from file.



Following are some simpler versions of the problem:

If given Binary Tree is Complete Tree? A Binary Tree is complete if all levels are completely filled except possibly the last level and all nodes of last level are as left as possible (Binary Heaps are complete Binary Tree). For a complete Binary Tree, level order traversal is sufficient to store the tree. We know that the first node is root, next two nodes are nodes of next level, next four nodes are nodes of 2nd level and so on.

If given Binary Tree is Full Tree? A full Binary is a Binary Tree where every node has either 0 or 2 children. It is easy to serialize such trees as every internal node has 2 children. We can simply store preorder traversal and store a bit with every node to indicate whether the node is an internal node or a leaf node.

How to store a general Binary Tree?

A simple solution is to store both Inorder and Preorder traversals. This solution requires space twice the size of Binary Tree.

We can save space by storing Preorder traversal and a marker for NULL pointers.

Let the marker for NULL pointers be '-1'

Input:

```
    12
   /
  13
```

Output: 12 13 -1 -1 -1

Input:

```
     20
   /   \
  8     22
```

Output: 20 8 -1 -1 22 -1 -1

Input:

```
       20
     /
    8
   / \
  4  12
    /  \
   10  14
```

Output: 20 8 4 -1 -1 12 10 -1 -1 14 -1 -1 -1

Input:

```
       20
      /
     8
    /
   10
  /
 5
```

Output: 20 8 10 5 -1 -1 -1 -1 -1

Input:

```
       20
         \
          8
           \
            10
              \
               5
```

Output: 20 -1 8 -1 10 -1 5 -1 -1

Deserialization can be done by simply reading data from file one by one.

Following is the implementation of the above idea.

```
1.  class Solution
2.  {
3.    public:
4.    void ser(Node*root,vector<int>&v)
5.    {
6.      if(root==NULL)
7.      {
8.        v.push_back(0);
9.        return;
10.     }
11.     v.push_back(root->data);
12.     ser(root->left,v);
13.     ser(root->right,v);
14.   }
15.   //Function to serialize a tree and return a list containing nodes of tree.
16.   vector<int> serialize(Node *root)
17.   {
18.     vector<int>v;
19.     ser(root,v);
20.     return v;
21.     //Your code here
22.   }
23.   Node* des(vector<int>&A,int &index)
24.   {
25.     if(A[index]==0)
26.     {
```

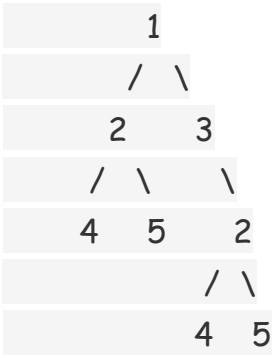
```
27.             index++;
28.             return NULL;
29.         }
30.         Node*node=new Node(A[index]);
31.         index++;
32.         node->left=des(A,index);
33.         node->right=des(A,index);
34.
35.         return node;
36.     }
37.     //Function to deserialize a list and construct the tree.
38.     Node * deSerialize(vector<int> &A)
39.     {
40.         int index=0;
41.         //Node*root=NULL;
42.         return des(A,index);
43.        //Your code here
44.     }
45.
46. };
```

Duplicate subtree in Binary Tree

Given a binary tree, find out whether it contains a duplicate sub-tree of size two or more, or not.

Example 1 :

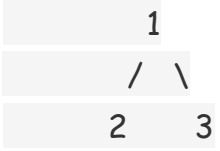
Input :



Output : 1

Example 2 :

Input :



Output: 0

Note: Two same leaf nodes are not considered as subtree as size of a leaf node is one.

Constraints:

1<=length of string<=100

[Method 1]

A simple solution is that, we pick every node of tree and try to find is any sub-tree of given tree is present in tree which is identical with that sub-tree. Here we can use below post to find if a subtree is present anywhere else in tree.

Check if a binary tree is subtree of another binary tree

Upar waala question

[Method 2](Efficient solution)

An Efficient solution based on tree serialization and hashing. The idea is to serialize subtrees as strings and store the strings in hash table. Once we find a serialized tree (which is not a leaf) already existing in hash-table, we return true.

```
string subtree(Node*root,unordered_set<string>&subtrees)
1.    {
2.       string s="";
3.       if(root==NULL)
4.          return s+"N";
5.
6.       string lstr=subtree(root->left,subtrees);
7.       if(lstr==s)
8.          return s;
9.
10.      string rstr=subtree(root->right,subtrees);
11.      if(rstr==s)
12.         return s;
13.
14.      s=s+root->data+lstr+rstr;
15.
16.
```

```
17.      if(s.length()>3 && subtrees.find(s)!=subtrees.end())
18.          return "";
19.
20.      subtrees.insert(s);
21.      return s;
22.    }
23.    /*This function returns true if the tree contains
24.    a duplicate subtree of size 2 or more else returns false*/
25.    bool dupSub(Node *root)
26.    {
27.      unordered_set<string>subtrees;
28.      string s=subtree(root,subtrees);
29.      if(s=="")
30.          return true;
31.      else
32.          return false;
33.       //your code here
34.    }
```

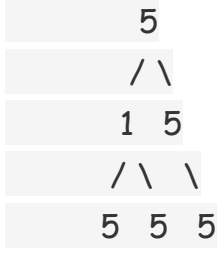
Time and space complexity??

Single valued subtree

Given a binary tree, count the number of Single Valued Subtrees. A Single Valued Subtree is one in which all the nodes have same value.

Example 1

Input :



Output : 4

Example 2:

Input:

Output: 5

Expected Time Complexity : O(n)
Expected Auxiliary Space : O(n)

Constraints :
1 <= n <= 10^5

```
1.  class Solution
2.  {
3.    public:
4.    bool subtree(Node*root,int &count)
```

```
5.      {
6.          if(root==NULL)
7.              return true;
8.          bool ls=subtree(root->left,count);
9.          bool rs=subtree(root->right,count);
10.
11.         if(!ls||!rs)
12.             return false;
13.         if(root->left && root->data!=root->left->data)
14.             return false;
15.         if(root->right && root->data!=root->right->data)
16.             return false;
17.
18.         count++;
19.         return true;
20.
21.     }
22.     int singlevalued(Node *root)
23.     {
24.         int count=0;
25.         subtree(root,count);
26.         return count;
27.         //code here
28.     }
29.
30. };
```