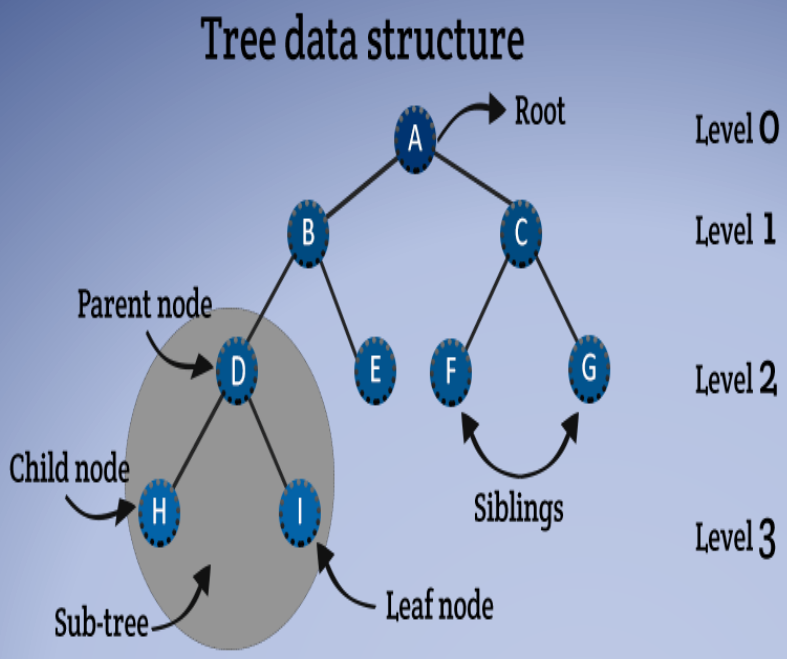


Problems on Tree

Contributer:

Santosh Kumar Mishra
Software Engineer at Microsoft
LinkedIn: @iamsantoshmishra
Email: 93mishra@gmail.com

In computer science, a tree is a widely used abstract data type (ADT)—or data structure implementing this ADT—that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.

Problems on Tree

Contents

1 Binary Tree | Set 1 (Introduction)

http://quiz.geeksforgeeks.org/binary-tree-set-1-introduction/

Trees: Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.

Tree Vocabulary: The topmost node is called root of the tree. The elements that are directly under an element are called its children. The element directly above something is called its parent. For example, a is a child of f and f is the parent of a. Finally, elements with no children are called leaves.

Why Trees?

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

```
file system
-----------
    /   <-- root
 /     \
...      home
     /       \
  ugrad       course
   /     /    |    \
 ...     cs101  cs112  cs113
```

2. Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).
3. Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).
4. Like Linked Lists and unlike Arrays, Trees don't have an upper limit on number of nodes as nodes are linked using pointers.

Main applications of trees include:

1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms
6. Form of a multi-stage decision-making (see business chess).

Binary Tree: A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.
Binary Tree Representation in C: A tree is represented by a pointer to the topmost node in tree. If the tree is empty, then value of root is NULL.
A Tree node contains following parts.
1. Data
2. Pointer to left child
3. Pointer to right child

Summary: Tree is a hierarchical data structure. Main uses of trees include maintaining hierarchical data, providing moderate access and insert/delete operations. Binary trees are special cases of tree where every node has at most two children.

2 Binary Tree | Set 2 (Properties)

http://quiz.geeksforgeeks.org/binary-tree-set-2-properties/
1) The maximum number of nodes at level 'l' of a binary tree is 2l-1.
Here level is number of nodes on path from root to the node (including root and node). Level of root is 1.
This can be proved by induction.
For root, l = 1, number of nodes = 21-1 = 1
Assume that maximum number of nodes on level l is 2l-1
Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e. 2 * 2l-1

2) Maximum number of nodes in a binary tree of height 'h' is 2h – 1.
Here height of a tree is maximum number of nodes on root to leaf path. Height of a leaf node is considered as 1.
This result can be derived from point 2 above. A tree has maximum nodes if all levels have maximum nodes. So maximum number of nodes in a binary tree of height h is 1 + 2 + 4 + .. + 2h-1. This is a simple geometric series with h terms and sum of this series is 2h – 1.
In some books, height of a leaf is considered as 0. In this convention, the above formula becomes 2h+1 – 1

3) In a Binary Tree with N nodes, minimum possible height or minimum number of levels is [Log2(N+1)]

This can be directly derived from point 2 above. If we consider the convention where height of a leaf node is considered as 0, then above formula for minimum possible height becomes [Log2(N+1)] – 1

4) **A Binary Tree with L leaves has at least [Log2L] + 1 levels**

A Binary tree has maximum number of leaves when all levels are fully filled. Let all leaves be at level l, then below is true for number of leaves L.

L <= 2l-1 [From Point 1]
Log2L <= l-1
l >= [Log2L] + 1

5) In Binary tree, number of leaf nodes is always one more than nodes with two children.

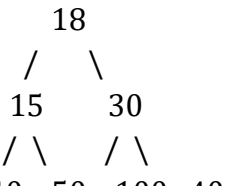
L = T + 1

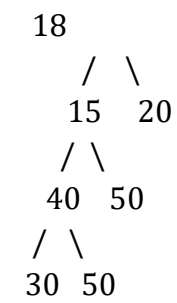
Where L = Number of leaf nodes
T = Number of internal nodes with two children

3 Binary Tree | Set 3 (Types of Binary Tree)

http://quiz.geeksforgeeks.org/binary-tree-set-3-types-of-binary-tree/

Full Binary Tree: A Binary Tree is full if every node has 0 or 2 children. Following are examples of full binary tree.





In a Full Binary, number of leaf nodes is number of internal nodes plus 1

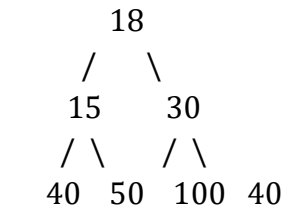
L = I + 1

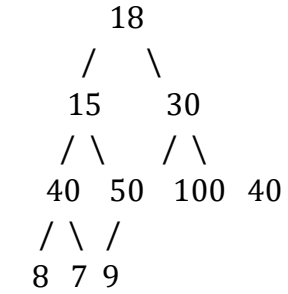
Where L = Number of leaf nodes, I = Number of internal nodes

See Handshaking Lemma and Tree for proof.

Complete Binary Tree: A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible

Following are examples of Complete Binary Trees



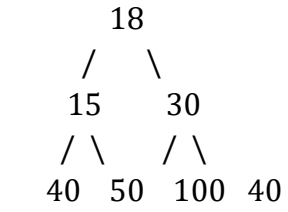


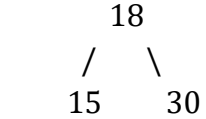
Practical example of Complete Binary Tree is Binary Heap.

Perfect Binary Tree

A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at same level.

Following are examples of Perfect Binaryr Trees.





A Perfect Binary Tree of height h (where height is number of nodes on path from root to leaf) has 2h – 1 node.

Example of Perfect binary tree is ancestors in family. Keep a person at root, parents as children, parents of parents as their children.

Balanced Binary Tree

A binary tree is balanced if height of the tree is O(Log n) where n is number of nodes. For Example, AVL tree maintain O(Log n) height by making sure that the difference between heights of left and right subtrees is 1. Red-Black trees maintain O(Log n) height by making sure that the number of Black nodes on every root to leaf paths are same and there are no adjacent red nodes. Balanced Binary Search trees are performance wise good as they provide O(log n) time for search, insert and delete.

A degenerate (or pathological) tree A Tree where every internal node has one child. Such trees are performance-wise same as linked list.

```
    10
    /
   20
    \
    30
     \
     40
```

4 Enumeration of Binary Trees

http://quiz.geeksforgeeks.org/enumeration-of-binary-trees/

A Binary Tree is labeled if every node is assigned a label and a Binary Tree is unlabeled if nodes are not assigned any label.

Below two are considered same unlabeled trees

```
  o          o
 / \        / \
o   o      o   o
```

Below two are considered different labeled trees

```
  A          C
 / \        / \
B   C      A   B
```

How many different Unlabeled Binary Trees can be there with n nodes?

For n = 1, there is only one tree

```
  o
```

For n = 2, there are two trees

```
  o    o
 /      \
o        o
```

The **idea** is to consider all possible pair of counts for nodes in left and right subtrees and multiply the counts for a particular pair. Finally add results of all pairs.

For example, let T(n) be count for n nodes.
T(0) = 1 [There is only 1 empty tree]
T(1) = 1
T(2) = 2

T(3) = T(0)*T(2) + T(1)*T(1) + T(2)*T(0) = 1*2 + 1*1 + 2*1 = 5

T(4) = T(0)*T(3) + T(1)*T(2) + T(2)*T(1) + T(3)*T(0)
= 1*5 + 1*2 + 2*1 + 5*1
= 14

The above pattern basically represents n'th Catalan Numbers. First few catalan numbers are 1 1 2 5 14 42 132 429 1430 4862,…

$$T(n) = \sum_{i=1}^{n} T(i-1)T(n-i) = \sum_{i=0}^{n-1} T(i)T(n-i-1) = C_n$$

Here,
T(i-1) represents number of nodes on the left-sub-tree
T(n−i-1) represents number of nodes on the right-sub-tree
n'th Catalan Number can also be evaluated using direct formula.
T(n) = (2n)! / (n+1)!n!

Number of Binary Search Trees (BST) with n nodes is also same as number of unlabeled trees. The reason for this is simple, in BST also we can make any key as root, If root is i'th key in sorted order, then i-1 keys can go on one side and (n-i) keys can go on other side.

How many labeled Binary Trees can be there with n nodes?
To count labeled trees, we can use above count for unlabeled trees. The idea is simple, every unlabeled tree with n nodes can create n! different labeled trees by assigning different permutations of labels to all nodes.
Therefore,
Number of Labeled Tees = (Number of unlabeled trees) * n!
= [(2n)! / (n+1)!n!] × n!
For example for n = 3, there are 5 * 3! = 5*6 = 30 different labeled trees

5 Applications of tree data structure

http://www.geeksforgeeks.org/applications-of-tree-data-structure/

Why Tree?

Unlike Array and Linked List, which are linear data structures, tree is hierarchical (or non-linear) data structure.
1) One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

file system
———–
/ <-- root
/ \
... home
/ \
ugrad course
/ / | \
... cs101 cs112 cs113

2) If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of O(Logn) for search.

3) We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of O(Logn) for insertion/deletion.

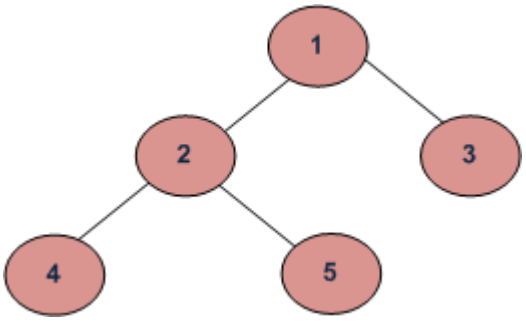
4) Like Linked Lists and unlike Arrays, Pointer implementation of trees don't have an upper limit on number of nodes as nodes are linked using pointers.
As per Wikipedia, following are the common uses of tree.
1. Manipulate hierarchical data.
2. Make information easy to search (see tree traversal).
3. Manipulate sorted lists of data.
4. As a workflow for compositing digital images for visual effects.
5. Router algorithms

6 BFS vs DFS for Binary Tree

http://www.geeksforgeeks.org/bfs-vs-dfs-binary-tree/

What are BFS and DFS for Binary Tree?

A Tree is typically traversed in two ways:
Breadth First Traversal (Or Level Order Traversal)
Depth First Traversals
Inorder Traversal (Left-Root-Right)
Preorder Traversal (Root-Left-Right)
Postorder Traversal (Left-Right-Root)



BFS and DFSs of above Tree

Breadth First Traversal : 1 2 3 4 5

Depth First Traversals:
Preorder Traversal : 1 2 4 5 3
Inorder Traversal : 4 2 5 1 3
Postorder Traversal : 4 5 2 3 1

Why do we care?
There are many tree questions that can be solved using any of the above four traversals. Examples of such questions are size, maximum, minimum, print left view, etc.

Is there any difference in terms of Time Complexity?
All four traversals require O(n) time as they visit every node exactly once.
Is there any difference in terms of Extra Space?
There is difference in terms of extra space required.

Extra Space required for Level Order Traversal is O(w) where w is maximum width of Binary Tree. In level order traversal, queue one by one stores nodes of different level.

Extra Space required for Depth First Traversals is O(h) where h is maximum height of Binary Tree. In Depth First Traversals, stack (or function call stack) stores all ancestors of a node.

Maximum Width of a Binary Tree at depth (or height) h can be 2h where h starts from 0. So the maximum number of nodes can be at the last level. And worst case occurs when Binary Tree is a perfect Binary Tree with numbers of nodes like 1, 3, 7, 15, ...etc. In worst case, value of 2h is Ceil(n/2).

Height for a Balanced Binary Tree is O(Log n). Worst case occurs for skewed tree and worst case height becomes O(n).

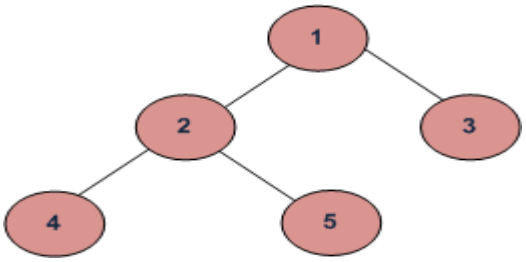
So in worst case extra space required is O(n) for both. But worst cases occur for different types of trees.
It is evident from above points that extra space required for Level order traversal is likely to be more when tree is more balanced and extra space for Depth First Traversal is likely to be more when tree is less balanced.

How to Pick One?
Extra Space can be one factor (Explained above)
Depth First Traversals are typically recursive and recursive code requires function call overheads.

The **most important points** is, BFS starts visiting nodes from root while DFS starts visiting nodes from leaves. So if our problem is to search something that is more likely to closer to root, we would prefer BFS. And if the target node is close to a leaf, we would prefer DFS.

7 Level Order Tree Traversal

http://www.geeksforgeeks.org/level-order-tree-traversal/
Level order traversal of a tree is breadth first traversal for the tree.



Example Tree
Level order traversal of the above tree is 1 2 3 4 5

METHOD 1 (Use function to print a given level)
Algorithm:
There are basically two functions in this method. One is to print all nodes at a given level (printGivenLevel), and other is to print level order traversal of the tree (printLevelorder). printLevelorder makes use of printGivenLevel to print nodes at all levels one by one starting from root.

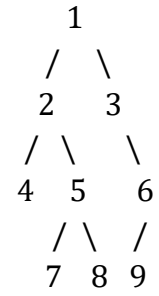
Time Complexity: O(n^2) in worst case. For a skewed tree, printGivenLevel() takes O(n) time where n is the number of nodes in the skewed tree. So time complexity of printLevelOrder() is O(n) + O(n-1) + O(n-2) + .. + O(1) which is O(n^2).

METHOD 2 (Use Queue)
Algorithm:
For each node, first the node is visited and then it's child nodes are put in a FIFO queue.
printLevelorder(tree)
1) Create an empty queue q
2) temp_node = root /*start from root*/
3) Loop while temp_node is not NULL
   a) print temp_node->data.
   b) Enqueue temp_node's children (first left then right children) to q
   c) Dequeue a node from q and assign it's value to temp_node

Time Complexity: O(n) where n is number of nodes in the binary tree

8 Print level order traversal line by line

http://quiz.geeksforgeeks.org/print-level-order-traversal-line-line/
Given a binary tree, print level order traversal in a way that nodes of all levels are printed in separate lines.
For example consider the following tree



Output for above tree should be
1
2 3
4 5 6
7 8 9
Note that this is different from simple level order traversal where we need to print all nodes together. Here we need to print nodes of different levels in different lines.
A simple solution is to print use the recursive function discussed in the level order traversal post and print a new line after every call to printGivenLevel().
The **time complexity** of the above solution is O(n2)

How to modify the iterative level order traversal (Method 2 of this) to levels line by line?
The **idea** is similar to this post. We count the nodes at current level. And for every node, we enqueue its children to queue.

```
// Iterative method to do level order traversal line by line
void printLevelOrder(node *root)
{
    // Base Case
    if (root == NULL)  return;

    // Create an empty queue for level order tarversal
    queue<node *> q;

    // Enqueue Root and initialize height
    q.push(root);
```

```
while (1)
{
    // nodeCount (queue size) indicates number of nodes
    // at current lelvel.
    int nodeCount = q.size();
    if (nodeCount == 0)
        break;

    // Dequeue all nodes of current level and Enqueue all
    // nodes of next level
    while (nodeCount > 0)
    {
        node *node = q.front();
        cout << node->data << " ";
        q.pop();
        if (node->left != NULL)
            q.push(node->left);
        if (node->right != NULL)
            q.push(node->right);
        nodeCount--;
    }
    cout << endl;
}}
```

Time complexity of this method is O(n) where n is number of nodes in given binary tree.

9 Inorder Tree Traversal without Recursion

http://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion/
Using Stack is the obvious way to traverse tree without recursion. Below is an algorithm for traversing binary tree using stack. See this for step wise step execution of the **algorithm**.

1) Create an empty stack S.
2) Initialize current node as root
3) Push the current node to S and set current = current->left until current is NULL
4) If current is NULL and stack is not empty then
   a) Pop the top item from stack.
   b) Print the popped item, set current = popped_item->right
   c) Go to step 3.
5) If current is NULL and stack is empty then we are done.

Time Complexity: O(n)

10 Inorder Tree Traversal without recursion and without stack!

http://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion-and-without-stack/

Using **Morris Traversal**, we can traverse the tree without using stack and recursion. The idea of Morris Traversal is based on Threaded Binary Tree. In this traversal, we first create links to Inorder successor and print the data using these links, and finally revert the changes to restore original tree.

1. Initialize current as root
2. While current is not NULL
   If current does not have left child
   a) Print current's data
   b) Go to the right, i.e., current = current->right
   Else
   a) Make current as right child of the rightmost node in current's left subtree
   b) Go to this left child, i.e., current = current->left

Although the tree is modified through the traversal, it is reverted back to its original shape after the completion. Unlike Stack based traversal, no extra space is required for this traversal.

```
/* Function to traverse binary tree without recursion and
   without stack */
void MorrisTraversal(struct tNode *root)
{
  struct tNode *current,*pre;

  if(root == NULL)
     return;

  current = root;
  while(current != NULL)
  {
    if(current->left == NULL)
    {
      printf("%d ", current->data);
      current = current->right;
    }
    else
    {
      /* Find the inorder predecessor of current */
      pre = current->left;
      while(pre->right != NULL && pre->right != current)
```

```
        pre = pre->right;

      /* Make current as right child of its inorder predecessor */
      if(pre->right == NULL)
      {
        pre->right = current;
        current = current->left;
      }

      /* Revert the changes made in if part to restore the original
        tree i.e., fix the right child of predecssor */
      else
      {
        pre->right = NULL;
        printf("%d ",current->data);
        current = current->right;
      } /* End of if condition pre->right == NULL */
    } /* End of if condition current->left == NULL*/
  } /* End of while */
```

11 Threaded Binary Tree

http://quiz.geeksforgeeks.org/threaded-binary-tree/

12 Write a program to Calculate Size of a tree

http://www.geeksforgeeks.org/write-a-c-program-to-calculate-size-of-a-tree/

Size() function recursively calculates the size of a tree. It works as follows:

Size of a tree = Size of left subtree + 1 + Size of right subtree

Algorithm:

size(tree)

1. If tree is empty then return 0
2. Else
   (a) Get the size of left subtree recursively i.e., call
   size(tree->left-subtree)
   (a) Get the size of right subtree recursively i.e., call
   size(tree->right-subtree)
   (c) Calculate size of the tree as following:
   tree_size = size(left-subtree) + size(right-subtree) + 1
   (d) Return tree_size

13 Write Code to Determine if Two Trees are Identical

http://www.geeksforgeeks.org/write-c-code-to-determine-if-two-trees-are-identical/
Two trees are identical when they have same data and arrangement of data is also same.

To identify if two trees are identical, we need to traverse both trees simultaneously, and while traversing we need to compare data and children of the trees.

```
/* Given two trees, return true if they are
 structurally identical */
int identicalTrees(struct node* a, struct node* b)
{
    /*1. both empty */
    if (a==NULL && b==NULL)
        return 1;

    /* 2. both non-empty -> compare them */
    if (a!=NULL && b!=NULL)
    {
        return
        (
            a->data == b->data &&
            identicalTrees(a->left, b->left) &&
            identicalTrees(a->right, b->right)
        );
    }

    /* 3. one empty, one not -> false */
    return 0;
}
```

Time Complexity of the identicalTree() will be according to the tree with lesser number of nodes. Let number of nodes in two trees be m and n then complexity of sameTree() is O(m) where m < n. http://www.geeksforgeeks.org/iterative-function-check-two-trees-identical/

14 Write a Program to Find the Maximum Depth or Height of a Tree

http://www.geeksforgeeks.org/write-a-c-program-to-find-the-maximum-depth-or-height-of-a-tree/

Given a binary tree, find height of it. Height of empty tree is
Recursively calculate height of left and right subtrees of a node and assign height to the node as max of the heights of two children plus 1. See below pseudo code and program for details.

```
int maxDepth(struct node* node)
{
   if (node==NULL)
       return 0;
   else
   {
       /* compute the depth of each subtree */
       int lDepth = maxDepth(node->left);
       int rDepth = maxDepth(node->right);

       /* use the larger one */
       if (lDepth > rDepth)
           return(lDepth+1);
       else return(rDepth+1);
   }
}
```

15 Write a program to Delete a Tree.

http://www.geeksforgeeks.org/write-a-c-program-to-delete-a-tree/

To delete a tree we must traverse all the nodes of the tree and delete them one by one. So which traversal we should use – Inorder or Preorder or Postorder.

Answer is **simple – Postorder**, because before deleting the parent node we should delete its children nodes first

We can **delete tree** with other traversals also with extra space complexity but why should we go for other traversals if we have Postorder available which does the work without storing anything in same time complexity.

```
/*  This function traverses tree in post order to
    to delete each and every node of the tree */
void deleteTree(struct node* node)
{
    if (node == NULL) return;

    /* first delete both subtrees */
    deleteTree(node->left);
    deleteTree(node->right);

    /* then delete the node */
    printf("\n Deleting node: %d", node->data);
    free(node);}
```

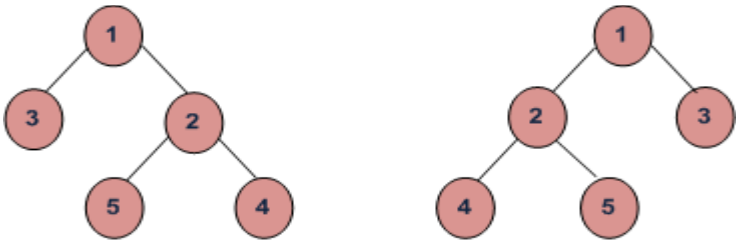
The above **deleteTree()** function deletes the tree, but doesn't change root to NULL which may cause problems if the user of deleteTree() doesn't change root to NULL and tires to access values using root pointer. We can modify the deleteTree() function to take reference to the root node so that this problem doesn't occur. See the following code.

```
/*  This function is same as deleteTree() in the previous program */
void _deleteTree(struct node* node)
{
    if (node == NULL) return;

    /* first delete both subtrees */
    _deleteTree(node->left);
    _deleteTree(node->right);

    /* then delete the node */
    printf("\n Deleting node: %d", node->data);
    free(node);
}

/* Deletes a tree and sets the root as NULL */
void deleteTree(struct node** node_ref)
{
  _deleteTree(*node_ref);
  *node_ref = NULL;}
```

16 Write an Efficient Function to Convert a Binary Tree into its Mirror Tree

http://www.geeksforgeeks.org/write-an-efficient-c-function-to-convert-a-tree-into-its-mirror-tree/

Mirror of a Tree: Mirror of a Binary Tree T is another Binary Tree M(T) with left and right children of all non-leaf nodes interchanged.



Mirror Trees

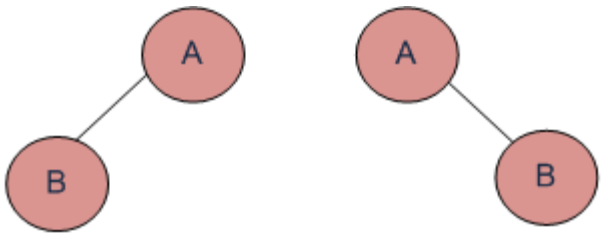
Algorithm – Mirror(tree):

(1) Call Mirror for left-subtree i.e., Mirror(left-subtree)
(2) Call Mirror for right-subtree i.e., Mirror(right-subtree)
(3) Swap left and right subtrees.
    temp = left-subtree
    left-subtree = right-subtree
    right-subtree = temp

17 If you are given two traversal sequences, can you construct the binary tree?

http://www.geeksforgeeks.org/if-you-are-given-two-traversal-sequences-can-you-construct-the-binary-tree/

It depends on what traversals are given. If one of the traversal methods is Inorder then the tree can be constructed, otherwise not.



Trees having Preorder, Postorder and Level-Order and traversals

Therefore, following combination can uniquely identify a tree.

Inorder and Preorder.
Inorder and Postorder.
Inorder and Level-order.

And following do not.

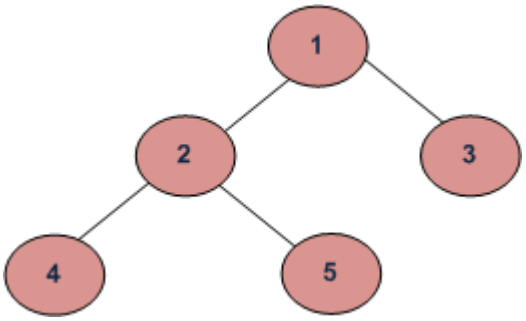
Postorder and Preorder.
Preorder and Level-order.
Postorder and Level-order.

For example, Preorder, Level-order and Postorder traversals are same for the trees given in above diagram.
Preorder Traversal = AB
Postorder Traversal = BA
Level-Order Traversal = AB
So, even if three of them (Pre, Post and Level) are given, the tree can not be constructed.

18 Given a binary tree, print out all of its root-to-leaf paths one per line.

http://www.geeksforgeeks.org/given-a-binary-tree-print-out-all-of-its-root-to-leaf-paths-one-per-line/

Example:



Example Tree

Output for the above example will be

1 2 4
1 2 5
1 3

Algorithm:

initialize: pathlen = 0, path[1000]
/*1000 is some max limit for paths, it can change*/

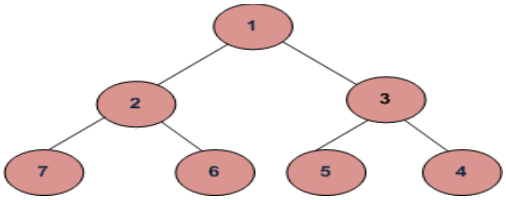
/*printPathsRecur traverses nodes of tree in preorder */

printPathsRecur(tree, path[], pathlen)

1) If node is not NULL then
   a) push data to path array:
      path[pathlen] = node->data.
   b) increment pathlen
      pathlen++
2) If node is a leaf node then print the path array.
3) Else
   a) Call printPathsRecur for left subtree
      printPathsRecur(node->left, path, pathLen)
   b) Call printPathsRecur for right subtree.
      printPathsRecur(node->right, path, pathLen)

19 Level order traversal in spiral form

http://www.geeksforgeeks.org/level-order-traversal-in-spiral-form/
Write a function to print spiral order traversal of a tree. For below tree, function should print 1, 2, 3, 4, 5, 6, 7.



Method 1 (Recursive)
This problem can bee seen as an extension of the level order traversal post.
To print the nodes in spiral order, nodes at different levels should be printed in alternating order. An additional Boolean variable ltr is used to change printing order of levels. If ltr is 1 then printGivenLevel() prints nodes from left to right else from right to left. Value of ltr is flipped in each iteration to change the order.
Function to print level order traversal of tree

printSpiral(tree)

```
bool ltr = 0;
for d = 1 to height(tree)
   printGivenLevel(tree, d, ltr);
   ltr ~= ltr /*flip ltr*/
```

Function to print all nodes at a given level

```
printGivenLevel(tree, level, ltr)
if tree is NULL then return;
if level is 1, then
   print(tree->data);
else if level greater than 1, then
   if(ltr)
      printGivenLevel(tree->left, level-1, ltr);
      printGivenLevel(tree->right, level-1, ltr);
   else
      printGivenLevel(tree->right, level-1, ltr);
      printGivenLevel(tree->left, level-1, ltr);
```

Time Complexity: Worst case time complexity of the above method is O(n^2). Worst case occurs in case of skewed trees.

Method 2 (Iterative)

We can print spiral order traversal in O(n) time and O(n) extra space. The idea is to use two stacks. We can use one stack for printing from left to right and other stack for printing from right to left. In every iteration, we have nodes of one level in one of the stacks. We print the nodes, and push nodes of next level in other stack.

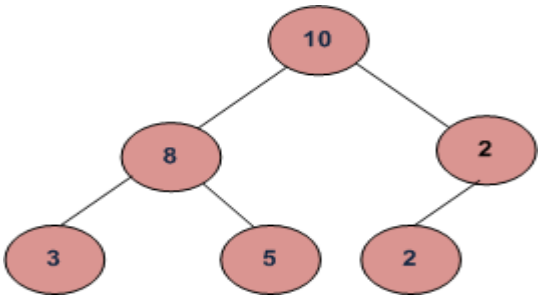
```
void printSpiral(struct node *root)
{
    if (root == NULL)  return;   // NULL check

    // Create two stacks to store alternate levels
    stack<struct node*> s1;  // For levels to be printed from right to
left
    stack<struct node*> s2;  // For levels to be printed from left to
right

    // Push first level to first stack 's1'
    s1.push(root);

    // Keep ptinting while any of the stacks has some nodes
    while (!s1.empty() || !s2.empty())
    {
        // Print nodes of current level from s1 and push nodes of
        // next level to s2
        while (!s1.empty())
        {
            struct node *temp = s1.top();
            s1.pop();
            cout << temp->data << " ";

            // Note that is right is pushed before left
            if (temp->right)
                s2.push(temp->right);
            if (temp->left)
                s2.push(temp->left);
        }

        // Print nodes of current level from s2 and push nodes of
        // next level to s1
        while (!s2.empty())
        {
            struct node *temp = s2.top();
            s2.pop();
            cout << temp->data << " ";

            // Note that is left is pushed before right
            if (temp->left)
                s1.push(temp->left);
            if (temp->right)
                s1.push(temp->right);
        }
}    }
```

20 Check for Children Sum Property in a Binary Tree

http://www.geeksforgeeks.org/check-for-children-sum-property-in-a-binary-tree/

Given a binary tree, write a function that returns true if the tree satisfies below property.

For every node, data value must be equal to sum of data values in left and right children. Consider data value as 0 for NULL children. Below tree is an example



Algorithm:

Traverse the given binary tree. For each node check (recursively) if the node and both its children satisfy the Children Sum Property, if so then return true else return false.

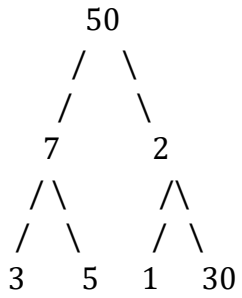
```
int isSumProperty(struct node* node)
{
  /* left_data is left child data and right_data is for right
     child data*/
  int left_data = 0,  right_data = 0;
   /* If node is NULL or it's a leaf node then
      return true */
  if(node == NULL ||
      (node->left == NULL && node->right == NULL))
    return 1;
  else
  {
    /* If left child is not present then 0 is used
       as data of left child */
    if(node->left != NULL)
      left_data = node->left->data;
     /* If right child is not present then 0 is used
      as data of right child */
    if(node->right != NULL)
      right_data = node->right->data;
     /* if the node and both of its children satisfy the
        property return 1 else 0*/
    if((node->data == left_data + right_data)&&
        isSumProperty(node->left) &&
        isSumProperty(node->right))
      return 1;
    else
      return 0;
  }}}
```

21 Convert an arbitrary Binary Tree to a tree that holds Children Sum Property

http://www.geeksforgeeks.org/convert-an-arbitrary-binary-tree-to-a-tree-that-holds-children-sum-property/

Given an arbitrary binary tree, convert it to a binary tree that holds Children Sum Property. You can only increment data values in any node (You cannot change structure of tree and cannot decrement value of any node).

For example, the below tree doesn't hold the children sum property, convert it to a tree that holds the property.



Algorithm:

Traverse given tree in post order to convert it, i.e., first change left and right children to hold the children sum property then change the parent node.

Let difference between node's data and children sum be diff.

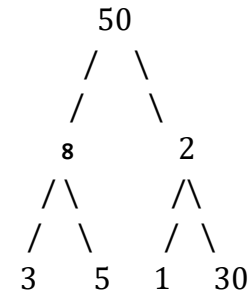
diff = node's children sum - node's data

If diff is 0 then nothing needs to be done.

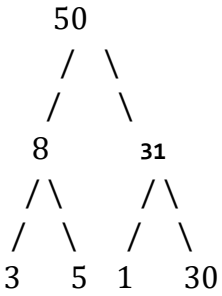
If diff > 0 (node's data is smaller than node's children sum) increment the node's data by diff.

If diff < 0 (node's data is greater than the node's children sum) then increment one child's data. We can choose to increment either left or right child if they both are not NULL.

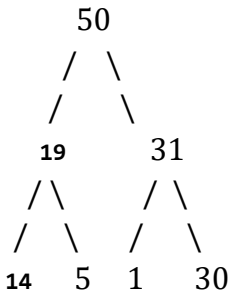
Let us always first increment the left child. Incrementing a child changes the subtree's children sum property so we need to change left subtree also. So we recursively increment the left child. If left child is empty then we recursively call increment() for right child. Let us run the algorithm for the given example. First convert the left subtree (increment 7 to 8).



Then convert the right subtree (increment 2 to 31)



Now convert the root, we have to increment left subtree for converting the root.



Please note the last step – we have incremented 8 to 19, and to fix the subtree we have incremented 3 to 14.

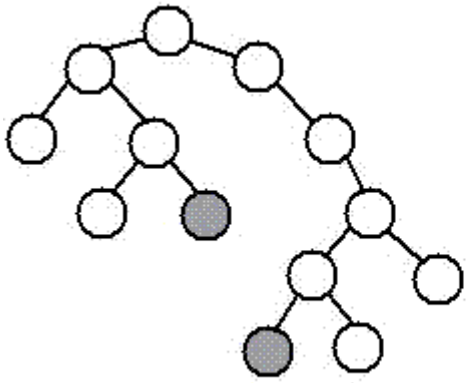
```
void convertTree(struct node* node)
{
  int left_data = 0,  right_data = 0, diff;

  /* If tree is empty or it's a leaf node then
     return true */
  if (node == NULL ||
     (node->left == NULL && node->right == NULL))
    return;
  else
  {
    /* convert left and right subtrees  */
    convertTree(node->left);
    convertTree(node->right);

    /* If left child is not present then 0 is used
```

```
       as data of left child */
    if (node->left != NULL)
      left_data = node->left->data;

    /* If right child is not present then 0 is used
      as data of right child */
    if (node->right != NULL)
      right_data = node->right->data;

    /* get the diff of node's data and children sum */
    diff = left_data + right_data - node->data;

    /* If node's children sum is greater than the node's data */
    if (diff > 0)
       node->data = node->data + diff;

    /* THIS IS TRICKY --> If node's data is greater than children sum,
      then increment subtree by diff */
    if (diff < 0)
      increment(node, -diff);  // -diff is used to make diff positive
  }
}

/* This function is used to increment subtree by diff */
void increment(struct node* node, int diff)
{
  /* IF left child is not NULL then increment it */
  if(node->left != NULL)
  {
    node->left->data = node->left->data + diff;

    // Recursively call to fix the descendants of node->left
    increment(node->left, diff);
  }
  else if (node->right != NULL) // Else increment right child
  {
    node->right->data = node->right->data + diff;

    // Recursively call to fix the descendants of node->right
    increment(node->right, diff);
  }}
```

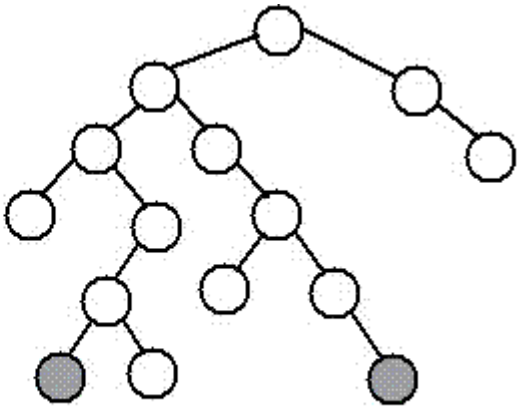
22 Diameter of a Binary Tree

http://www.geeksforgeeks.org/diameter-of-a-binary-tree/

The diameter of a tree (sometimes called the width) is the number of nodes on the longest path between two leaves in the tree. The diagram below shows two trees each with diameter nine, the leaves that form the ends of a longest path are shaded (note that there is more than one path in each tree of length nine, but no path longer than nine nodes).



diameter, 9 nodes, through root



diameter, 9 nodes, NOT through root

The diameter of a tree T is the largest of the following quantities:
* the diameter of T's left subtree
* the diameter of T's right subtree
* the longest path between leaves that goes through the root of T (this can be computed from the heights of the subtrees of T)

Time Complexity: O(n^2)

Optimized implementation: The above implementation can be optimized by calculating the height in the same recursion rather than calling a height() separately. Thanks to Amar for suggesting this optimized version. This optimization reduces **time complexity** to O(n).

```
/*The second parameter is to store the height of tree.
   Initially, we need to pass a pointer to a location with value
   as 0. So, function should be used as follows:

   int height = 0;
   struct node *root = SomeFunctionToMakeTree();
   int diameter = diameterOpt(root, &height); */
int diameterOpt(struct node *root, int* height)
```

```
{
  /* lh --> Height of left subtree
     rh --> Height of right subtree */
  int lh = 0, rh = 0;

  /* ldiameter  --> diameter of left subtree
     rdiameter  --> Diameter of right subtree */
  int ldiameter = 0, rdiameter = 0;

  if(root == NULL)
  {
    *height = 0;
     return 0; /* diameter is also 0 */
  }

  /* Get the heights of left and right subtrees in lh and rh
    And store the returned values in ldiameter and ldiameter */
  ldiameter = diameterOpt(root->left, &lh);
  rdiameter = diameterOpt(root->right, &rh);

  /* Height of current node is max of heights of left and
     right subtrees plus 1*/
  *height = max(lh, rh) + 1;

  return max(lh + rh + 1, max(ldiameter, rdiameter));
}
```

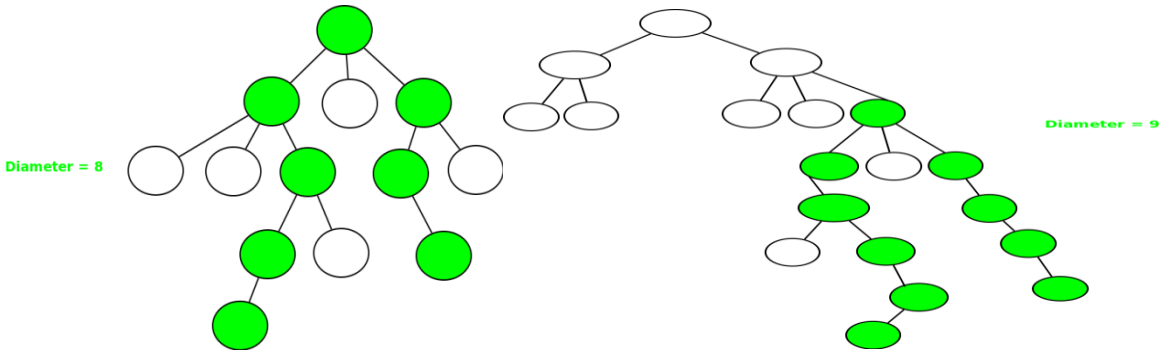
Time Complexity: O(n)

http://www.geeksforgeeks.org/diameter-n-ary-tree/

23 Diameter of an N-ary tree

http://www.geeksforgeeks.org/diameter-n-ary-tree/

The diameter of an N-ary tree is the longest path present between any two nodes of the tree. These two nodes must be two leaf nodes. The following examples have the longest path[diameter] shaded.



Examples:1 Example 2

The path can either start from one of the node and goes up to one of the LCAs of these nodes and again come down to the deepest node of some other subtree **or** can exist as a diameter of one of the child of the current node.

The solution will exist in any one of these:
I] Diameter of one of the children of the current node
II] Sum of Height of highest two subtree + 1

```
// Function to calculate the diameter
// of the tree
int diameter(struct Node *ptr)
{
    // Base case
    if (!ptr)
        return 0;

    // Find top two highest children
    int max1 = 0, max2 = 0;
    for (vector<Node*>::iterator it = ptr->child.begin();
                          it != ptr->child.end(); it++)
    {
        int h = depthOfTree(*it);
        if (h > max1)
           max2 = max1, max1 = h;
        else if (h > max2)
           max2 = h;
    }

    // Iterate over each child for diameter
    int maxChildDia = 0;
    for (vector<Node*>::iterator it = ptr->child.begin();
                           it != ptr->child.end(); it++)
        maxChildDia = max(maxChildDia, diameter(*it));

    return max(maxChildDia, max1 + max2 + 1);
}
```

Optimizations to above solution :
We can make a hash table to store heights of all nodes. If we precompute these heights, we don't need to call depthOfTree() for every node.

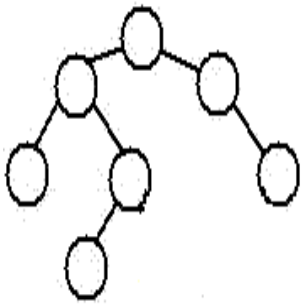
A different optimized solution :
Longest path in an undirected tree
http://www.geeksforgeeks.org/longest-path-undirected-tree/

24 How to determine if a binary tree is height-balanced?

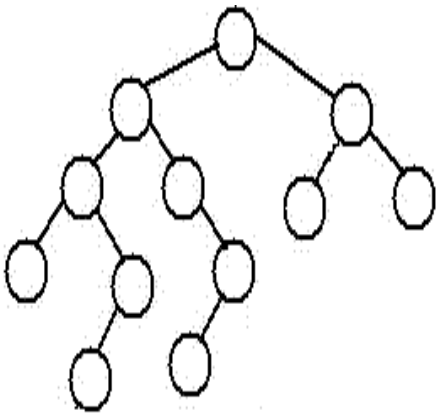
http://www.geeksforgeeks.org/how-to-determine-if-a-binary-tree-is-balanced/
A tree where no leaf is much farther away from the root than any other leaf. Different balancing schemes allow different definitions of "much farther" and different amounts of work to keep them balanced.

Consider a **height-balancing** scheme where following conditions should be checked to determine if a binary tree is balanced.
An empty tree is height-balanced. A non-empty binary tree T is balanced if:
1) Left subtree of T is balanced
2) Right subtree of T is balanced
3) The difference between heights of left subtree and right subtree is not more than 1.

The above height-balancing scheme is used in AVL trees. The diagram below shows two trees, one of them is height-balanced and other is not. The second tree is not height-balanced because height of left subtree is 2 more than height of right subtree.



A height-balanced Tree



Not a height-balanced tree

To check if a tree is height-balanced, get the height of left and right subtrees. Return true if difference between heights is not more than 1 and left and right subtrees are balanced, otherwise return false.

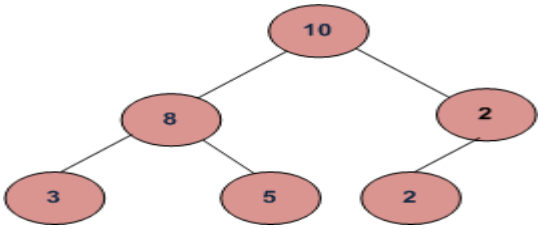
Time Complexity: O(n^2) Worst case occurs in case of skewed tree.

Optimized implementation:
Above implementation can be optimized by calculating the height in the same recursion rather than calling a height() function separately. Thanks to Amar for suggesting this optimized version.
This optimization reduces **time complexity** to O(n).

```
/* The function returns true if root is balanced else false
   The second parameter is to store the height of tree.
   Initially, we need to pass a pointer to a location with value
   as 0. We can also write a wrapper over this function */
bool isBalanced(struct node *root, int* height)
{
  /* lh --> Height of left subtree
     rh --> Height of right subtree */
  int lh = 0, rh = 0;

  /* l will be true if left subtree is balanced
    and r will be true if right subtree is balanced */
  int l = 0, r = 0;

  if(root == NULL)
  {
    *height = 0;
     return 1;
  }

  /* Get the heights of left and right subtrees in lh and rh
    And store the returned values in l and r */
  l = isBalanced(root->left, &lh);
  r = isBalanced(root->right,&rh);

  /* Height of current node is max of heights of left and
     right subtrees plus 1*/
  *height = (lh > rh? lh: rh) + 1;

  /* If difference between heights of left and right
     subtrees is more than 2 then this node is not balanced
     so return 0 */
  if((lh - rh >= 2) || (rh - lh >= 2))
    return 0;

  /* If this node is balanced and left and right subtrees
    are balanced then return true */
  else return l&&r;
}
```

25 Root to leaf path sum equal to a given number

http://www.geeksforgeeks.org/root-to-leaf-path-sum-equal-to-a-given-number/
Given a binary tree and a number, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals the given number. Return false if no such path can be found.



For example, in the above tree root to leaf paths exist with following sums.
21 –> 10 – 8 – 3
23 –> 10 – 8 – 5
14 –> 10 – 2 – 2
So the returned value should be true only for numbers 21, 23 and 14. For any other number, returned value should be false.

Algorithm:
Recursively check if left or right child has path sum equal to (number – value at current node)

```
bool hasPathSum(struct node* node, int sum)
{
  if (node == NULL)
  {
     return (sum == 0);
  }
  else
  {
    bool ans = 0;

    /* otherwise check both subtrees */
    int subSum = sum - node->data;
      /* If we reach a leaf node and sum becomes 0 then return true*/
    if ( subSum == 0 && node->left == NULL && node->right == NULL )
      return 1;
      if(node->left)
      ans = ans || hasPathSum(node->left, subSum);
    if(node->right)
      ans = ans || hasPathSum(node->right, subSum);

    return ans;
  }}
```

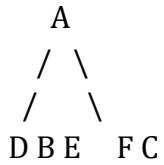
Time Complexity: O(n)

26 Construct Tree from given Inorder and Preorder traversals

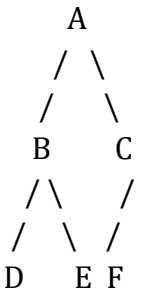
http://www.geeksforgeeks.org/construct-tree-from-given-inorder-and-preorder-traversal/

Let us consider the below traversals:
Inorder sequence: D B E A F C
Preorder sequence: A B D E C F

In a **Preorder sequence**, leftmost element is the root of the tree. So we know 'A' is root for given sequences. By searching 'A' in Inorder sequence, we can find out all elements on left side of 'A' are in left subtree and elements on right are in right subtree. So we know below structure now.



We recursively follow above steps and get the following tree.



Algorithm: buildTree()

1) Pick an element from Preorder. Increment a Preorder Index Variable (preIndex in below code) to pick next element in next recursive call.
2) Create a new tree node tNode with the data as picked element.
3) Find the picked element's index in Inorder. Let the index be inIndex.
4) Call buildTree for elements before inIndex and make the built tree as left subtree of tNode.
5) Call buildTree for elements after inIndex and make the built tree as right subtree of tNode.
6) return tNode.

```
/* Recursive function to construct binary of size len from
   Inorder traversal in[] and Preorder traversal pre[].  Initial values
   of inStrt and inEnd should be 0 and len -1.  The function doesn't
   do any error checking for cases where inorder and preorder
   do not form a tree */
struct node* buildTree(char in[], char pre[], int inStrt, int inEnd)
{
  static int preIndex = 0;

  if(inStrt > inEnd)
     return NULL;

  /* Pick current node from Preorder traversal using preIndex
    and increment preIndex */
  struct node *tNode = newNode(pre[preIndex++]);

  /* If this node has no children then return */
  if(inStrt == inEnd)
    return tNode;

  /* Else find the index of this node in Inorder traversal */
  int inIndex = search(in, inStrt, inEnd, tNode->data);

  /* Using index in Inorder traversal, construct left and
     right subtress */
  tNode->left = buildTree(in, pre, inStrt, inIndex-1);
  tNode->right = buildTree(in, pre, inIndex+1, inEnd);

  return tNode;
}
```

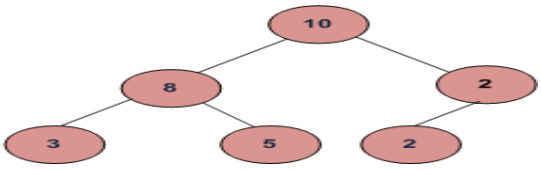
27 Given a binary tree, print all root-to-leaf paths

http://www.geeksforgeeks.org/given-a-binary-tree-print-all-root-to-leaf-paths/

For the below example tree, all root-to-leaf paths are:

10 –> 8 –> 3

10 –> 8 –> 5



Algorithm:

Use a path array path[] to store current root to leaf path. Traverse from root to all leaves in top-down fashion. While traversing, store data of all nodes in current path in array path[]. When we reach a leaf node, print the path array.

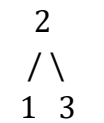
Time Complexity: O(n)

28 Double Tree

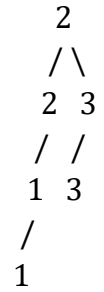
http://www.geeksforgeeks.org/double-tree/

Write a program that converts a given tree to its Double tree. To create Double tree of the given tree, create a new duplicate for each node, and insert the duplicate as the left child of the original node.

So the tree...



is changed to...



Algorithm:

Recursively convert the tree to double tree in postorder fashion. For each node, first convert the left subtree of the node, then right subtree, finally create a duplicate node of the node and fix the left child of the node and left child of left child.

```
/* Function to convert a tree to double tree */
void doubleTree(struct node* node)
{
  struct node* oldLeft;

  if (node==NULL) return;

  /* do the subtrees */
  doubleTree(node->left);
  doubleTree(node->right);

  /* duplicate this node to its left */
  oldLeft = node->left;
  node->left = newNode(node->data);
  node->left->left = oldLeft;
}
```

Time Complexity: O(n) where n is the number of nodes in the tree.

29 Maximum width of a binary tree

http://www.geeksforgeeks.org/maximum-width-of-a-binary-tree/

Given a binary tree, write a function to get the maximum width of the given tree. Width of a tree is maximum of widths of all levels.

Let us consider the below example tree.

```
      1
     / \
    2   3
   / \   \
  4   5   8
```

width of level 1 is 1,
width of level 2 is 2,
width of level 3 is 3
So the maximum width of the tree is 3.

Method 1 (Using Level Order Traversal)

This method mainly involves two functions. One is to count nodes at a given level (getWidth), and other is to get the maximum width of the tree(getMaxWidth). getMaxWidth() makes use of getWidth() to get the width of all levels starting from root.

```
getMaxWidth(tree)
maxWdth = 0
for i = 1 to height(tree)
 width =   getWidth(tree, i);
 if(width > maxWdth)
    maxWdth  = width
return width
```

/*Function to get width of a given level */

```
getWidth(tree, level)
if tree is NULL then return 0;
if level is 1, then return 1;
else if level greater than 1, then
   return getWidth(tree->left, level-1) +
   getWidth(tree->right, level-1);
```

Time Complexity: O(n^2) in the worst case.

We can use **Queue** based level order traversal to optimize the time complexity of this method. The Queue based level order traversal will take O(n) time in worst case.

Method 2 (Using Level Order Traversal with Queue)

In this method we store all the child nodes at the current level in the queue and then count the total number of nodes after the level order traversal for a particular level is completed. Since the queue now contains all the nodes of the next level, we can easily find out the total number of nodes in the next level by finding the size of queue. We then follow the same procedure for the successive levels. We store and update the maximum number of nodes found at each level.

Method 3 (Using Preorder Traversal)

In this method we create a temporary array count[] of size equal to the height of tree. We initialize all values in count as 0. We traverse the tree using preorder traversal and fill the entries in count so that the count array contains count of nodes at each level in Binary Tree

```
// A function that fills count array with count of nodes at every
// level of given binary tree
void getMaxWidthRecur(struct node *root, int count[], int level);

/* Function to get the maximum width of a binary tree*/
int getMaxWidth(struct node* root)
{
  int width;
  int h = height(root);

  // Create an array that will store count of nodes at each level
  int *count = (int *)calloc(sizeof(int), h);
   int level = 0;
   // Fill the count array using preorder traversal
  getMaxWidthRecur(root, count, level);
   // Return the maximum value from count array
  return getMax(count, h);
}
 // A function that fills count array with count of nodes at every
// level of given binary tree
void getMaxWidthRecur(struct node *root, int count[], int level)
{
  if(root)
  {
    count[level]++;
    getMaxWidthRecur(root->left, count, level+1);
    getMaxWidthRecur(root->right, count, level+1);
  }
}
```

Time Complexity: O(n)

30 Foldable Binary Trees

http://www.geeksforgeeks.org/foldable-binary-trees/

Question: Given a binary tree, find out if the tree can be folded or not.

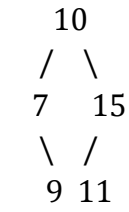
A tree can be folded if left and right subtrees of the tree are structure wise mirror image of each other. An empty tree is considered as foldable.

Consider the below trees:

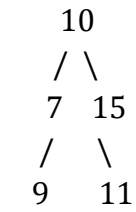
(a) and (b) can be folded.

(c) and (d) cannot be folded.

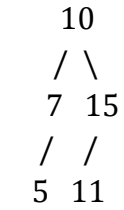
(a)



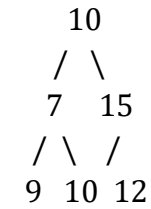
(b)



(c)



(d)



Method 1 (Change Left subtree to its Mirror and compare it with Right subtree)

Algorithm: isFoldable(root)

1) If tree is empty, then return true.
2) Convert the left subtree to its mirror image
   mirror(root->left); /* See this post */
3) Check if the structure of left subtree and right subtree is same
  and store the result.
   res = isStructSame(root->left, root->right); /*isStructSame()
      recursively compares structures of two subtrees and returns
      true if structures are same */
4) Revert the changes made in step (2) to get the original tree.
   mirror(root->left);
5) Return result res stored in step 2.

```
bool isStructSame(struct node *a, struct node *b)
{
  if (a == NULL && b == NULL)
  {  return true; }
  if ( a != NULL && b != NULL &&
       isStructSame(a->left, b->left) &&
       isStructSame(a->right, b->right)
     )
  {  return true; }

  return false;
}
```

Time complexity: O(n)

Method 2 (Check if Left and Right subtrees are Mirror)

There are mainly two functions:

// Checks if tree can be folded or not
IsFoldable(root)

1) If tree is empty then return true
2) Else check if left and right subtrees are structure wise mirrors of
   each other. Use utility function IsFoldableUtil(root->left,
   root->right) for this.

// Checks if n1 and n2 are mirror of each other.

IsFoldableUtil(n1, n2)

1) If both trees are empty then return true.
2) If one of them is empty and other is not then return false.
3) Return true if following conditions are met
   a) n1->left is mirror of n2->right
   b) n1->right is mirror of n2->left

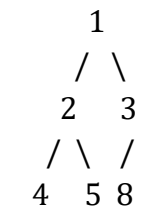
```
/* A utility function that checks if trees with roots as n1 and n2
  are mirror of each other */
bool IsFoldableUtil(struct node *n1, struct node *n2)
{
    /* If both left and right subtrees are NULL,
      then return true */
    if (n1 == NULL && n2 == NULL)
    {  return true;  }

    /* If one of the trees is NULL and other is not,
      then return false */
    if (n1 == NULL || n2 == NULL)
    {  return false; }

    /* Otherwise check if left and right subtrees are mirrors of
       their counterparts */
    return IsFoldableUtil(n1->left, n2->right) &&
           IsFoldableUtil(n1->right, n2->left);
}
```

31 Print nodes at k distance from root

http://www.geeksforgeeks.org/print-nodes-at-k-distance-from-root/

Given a root of a tree, and an integer k. Print all the nodes which are at k distance from root.

For example, in the below tree, 4, 5 & 8 are at distance 2 from root.

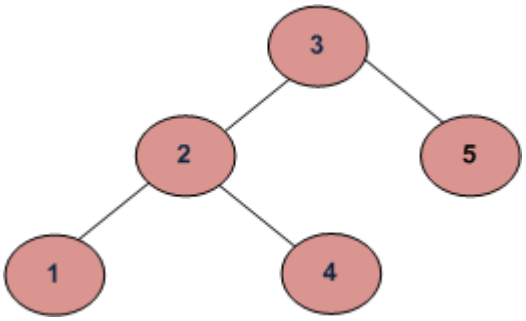


Time Complexity: O(n) where n is number of nodes in the given binary tree.

32 Get Level of a node in a Binary Tree

http://www.geeksforgeeks.org/get-level-of-a-node-in-a-binary-tree/

Given a Binary Tree and a key, write a function that returns level of the key.
For example, consider the following tree. If the input key is 3, then your function should return 1. If the input key is 4, then your function should return 3. And for key which is not present in key, then your function should return 0.



The **idea** is to start from the root and level as 1. If the key matches with root's data, return level. Else recursively call for left and right subtrees with level as level + 1.

```
/* Helper function for getLevel().  It returns level of the data if data
is
   present in tree, otherwise returns 0.*/
int getLevelUtil(struct node *node, int data, int level)
{
    if (node == NULL)
        return 0;

    if (node->data == data)
        return level;

    int downlevel = getLevelUtil(node->left, data, level+1);
    if (downlevel != 0)
        return downlevel;

    downlevel = getLevelUtil(node->right, data, level+1);
    return downlevel;
}

/* Returns level of given data value */
int getLevel(struct node *node, int data)
{
    return getLevelUtil(node,data,1);}
```

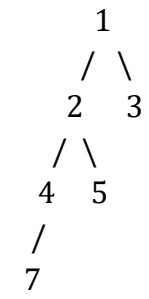
Time Complexity: O(n) where n is the number of nodes in the given Binary Tree.

33 Print Ancestors of a given node in Binary Tree

http://www.geeksforgeeks.org/print-ancestors-of-a-given-node-in-binary-tree/

Given a Binary Tree and a key, write a function that prints all the ancestors of the key in the given binary tree.

For example, if the given tree is following Binary Tree and key is 7, then your function should print 4, 2 and 1.



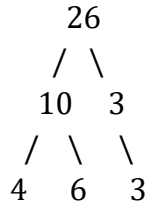

```
/* If target is present in tree, then prints the ancestors
   and returns true, otherwise returns false. */
bool printAncestors(struct node *root, int target)
{
  /* base cases */
  if (root == NULL)
     return false;

  if (root->data == target)
     return true;

  /* If target is present in either left or right subtree of this node,
     then print this node */
  if ( printAncestors(root->left, target) ||
       printAncestors(root->right, target) )
  {
    cout << root->data << " ";
    return true;
  }

  /* Else return false */
  return false;
}
```

Time Complexity: O(n) where n is the number of nodes in the given Binary Tree.

34 Check if a given Binary Tree is SumTree

http://www.geeksforgeeks.org/check-if-a-given-binary-tree-is-sumtree/

Write a function that returns true if the given Binary Tree is SumTree else false. A SumTree is a Binary Tree where the value of a node is equal to sum of the nodes present in its left subtree and right subtree. An empty tree is SumTree and sum of an empty tree can be considered as 0. A leaf node is also considered as SumTree. Following is an example of SumTree.



Method 1 (Simple)

Get the sum of nodes in left subtree and right subtree. Check if the sum calculated is equal to root's data. Also, recursively check if the left and right subtrees are SumTrees.
Time Complexity: O(n^2) in worst case. Worst case occurs for a skewed tree.

Method 2 (Tricky)

The Method 1 uses sum() to get the sum of nodes in left and right subtrees. The method 2 uses following rules to get the sum directly.
1) If the node is a leaf node then sum of subtree rooted with this node is equal to value of this node.
2) If the node is not a leaf node then sum of subtree rooted with this node is twice the value of this node (Assuming that the tree rooted with this node is SumTree).

```
/* returns 1 if SumTree property holds for the given
    tree */
int isSumTree(struct node* node)
{
    int ls; // for sum of nodes in left subtree
    int rs; // for sum of nodes in right subtree
     /* If node is NULL or it's a leaf node then
       return true */
    if(node == NULL || isLeaf(node))
        return 1;
     if( isSumTree(node->left) && isSumTree(node->right))
    {
        // Get the sum of nodes in left subtree
        if(node->left == NULL)
            ls = 0;
        else if(isLeaf(node->left))
            ls = node->left->data;
```

```
        else
            ls = 2*(node->left->data);

        // Get the sum of nodes in right subtree
        if(node->right == NULL)
            rs = 0;
        else if(isLeaf(node->right))
            rs = node->right->data;
        else
            rs = 2*(node->right->data);

        /* If root's data is equal to sum of nodes in left
           and right subtrees then return 1 else return 0*/
        return(node->data == ls + rs);
    }

    return 0;}
```

Time Complexity: O(n)

35 Check if a binary tree is subtree of another binary tree

http://www.geeksforgeeks.org/check-if-a-binary-tree-is-subtree-of-another-binary-tree/

Given two binary trees, check if the first tree is subtree of the second one. A subtree of a tree T is a tree S consisting of a node in T and all of its descendants in T. The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree.

For example, in the following case, tree S is a subtree of tree T.

```
Tree 2
  10
 /  \
4    6
 \
  30
Tree 1
     26
    /  \
   10   3
  /  \   \
 4    6   3
  \
   30
```

Solution: Traverse the tree T in preorder fashion. For every visited node in the traversal, see if the subtree rooted with this node is identical to S.
Time Complexity: Time worst case complexity of above solution is O(mn) where m and n are number of nodes in given two trees.
We can solve the above problem in **O(n) time**

The **idea** is based on the fact that inorder and preorder/postorder uniquely identify a binary tree. Tree S is a subtree of T if both inorder and preorder traversals of S arew substrings of inorder and preorder traversals of T respectively.

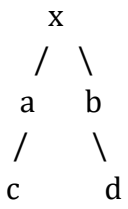
Following are detailed steps.
1) Find inorder and preorder traversals of T, store them in two auxiliary arrays inT[] and preT[].
2) Find inorder and preorder traversals of S, store them in two auxiliary arrays inS[] and preS[].
3) If inS[] is a subarray of inT[] and preS[] is a subarray preT[], then S is a subtree of T. Else not.
We can also use postorder traversal in place of preorder in the above algorithm.

Let us consider the above example
Inorder and Preorder traversals of the big tree are.
inT[] = {a, c, x, b, z, e, k}
preT[] = {z, x, a, c, b, e, k}
Inorder and Preorder traversals of small tree are
inS[] = {a, c, x, b}
preS[] = {x, a, c, b}
We can easily figure out that inS[] is a subarray of
inT[] and preS[] is a subarray of preT[].

EDIT
The above algorithm doesn't work for cases where a tree is present
in another tree, but not as a subtree. Consider the following example.

```
    Tree1
      x
     /  \
    a    b
   /
  c
```

Tree2



Inorder and Preorder traversals of the big tree or Tree2 are.
Inorder and Preorder traversals of small tree or Tree1 are

The Tree2 is not a subtree of Tree1, but inS[] and preS[] are subarrays of inT[] and preT[] respectively.

The **above algorithm** can be extended to handle such cases by adding a special character whenever we encounter NULL in inorder and preorder traversals.

```
/* This function returns true if S is a subtree of T, otherwise false */
bool isSubtree(Node *T, Node *S)
{
    /* base cases */
    if (S == NULL)  return true;
    if (T == NULL)  return false;

    // Store Inorder traversals of T and S in inT[0..m-1]
    // and inS[0..n-1] respectively
    int m = 0, n = 0;
    char inT[MAX], inS[MAX];
    storeInorder(T, inT, m);
    storeInorder(S, inS, n);
    inT[m] = '\0', inS[n] = '\0';

    // If inS[] is not a substring of preS[], return false
    if (strstr(inT, inS) == NULL)
        return false;

    // Store Preorder traversals of T and S in inT[0..m-1]
    // and inS[0..n-1] respectively
    m = 0, n = 0;
    char preT[MAX], preS[MAX];
    storePreOrder(T, preT, m);
    storePreOrder(S, preS, n);
    preT[m] = '\0', preS[n] = '\0';

    // If inS[] is not a substring of preS[], return false
    // Else return true
    return (strstr(preT, preS) != NULL);}
```

Time Complexity: Inorder and Preorder traversals of Binary Tree take O(n) time. The function strstr() can also be implemented in O(n) time using KMP string matching algorithm.
Auxiliary Space: O(n)

36 Connect nodes at same level

http://www.geeksforgeeks.org/connect-nodes-at-same-level/
Write a function to connect all the adjacent nodes at the same level in a binary tree. Structure of the given Binary Tree node is like following.

```
struct node{
  int data;
  struct node* left;
  struct node* right;
  struct node* nextRight;
}
```

Initially, all the nextRight pointers point to garbage values. Your function should set these pointers to point next right for each node.
Example
Input Tree

```
    A
   /\
  B  C
 /\  \
D  E  F
```

Output Tree

```
    A--->NULL
   /\
  B-->C-->NULL
 /\  \
D-->E-->F-->NULL
```

Method 1 (Extend Level Order Traversal or BFS)
Consider the method 2 of Level Order Traversal. The method 2 can easily be extended to connect nodes of same level. We can augment queue entries to contain level of nodes also which is 0 for root, 1 for root's children and so on. So a queue node will now contain a pointer to a tree node and an integer level. When we enqueue a node, we make sure that correct level value for node is being set in queue. To set nextRight, for

every node N, we dequeue the next node from queue, if the level number of next node is same, we set the nextRight of N as address of the dequeued node, otherwise we set nextRight of N as NULL.
Time Complexity: O(n)

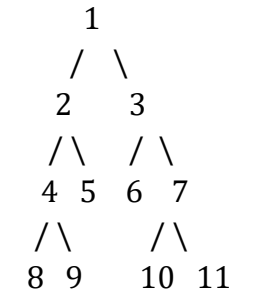
Method 2 (Extend Pre Order Traversal)
This approach works only for Complete Binary Trees. In this method we set nextRight in Pre Order fashion to make sure that the nextRight of parent is set before its children. When we are at node p, we set the nextRight of its left and right children. Since the tree is complete tree, nextRight of p's left child (p->left->nextRight) will always be p's right child, and nextRight of p's right child (p->right->nextRight) will always be left child of p's nextRight (if p is not the rightmost node at its level). If p is the rightmost node, then nextRight of p's right child will be NULL.

```
// Sets the nextRight of root and calls connectRecur() for other nodes
void connect (struct node *p)
{
    // Set the nextRight for root
    p->nextRight = NULL;

    // Set the next right for rest of the nodes (other than root)
    connectRecur(p);
}

/* Set next right of all descendents of p.
   Assumption:  p is a compete binary tree */
void connectRecur(struct node* p)
{
  // Base case
  if (!p)
    return;

  // Set the nextRight pointer for p's left child
  if (p->left)
    p->left->nextRight = p->right;

  // Set the nextRight pointer for p's right child
  // p->nextRight will be NULL if p is the right most child at its level
  if (p->right)
    p->right->nextRight = (p->nextRight)? p->nextRight->left: NULL;

  // Set nextRight for other nodes in pre order fashion
  connectRecur(p->left);
  connectRecur(p->right);
}
```

Time Complexity: O(n)

Why doesn't method 2 work for trees which are not Complete Binary Trees?
Let us consider following tree as an example. In Method 2, we set the nextRight pointer in pre order fashion. When we are at node 4, we set the nextRight of its children which are 8 and 9 (the nextRight of 4 is already set as node 5). nextRight of 8 will simply be set as 9, but nextRight of 9 will be set as NULL which is incorrect.

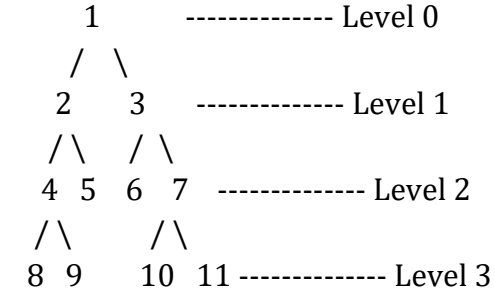
We can't set the correct nextRight, because when we set nextRight of 9, we only have nextRight of node 4 and ancestors of node 4, we don't have nextRight of nodes in right subtree of root.



http://www.geeksforgeeks.org/connect-nodes-at-same-level-with-o1-extra-space/

we traversed the nodes in pre order fashion. Instead of traversing in Pre Order fashion (root, left, right),
if we traverse the nextRight node before the left and right children (root, nextRight, left), then we can make sure that all nodes at level i have the nextRight set, before the level i+1 nodes. Let us consider the following example (same example as previous post).

The **method 2 fails** for right child of node 4. In this method, we make sure that all nodes at the 4's level (level 2) have nextRight set, before we try to set the nextRight of 9.
So when we set the nextRight of 9, we search for a nonleaf node on right side of node 4 (getNextRight() does this for us).



```
void connectRecur(struct node* p);
struct node *getNextRight(struct node *p);

// Sets the nextRight of root and calls connectRecur() for other nodes
void connect (struct node *p)
{
    // Set the nextRight for root
    p->nextRight = NULL;

    // Set the next right for rest of the nodes (other than root)
    connectRecur(p);
}

/* Set next right of all descendents of p. This function makes sure that
nextRight of nodes ar level i is set before level i+1 nodes. */
void connectRecur(struct node* p)
{
    // Base case
    if (!p)
       return;

    /* Before setting nextRight of left and right children, set nextRight
    of children of other nodes at same level (because we can access
    children of other nodes using p's nextRight only) */
    if (p->nextRight != NULL)
       connectRecur(p->nextRight);

    /* Set the nextRight pointer for p's left child */
    if (p->left)
    {
       if (p->right)
       {
           p->left->nextRight = p->right;
           p->right->nextRight = getNextRight(p);
       }
       else
           p->left->nextRight = getNextRight(p);

       /* Recursively call for next level nodes.  Note that we call only
       for left child. The call for left child will call for right child
*/
       connectRecur(p->left);
    }

    /* If left child is NULL then first node of next level will either be
      p->right or getNextRight(p) */
    else if (p->right)
    {
        p->right->nextRight = getNextRight(p);
        connectRecur(p->right);
    }
    else
       connectRecur(getNextRight(p));
}
```

```
/* This function returns the leftmost child of nodes at the same level as
p.
   This function is used to getNExt right of p's right child
   If right child of p is NULL then this can also be used for the left
child */
struct node *getNextRight(struct node *p)
{
    struct node *temp = p->nextRight;

    /* Traverse nodes at p's level and find and return
       the first node's first child */
    while(temp != NULL)
    {
        if(temp->left != NULL)
            return temp->left;
        if(temp->right != NULL)
            return temp->right;
        temp = temp->nextRight;
    }

    // If all the nodes at p's level are leaf nodes then return NULL
    return NULL;
}
```

An Iterative Solution

The recursive approach discussed above can be easily converted to iterative. In the iterative version, we use nested loop. The outer loop, goes through all the levels and the inner loop goes through all the nodes at every level. This solution uses constant space.

```
/* Sets nextRight of all nodes of a tree with root as p */
void connect(struct node* p)
{
    struct node *temp;

    if (!p)
      return;

    // Set nextRight for root
    p->nextRight = NULL;

    // set nextRight of all levels one by one
    while (p != NULL)
    {
        struct node *q = p;

   /*Connect all childrem nodes of p and children nodes of all other nodes
          at same level as p */
        while (q != NULL)
```

```
        {
            // Set the nextRight pointer for p's left child
            if (q->left)
            {
                // If q has right child, then right child is nextRight of
                // p and we also need to set nextRight of right child
                if (q->right)
                    q->left->nextRight = q->right;
                else
                    q->left->nextRight = getNextRight(q);
            }

            if (q->right)
                q->right->nextRight = getNextRight(q);

            // Set nextRight for other nodes in pre order fashion
            q = q->nextRight;
        }

        // start from the first node of next level
        if (p->left)
           p = p->left;
        else if (p->right)
           p = p->right;
        else
           p = getNextRight(p);
    }}
```

37 Populate Inorder Successor for all nodes

http://www.geeksforgeeks.org/populate-inorder-successor-for-all-nodes/

Given a Binary Tree where each node has following structure, write a function to populate next pointer for all nodes. The next pointer for every node should be set to point to inorder successor.

```
struct node
{
  int data;
  struct node* left;
  struct node* right;
  struct node* next;
}
```

Initially, all next pointers have NULL values. Your function should fill these next pointers so that they point to inorder successor.

Solution (Use Reverse Inorder Traversal)

Traverse the given tree in reverse inorder traversal and keep track of previously visited node. When a node is being visited, assign previously visited node as next.

```
/* Set next of p and all descendents of p by traversing them in reverse
Inorder */
void populateNext(struct node* p)
{
    // The first visited node will be the rightmost node
    // next of the rightmost node will be NULL
    static struct node *next = NULL;

    if (p)
    {
        // First set the next pointer in right subtree
        populateNext(p->right);

        // Set the next as previously visited node in reverse Inorder
        p->next = next;

        // Change the prev for subsequent node
        next = p;

        // Finally, set the next pointer in left subtree
        populateNext(p->left);
    }}
```

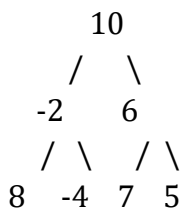
We can avoid the use of static variable by passing reference to next as parameter.

38 Convert a given tree to its Sum Tree

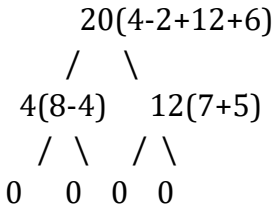
http://www.geeksforgeeks.org/convert-a-given-tree-to-sum-tree/

Given a Binary Tree where each node has positive and negative values. Convert this to a tree where each node contains the sum of the left and right sub trees in the original tree. The values of leaf nodes are changed to 0.

For example, the following tree



should be changed to

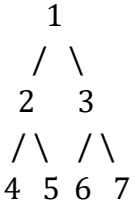


Do a **traversal** of the given tree. In the traversal, store the old value of the current node, recursively call for left and right subtrees and change the value of current node as sum of the values returned by the recursive calls.

Finally return the sum of new value and value (which is sum of values in the subtree rooted with this node).
Time Complexity: The solution involves a simple traversal of the given tree. So the time complexity is O(n) where n is the number of nodes in the given Binary Tree.

39 Vertical Sum in a given Binary Tree

http://www.geeksforgeeks.org/vertical-sum-in-a-given-binary-tree/
Given a Binary Tree, find vertical sum of the nodes that are in same vertical line. Print all sums through different vertical lines.
Examples:



The tree has 5 vertical lines
Vertical-Line-1 has only one node 4 => vertical sum is 4
Vertical-Line-2: has only one node 2=> vertical sum is 2
Vertical-Line-3: has three nodes: 1,5,6 => vertical sum is 1+5+6 = 12
Vertical-Line-4: has only one node 3 => vertical sum is 3
Vertical-Line-5: has only one node 7 => vertical sum is 7
So expected output is 4, 2, 12, 3 and 7

Solution:

We need to check the Horizontal Distances from root for all nodes. If two nodes have the same Horizontal Distance (HD), then they are on same vertical line. The idea of HD is simple.

HD for root is 0, a **right edge** (edge connecting to right subtree) is considered **as +1** horizontal distance and a **left edge** is considered **as -1** horizontal distance.

For example, in the above tree, HD for Node 4 is at -2, HD for Node 2 is -1, HD for 5 and 6 is 0 and HD for node 7 is +2.

We can do **inorder traversal** of the given Binary Tree. While traversing the tree, we can recursively calculate HDs. We initially pass the horizontal distance as 0 for root. For **left subtree**, we pass the Horizontal Distance as Horizontal distance of root minus 1. For right subtree, we pass the Horizontal Distance as Horizontal Distance of root plus 1.

Following is Java implementation for the same. HashMap is used to store the vertical sums for different horizontal distances.

```
    // Traverses the tree in Inoorder form and builds a hashMap hM that
    // contains the vertical sum
    private void VerticalSumUtil(TreeNode root, int hD,
                                        HashMap<Integer, Integer> hM) {

        // base case
        if (root == null) {  return; }

        // Store the values in hM for left subtree
        VerticalSumUtil(root.left(), hD - 1, hM);

        // Update vertical sum for hD of this node
        int prevSum = (hM.get(hD) == null) ? 0 : hM.get(hD);
        hM.put(hD, prevSum + root.key());

        // Store the values in hM for right subtree
        VerticalSumUtil(root.right(), hD + 1, hM);
    }
}
```

Time Complexity: O(n)

40 Vertical Sum in Binary Tree | Set 2 (Space Optimized)

http://www.geeksforgeeks.org/vertical-sum-in-binary-tree-set-space-optimized/
in Set 1. Hashing based solution requires a Hash Table to be maintained. We know that hashing requires more space than the number of entries in it. In this post, Doubly Linked List based solution is discussed. The solution discussed here requires only n nodes of linked list where n is total number of vertical lines in binary tree.

Algorithm
VerticalSumDLL(root)

```
1)  Create a node of doubly linked list node
    with value 0. Let the node be llnode.
2)  verticalSumDLL(root, llnode)
```

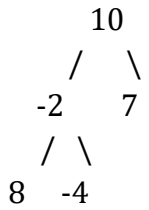
VerticalSumDLL(tnode, llnode)

```
1) Add current node's data to its vertical line
      llnode.data = llnode.data + tnode.data;
2) Recursively process left subtree
 // If left child is not empty
  if (tnode.left != null)
  {
    if (llnode.prev == null)
    {
       llnode.prev = new LLNode(0);
       llnode.prev.next = llnode;
    }
    verticalSumDLLUtil(tnode.left, llnode.prev);
  }
3)  Recursively process right subtree
  if (tnode.right != null)
  {
    if (llnode.next == null)
    {
       llnode.next = new LLNode(0);
       llnode.next.prev = llnode;
    }
    verticalSumDLLUtil(tnode.right, llnode.next);   }
```

```
public class VerticalSumBinaryTree
{
    // Prints vertical sum of different vertical
    // lines in tree. This method mainly uses
    // verticalSumDLLUtil().
    static void verticalSumDLL(TNode root)
    {
        // Create a doubly linked list node to
        // store sum of lines going through root.
        // Vertical sum is initialized as 0.
        LLNode llnode = new LLNode(0);

        // Compute vertical sum of different lines
        verticalSumDLLUtil(root, llnode);

        // llnode refers to sum of vertical line
        // going through root. Move llnode to the
        // leftmost line.
        while (llnode.prev != null)
            llnode = llnode.prev;

        // Prints vertical sum of all lines starting
        // from leftmost vertical line
        while (llnode != null)
        {
            System.out.print(llnode.data +" ");
            llnode = llnode.next;
        }
    }
     // Constructs linked list
    static void verticalSumDLLUtil(TNode tnode,
                                   LLNode llnode)
    {
        // Add current node's data to its vertical line
        llnode.data = llnode.data + tnode.data;
         // Recursively process left subtree
        if (tnode.left != null)
        {
            if (llnode.prev == null)
            {   llnode.prev = new LLNode(0);
                llnode.prev.next = llnode;
            }
            verticalSumDLLUtil(tnode.left, llnode.prev);
        }
         // Process right subtree
        if (tnode.right != null)
        {
            if (llnode.next == null)
            {
                llnode.next = new LLNode(0);
                llnode.next.prev = llnode;
            }
            verticalSumDLLUtil(tnode.right, llnode.next);
        }}
```

41 Find the maximum sum leaf to root path in a Binary Tree

http://www.geeksforgeeks.org/find-the-maximum-sum-path-in-a-binary-tree/
Given a Binary Tree, find the maximum sum path from a leaf to root. For example, in the following tree, there are three leaf to root paths 8->-2->10, -4->-2->10 and 7->10. The sums of these three paths are 16, 4 and 17 respectively. The maximum of them is 17 and the path for maximum is 7->10.



Solution

1) First find the leaf node that is on the maximum sum path. In the following code getTargetLeaf() does this by assigning the result to *target_leaf_ref.

2) Once we have the target leaf node, we can print the maximum sum path by traversing the tree. In the following code, printPath() does this.
The main function is maxSumPath() that uses above two functions to get the complete

solution.

```
// A utility function that prints all nodes
// on the path from root to target_leaf
bool printPath (struct node *root,
                struct node *target_leaf)
{
    // base case
    if (root == NULL)
        return false;

    // return true if this node is the target_leaf
    // or target leaf is present in one of its
    // descendants
    if (root == target_leaf ||
        printPath(root->left, target_leaf) ||
        printPath(root->right, target_leaf) )
    {
        printf("%d ", root->data);
        return true;
    }

    return false;
}

// This function Sets the target_leaf_ref to refer
```

```
// the leaf node of the maximum  path sum.  Also,
// returns the max_sum using max_sum_ref
void getTargetLeaf (struct node *node, int *max_sum_ref,
                    int curr_sum, struct node **target_leaf_ref)
{
    if (node == NULL)
        return;

    // Update current sum to hold sum of nodes on path
    // from root to this node
    curr_sum = curr_sum + node->data;

    // If this is a leaf node and path to this node has
    // maximum sum so far, then make this node target_leaf
    if (node->left == NULL && node->right == NULL)
    {
        if (curr_sum > *max_sum_ref)
        {
            *max_sum_ref = curr_sum;
            *target_leaf_ref = node;
        }
    }

    // If this is not a leaf node, then recur down
    // to find the target_leaf
    getTargetLeaf (node->left, max_sum_ref, curr_sum,
                   target_leaf_ref);
    getTargetLeaf (node->right, max_sum_ref, curr_sum,
                   target_leaf_ref);
}

// Returns the maximum sum and prints the nodes on max
// sum path
int maxSumPath (struct node *node)
{
    // base case
    if (node == NULL)
        return 0;

    struct node *target_leaf;
    int max_sum = INT_MIN;

    // find the target leaf and maximum sum
    getTargetLeaf (node, &max_sum, 0, &target_leaf);

    // print the path from root to the target leaf
    printPath (node, target_leaf);

    return max_sum;  // return maximum sum
}
```

Time Complexity: Time complexity of the above solution is O(n) as it involves tree traversal two times.

42 Construct Special Binary Tree from given Inorder traversal

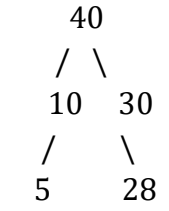
http://www.geeksforgeeks.org/construct-binary-tree-from-inorder-traversal/

Given Inorder Traversal of a Special Binary Tree in which key of every node is greater than keys in left and right children, construct the Binary Tree and return root.

Examples:

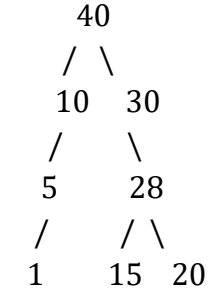
Input: inorder[] = {5, 10, 40, 30, 28}

Output: root of following tree

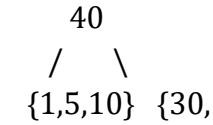


Input: inorder[] = {1, 5, 10, 40, 30, 15, 28, 20}

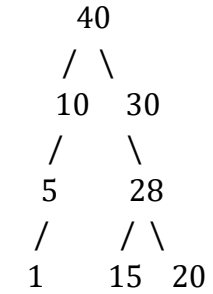
Output: root of following tree



The **idea** used in **Construction of Tree from given Inorder and Preorder traversals** can be used here. Let the given array is {1, 5, 10, 40, 30, 15, 28, 20}. The maximum element in given array must be root. The elements on left side of the maximum element are in left subtree and elements on right side are in right subtree.



We recursively follow above step for left and right subtrees, and finally get the following tree.



Algorithm: buildTree()

1) Find index of the maximum element in array. The maximum element must be root of Binary Tree.

2) Create a new tree node 'root' with the data as the maximum value found in step 1.

3) Call buildTree for elements before the maximum element and make the built tree as left subtree of 'root'.

5) Call buildTree for elements after the maximum element and make the built tree as right subtree of 'root'.

6) return 'root'.

```
/* Recursive function to construct binary of size len from
   Inorder traversal inorder[]. Initial values of start and end
   should be 0 and len -1.  */
struct node* buildTree (int inorder[], int start, int end)
{
    if (start > end)
        return NULL;

    /* Find index of the maximum element from Binary Tree */
    int i = max (inorder, start, end);

    /* Pick the maximum value and make it root */
    struct node *root = newNode(inorder[i]);

    /* If this is the only element in inorder[start..end],
       then return it */
    if (start == end)
        return root;

    /* Using index in Inorder traversal, construct left and
       right subtress */
    root->left  = buildTree (inorder, start, i-1);
    root->right = buildTree (inorder, i+1, end);

    return root;
}
```

Time Complexity: O(n^2)

43 Construct a special tree from given preorder traversal

http://www.geeksforgeeks.org/construct-a-special-tree-from-given-preorder-traversal/

Given an array 'pre[]' that represents Preorder traversal of a spacial binary tree where every node has either 0 or 2 children. One more array 'preLN[]' is given which has only two possible values 'L' and 'N'. The value 'L' in 'preLN[]' indicates that the corresponding node in Binary Tree is a leaf node and value 'N' indicates that the corresponding node is non-leaf node. Write a function to construct the tree from the given two arrays.

Example:
Input: pre[] = {10, 30, 20, 5, 15}, preLN[] = {'N', 'N', 'L', 'L', 'L'}
Output: Root of following tree

```
      10
     / \
    30  15
   / \
  20  5
```

The **first element** in **pre**[] will always be root. So we can easily figure out root. If left subtree is empty, the right subtree must also be empty and preLN[] entry for root must be 'L'. We can simply create a node and return it. If left and right subtrees are not empty, then recursively call for left and right subtrees and link the returned nodes to root.

```
/* A recursive function to create a Binary Tree from given pre[]
   preLN[] arrays. The function returns root of tree. index_ptr is used
   to update index values in recursive calls. index must be initially
   passed as 0 */
struct node *constructTreeUtil(int pre[], char preLN[], int *index_ptr,
int n)
{
    int index = *index_ptr; // store the current value of index in pre[]

    // Base Case: All nodes are constructed
    if (index == n)
        return NULL;

    // Allocate memory for this node and increment index for
    // subsequent recursive calls
    struct node *temp = newNode ( pre[index] );
    (*index_ptr)++;
```

```
    // If this is an internal node, construct left and right subtrees and
link the subtrees
    if (preLN[index] == 'N')
    {
      temp->left  = constructTreeUtil(pre, preLN, index_ptr, n);
      temp->right = constructTreeUtil(pre, preLN, index_ptr, n);
    }

    return temp;
}

// A wrapper over constructTreeUtil()
struct node *constructTree(int pre[], char preLN[], int n)
{
    // Initialize index as 0. Value of index is used in recursion to
maintain
    // the current index in pre[] and preLN[] arrays.
    int index = 0;

    return constructTreeUtil (pre, preLN, &index, n);
}
```

Time Complexity: O(n)

44 Check whether a given Binary Tree is Complete or not | Set 1 (Iterative Solution)

http://www.geeksforgeeks.org/check-if-a-given-binary-tree-is-complete-tree-or-not/
Given a Binary Tree, write a function to check whether the given Binary Tree is Complete Binary Tree or not.

A **complete binary tree** is a binary tree in which **every level, except possibly the last, is completely filled, and all nodes are as far left as possible**. See following examples.
The following trees are examples of Complete Binary Trees

```
    1
  /   \
 2     3
```

```
      1
    /   \
   2     3
  /
 4
```

The following trees are examples of Non-Complete Binary Trees

```
    1
     \
      3
```

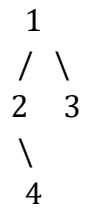
```
      1
    /   \
   2     3
    \   / \
     4 5   6
```

The method 2 of level order traversal post can be easily modified to check whether a tree is Complete or not.

To understand the approach, let us first define a term 'Full Node'. A node is 'Full Node' if both left and right children are not empty (or not NULL).

The approach is to do a level order traversal starting from root. In the traversal, once a node is found which is NOT a Full Node, all the following nodes must be leaf nodes.

Also, one more thing needs to be checked to handle the below case: If a node has empty left child, then the right child must be empty.




```
/* Given a binary tree, return true if the tree is complete
   else false */
bool isCompleteBT(struct node* root)
{
  // Base Case: An empty tree is complete Binary Tree
  if (root == NULL)
    return true;

  // Create an empty queue
  int rear, front;
  struct node **queue = createQueue(&front, &rear);

  // Create a flag variable which will be set true
  // when a non full node is seen
  bool flag = false;

  // Do level order traversal using queue.
  enQueue(queue, &rear, root);
  while(!isQueueEmpty(&front, &rear))
  {
    struct node *temp_node = deQueue(queue, &front);

    /* Check if left child is present*/
    if(temp_node->left)
    {
       // If we have seen a non full node, and we see a node
       // with non-empty left child, then the given tree is not
       // a complete Binary Tree
       if (flag == true)
         return false;

       enQueue(queue, &rear, temp_node->left);  // Enqueue Left Child
    }
```

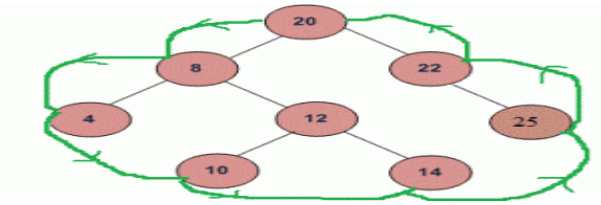
```
    else // If this a non-full node, set the flag as true
       flag = true;

    /* Check if right child is present*/
    if(temp_node->right)
    {
       // If we have seen a non full node, and we see a node
       // with non-empty right child, then the given tree is not
       // a complete Binary Tree
       if(flag == true)
         return false;

       enQueue(queue, &rear, temp_node->right);  // Enqueue Right Child
    }
    else // If this a non-full node, set the flag as true
       flag = true;
  }

  // If we reach here, then the tree is complete Bianry Tree
  return true;
}
```

Time Complexity: O(n) where n is the number of nodes in given Binary Tree
Auxiliary Space: O(n) for queue.

45 Boundary Traversal of binary tree

http://www.geeksforgeeks.org/boundary-traversal-of-binary-tree/

Given a binary tree, print boundary nodes of the binary tree Anti-Clockwise starting from the root. For example, boundary traversal of the following tree is "20 8 4 10 14 25 22"



We break the problem in 3 parts:
1. Print the left boundary in top-down manner.
2. Print all leaf nodes from left to right, which can again be sub-divided into two sub-parts:
.....2.1 Print all leaf nodes of left sub-tree from left to right.
.....2.2 Print all leaf nodes of right subtree from left to right.
3. Print the right boundary in bottom-up manner.

We need to take care of one thing that nodes are not printed again. e.g. The left most node is also the leaf node of the tree.

Based on the above cases, below is the implementation:

```
// A function to do boundary traversal of a given binary tree
void printBoundary (struct node* root)
{
    if (root)
    {
        printf("%d ",root->data);

        // Print the left boundary in top-down manner.
        printBoundaryLeft(root->left);

        // Print all leaf nodes
        printLeaves(root->left);
        printLeaves(root->right);

        // Print the right boundary in bottom-up manner
        printBoundaryRight(root->right);
    }
}
```

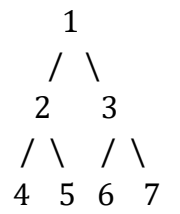
Time Complexity: O(n) where n is the number of nodes in binary tree.

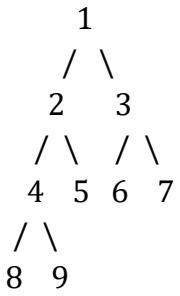
46 Construct Full Binary Tree from given preorder and postorder traversals

http://www.geeksforgeeks.org/full-and-complete-binary-tree-from-given-preorder-and-postorder-traversals/

Given two arrays that represent preorder and postorder traversals of a full binary tree, construct the binary tree.

A **Full Binary Tree** is a binary tree where every node has either 0 or 2 children
Following are examples of Full Trees.

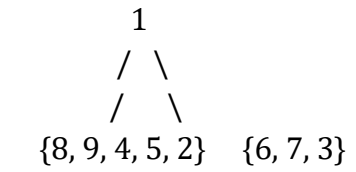




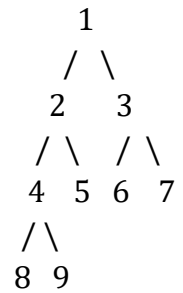
It is not possible to construct a general Binary Tree from preorder and postorder traversals (See this). But if know that the Binary Tree is Full, we can construct the tree without ambiguity. Let us understand this with the help of following example.
Let us consider the two given arrays as pre[] = {1, 2, 4, 8, 9, 5, 3, 6, 7} and post[] = {8, 9, 4, 5, 2, 6, 7, 3, 1};

In pre[], the leftmost element is root of tree. Since the tree is full and array size is more than 1. The value next to 1 in pre[], must be left child of root. So we know 1 is root and 2 is left child.

How to find the all nodes in left subtree? We know 2 is root of all nodes in left subtree. All nodes before 2 in post[] must be in left subtree. Now we know 1 is root, elements {8, 9, 4, 5, 2} are in left subtree, and the elements {6, 7, 3} are in right subtree.



We recursively follow the above approach and get the following tree.




```
// A recursive function to construct Full from pre[] and post[].
// preIndex is used to keep track of index in pre[].
// l is low index and h is high index for the current subarray in post[]
struct node* constructTreeUtil (int pre[], int post[], int* preIndex,
                                int l, int h, int size)
{
    // Base case
    if (*preIndex >= size || l > h)
        return NULL;

    // The first node in preorder traversal is root. So take the node at
    // preIndex from preorder and make it root, and increment preIndex
    struct node* root = newNode ( pre[*preIndex] );
    ++*preIndex;

    // If the current subarry has only one element, no need to recur
    if (l == h)
        return root;

    // Search the next element of pre[] in post[]
    int i;
    for (i = l; i <= h; ++i)
        if (pre[*preIndex] == post[i])
            break;
     // Use the index of element found in postorder to divide postorder
array in
    // two parts. Left subtree and right subtree
    if (i <= h)
    {
        root->left = constructTreeUtil (pre, post, preIndex, l, i, size);
        root->right = constructTreeUtil (pre, post, preIndex, i + 1, h,
size);
    }

    return root;
}
// The main function to construct Full Binary Tree from given preorder and
```

```
// postorder traversals. This function mainly uses constructTreeUtil()
struct node *constructTree (int pre[], int post[], int size)
{
    int preIndex = 0;
    return constructTreeUtil (pre, post, &preIndex, 0, size - 1, size);
}
```

47 Iterative Preorder Traversal

http://www.geeksforgeeks.org/iterative-preorder-traversal/

Given a Binary Tree, write an iterative function to print Preorder traversal of the given binary tree.

Refer this for recursive preorder traversal of Binary Tree. To convert an inherently recursive procedures to iterative, we need an explicit stack. Following is a simple stack based iterative process to print Preorder traversal.

1) Create an empty stack nodeStack and push root node to stack.

2) Do following while nodeStack is not empty.

....a) Pop an item from stack and print it.
....b) Push right child of popped item to stack
....c) Push left child of popped item to stack

Right child is pushed before left child to make sure that left subtree is processed first.

```
// An iterative process to print preorder traversal of Binary tree
void iterativePreorder(node *root)
{
    // Base Case
    if (root == NULL)
       return;

    // Create an empty stack and push root to it
    stack<node *> nodeStack;
    nodeStack.push(root);

    /* Pop all items one by one. Do following for every popped item
       a) print it
       b) push its right child
       c) push its left child
    Note that right child is pushed first so that left is processed first
*/
```

```
    while (nodeStack.empty() == false)
    {
        // Pop the top item from stack and print it
        struct node *node = nodeStack.top();
        printf ("%d ", node->data);
        nodeStack.pop();

        // Push right and left children of the popped node to stack
        if (node->right)
            nodeStack.push(node->right);
        if (node->left)
            nodeStack.push(node->left);
    }
}
```

48 Morris traversal for Preorder

http://www.geeksforgeeks.org/morris-traversal-for-preorder/

Using **Morris Traversal**, we can traverse the tree without using stack and recursion. The algorithm for Preorder is almost similar to Morris traversal for Inorder.

1...If left child is null, print the current node data. Move to right child.
....Else, Make the right child of the inorder predecessor point to the current node. Two cases arise:
.....a) The right child of the inorder predecessor already points to the current node. Set right child to NULL. Move to right child of current node.
.....b) The right child is NULL. Set it to current node. Print current node's data and move to left child of current node.

2...Iterate until current node is not NULL.

Following is the implementation of the above algorithm

```
// Preorder traversal without recursion and without stack
void morrisTraversalPreorder(struct node* root)
{
    while (root)
    {
        // If left child is null, print the current node data. Move to
        // right child.
        if (root->left == NULL)
        {
            printf( "%d ", root->data );
            root = root->right;
        }
        else
        {
            // Find inorder predecessor
            struct node* current = root->left;
            while (current->right && current->right != root)
                current = current->right;

            // If the right child of inorder predecessor already points to
            // this node
            if (current->right == root)
            {
                current->right = NULL;
                root = root->right;
            }

            // If right child doesn't point to this node, then print this
            // node and make right child point to this node
```

```
            else
            {
                printf("%d ", root->data);
                current->right = root;
                root = root->left;
            }
        }
    }
}
```

Limitations:

Morris traversal modifies the tree during the process. It establishes the right links while moving down the tree and resets the right links while moving up the tree. So the algorithm cannot be applied if write operations are not allowed.

49 Linked complete binary tree & its creation

http://www.geeksforgeeks.org/linked-complete-binary-tree-its-creation/

A complete binary tree is a binary tree where each level 'l' except the last has 2^l nodes and the nodes at the last level are all left aligned. Complete binary trees are mainly used in heap based data structures.

The nodes in the complete binary tree are inserted from left to right in one level at a time. If a level is full, the node is inserted in a new level.

Below are some of the complete binary trees.

```
    1
   / \
  2   3
```

```
     1
    / \
   2   3
  / \ /
 4  5 6
```

Complete binary trees are generally **represented** using **arrays**. The array representation is better because it doesn't contain any empty slot. Given parent index i, its **left child** is given by **2 * i + 1** and its **right child** is given by **2 * i + 2**. So no extra space is wasted and space to store left and right pointers is saved.

However, it may be an interesting programming question to create a Complete Binary Tree using linked representation. Here Linked mean a non-array representation where left and right pointers(or references) are used to refer left and right children respectively.

How to write an insert function that always adds a new node in the last level and at the leftmost available position?

To create a linked complete binary tree, we need to keep track of the nodes in a level order fashion such that the next node to be inserted lies in the leftmost position. A queue data structure can be used to keep track of the inserted nodes.

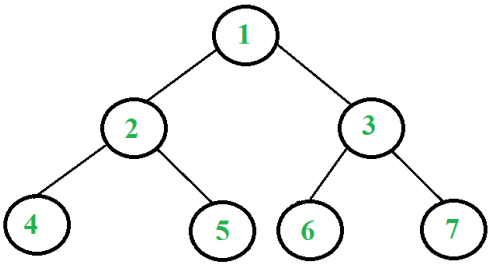
Following are steps to insert a new node in Complete Binary Tree.
1. If the tree is empty, initialize the root with new node.
2. Else, get the front node of the queue.
…….If the left child of this front node doesn't exist, set the left child as the new node.
…….else if the right child of this front node doesn't exist, set the right child as the new node.
3. If the front node has both the left child and right child, Dequeue() it.
4. Enqueue() the new node.

50 Iterative Postorder Traversal | Set 1 (Using Two Stacks)

http://www.geeksforgeeks.org/iterative-postorder-traversal/
We have discussed iterative inorder and iterative preorder traversals. In this post, iterative postorder traversal is discussed which is more complex than the other two traversals (due to its nature of non-tail recursion, there is an extra statement after the final recursive call to itself). The postorder traversal can easily be done using two stacks though.

The **idea** is to **push reverse postorder traversal to a stack**. Once we have reverse postorder traversal in a stack, we can just **pop all items one by one from the stack and print them**, this order of printing will be in postorder because of LIFO property of stacks. Now the question is, how to get reverse post order elements in a stack – the other stack is used for this purpose.

For example, in the



Following tree, we need to get 1, 3, 7, 6, 2, 5, 4 in a stack. If take a closer look at this sequence, we can observe that this sequence is very similar to preorder traversal. The only difference is right child is visited before left child and therefore sequence is "root right left" instead of "root left right". So we can do something like iterative preorder traversal with following differences.

a) Instead of printing an item, we push it to a stack.
b) We push left subtree before right subtree.

Following is the complete algorithm. After step 2, we get reverse postorder traversal in second stack. We use first stack to get this order.

1. Push root to first stack.
2. Loop while first stack is not empty
   2.1 Pop a node from first stack and push it to second stack
   2.2 Push left and right children of the popped node to first stack
3. Print contents of second stack

```
// An iterative function to do post order traversal of a given binary tree
void postOrderIterative(struct Node* root)
{
    if (root == NULL)
        return;

    // Create two stacks
    struct Stack* s1 = createStack(MAX_SIZE);
    struct Stack* s2 = createStack(MAX_SIZE);

    // push root to first stack
    push(s1, root);
    struct Node* node;
```

```
  // Run while first stack is not empty
    while (!isEmpty(s1))
    {
        // Pop an item from s1 and push it to s2
        node = pop(s1);
        push(s2, node);

        // Push left and right children of removed item to s1
        if (node->left)
            push(s1, node->left);
        if (node->right)
            push(s1, node->right);
    }

    // Print all elements of second stack
    while (!isEmpty(s2))
    {
        node = pop(s2);
        printf("%d ", node->data);
    }
}
```

Following is overview of the above post.
Iterative preorder traversal can be easily implemented using two stacks. The first stack is used to get the reverse postorder traversal in second stack. The steps to get reverse postorder are similar to **iterative preorder**.

51 Iterative Postorder Traversal | Set 2 (Using One Stack)

http://www.geeksforgeeks.org/iterative-postorder-traversal-using-stack/
We have discussed a simple iterative postorder traversal using two stacks in the previous post. In this post, an approach with only one stack is discussed.

The **idea** is to **move down to leftmost node using left pointer**. While **moving down**, push root and **root's right child to stack**. Once we reach leftmost node, print it if it doesn't have a right child. If it has a right child, then change root so that the right child is processed before.

Following is detailed algorithm.

1.1 Create an empty stack
2.1 Do following while root is not NULL
  a) Push root's right child and then root to stack.
  b) Set root as root's left child.

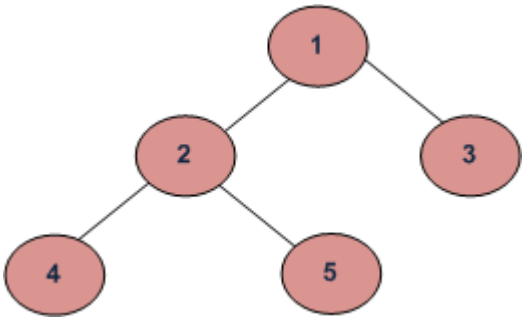
2.2 Pop an item from stack and set it as root.
  a) If the popped item has a right child and the right child
    is at top of stack, then remove the right child from stack,
    push the root back and set root as root's right child.
  b) Else print root's data and set root as NULL.
2.3 Repeat steps 2.1 and 2.2 while stack is not empty.

```
// An iterative function to do postorder traversal of a given binary tree
void postOrderIterative(struct Node* root)
{
    // Check for empty tree
    if (root == NULL)
        return;

    struct Stack* stack = createStack(MAX_SIZE);
    do
    {
        // Move to leftmost node
        while (root)
        {
            // Push root's right child and then root to stack.
            if (root->right)
                push(stack, root->right);
            push(stack, root);

            // Set root as root's left child
            root = root->left;
        }

        // Pop an item from stack and set it as root
        root = pop(stack);

        // If the popped item has a right child and the right child is not
     // processed yet, then make sure right child is processed before root
        if (root->right && peek(stack) == root->right)
        {
            pop(stack);  // remove right child from stack
            push(stack, root);  // push root back to stack
            root = root->right; // change root so that the right
                                // child is processed next
        }
        else  // Else print root's data and set root as NULL
        {
            printf("%d ", root->data);
            root = NULL;
        }
    } while (!isEmpty(stack));
}
```

52 Reverse Level Order Traversal

http://www.geeksforgeeks.org/reverse-level-order-traversal/

We have discussed level order traversal of a post in previous post. The idea is to print last level first, then second last level, and so on. Like Level order traversal, every level is printed from left to right.



Example Tree

Reverse Level order traversal of the above tree is "4 5 2 3 1".

Both methods for normal level order traversal can be easily modified to do reverse level order traversal.

METHOD 1 (Recursive function to print a given level)

We can easily modify the method 1 of the normal level order traversal. In method 1, we have a method printGivenLevel() which prints a given level number. The only thing we need to change is, instead of calling printGivenLevel() from first level to last level, we call it from last level to first level.

Time Complexity: The worst case time complexity of this method is O(n^2). For a skewed tree, printGivenLevel() takes O(n) time where n is the number of nodes in the skewed tree. So time complexity of printLevelOrder() is O(n) + O(n-1) + O(n-2) + .. + O(1) which is O(n^2).

METHOD 2 (Using Queue and Stack)

The method 2 of normal level order traversal can also be easily modified to print level order traversal in reverse order. The idea is to use a stack to get the reverse level order. If we do normal level order traversal and instead of printing a node, push the node to a stack and then print contents of stack, we get "5 4 3 2 1" for above example tree, but output should be "4 5 2 3 1". So to get the correct sequence (left to right at every level), we process children of a node in reverse order, we first push the right subtree to stack, then left subtree.

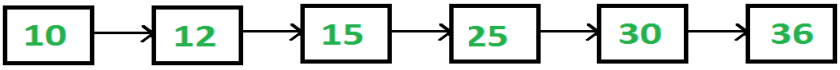
Time Complexity: O(n) where n is number of nodes in the binary tree.

53 Construct Complete Binary Tree from its Linked List Representation

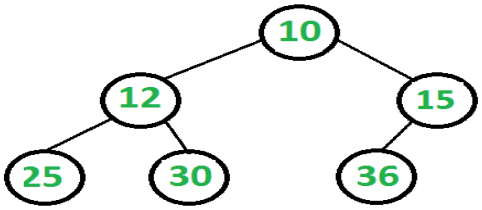
http://www.geeksforgeeks.org/given-linked-list-representation-of-complete-tree-convert-it-to-linked-representation/

Given Linked List Representation of Complete Binary Tree, construct the Binary tree. A complete binary tree can be represented in an array in the following approach.
If root node is stored at index i, its left, and right children are stored at indices 2*i+1, 2*i+2 respectively.

Suppose tree is represented by a linked list in same way, how do we convert this into normal linked representation of binary tree where every node has data, left and right pointers? In the linked list representation, we cannot directly access the children of the current node unless we traverse the list.



The above linked list represents following binary tree



We are mainly given level order traversal in sequential access form. We know head of linked list is always is root of the tree. We take the first node as root and we also know that the next two nodes are left and right children of root. So we know partial Binary Tree.
The **idea** is to do Level order traversal of the partially built Binary Tree using queue and traverse the linked list at the same time. At every step, we take the parent node from queue, make next two nodes of linked list as children of the parent node, and enqueue the next two nodes to queue.

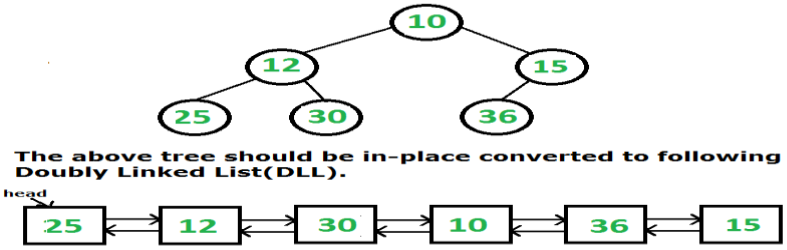
1. Create an empty queue.
2. Make the first node of the list as root, and enqueue it to the queue.
3. Until we reach the end of the list, do the following.
.....a. Dequeue one node from the queue. This is the current parent.
.....b. Traverse two nodes in the list, add them as children of the current parent.
.....c. Enqueue the two nodes into the queue.

Time Complexity: O(n)

54 Convert a given Binary Tree to Doubly Linked List

http://www.geeksforgeeks.org/in-place-convert-a-given-binary-tree-to-doubly-linked-list/

Given a Binary Tree (Bt), convert it to a Doubly Linked List(DLL). The left and right pointers in nodes are to be used as previous and next pointers respectively in converted DLL. The order of nodes in DLL must be same as Inorder of the given Binary Tree. The first node of Inorder traversal (left most node in BT) must be head node of the DLL.



I came across this interview during one of my interviews. A similar problem is discussed in this post. The problem here is simpler as we don't need to create circular DLL, but a simple DLL.

The **idea** behind its solution is quite simple and straight.

1. If left subtree exists, process the left subtree
…..1.a) Recursively convert the left subtree to DLL.
…..1.b) Then find inorder predecessor of root in left subtree (inorder predecessor is rightmost node in left subtree).
…..1.c) Make inorder predecessor as previous of root and root as next of inorder predecessor.

2. If right subtree exists, process the right subtree (Below 3 steps are similar to left subtree).
…..2.a) Recursively convert the right subtree to DLL.
…..2.b) Then find inorder successor of root in right subtree (inorder successor is leftmost node in right subtree).
…..2.c) Make inorder successor as next of root and root as previous of inorder successor.
3. Find the leftmost node and return it (the leftmost node is always head of converted DLL).

Method 2

http://www.geeksforgeeks.org/convert-a-given-binary-tree-to-doubly-linked-list-set-2/

In this post, another simple and efficient solution is discussed. The solution discussed here has two simple steps.

1) **Fix Left Pointers**: In this step, we change left pointers to point to previous nodes in DLL. The idea is simple, we do inorder traversal of tree. In inorder traversal, we keep track of previous visited node and change left pointer to the previous node.
See fixPrevPtr() in below implementation.

2) **Fix Right Pointers**: The above is intuitive and simple. How to change right pointers to point to next node in DLL? The idea is to use left pointers fixed in step 1. We start from the rightmost node in Binary Tree (BT). The rightmost node is the last node in DLL. Since left pointers are changed to point to previous node in DLL, we can linearly traverse the complete DLL using these pointers.
The traversal would be from last to first node. While traversing the DLL, we keep track of the previously visited node and change the right pointer to the previous node.
See fixNextPtr() in below implementation.

```
// Changes left pointers to work as previous pointers in converted DLL
// The function simply does inorder traversal of Binary Tree and updates
// left pointer using previously visited node
void fixPrevPtr(struct node *root)
{
    static struct node *pre = NULL;

    if (root != NULL)
    {
        fixPrevPtr(root->left);
        root->left = pre;
        pre = root;
        fixPrevPtr(root->right);
    }
}

// Changes right pointers to work as next pointers in converted DLL
struct node *fixNextPtr(struct node *root)
{
    struct node *prev = NULL;

    // Find the right most node in BT or last node in DLL
    while (root && root->right != NULL)
        root = root->right;
```

```
     // Start from the rightmost node, traverse back using left pointers.
    // While traversing, change right pointer of nodes.
    while (root && root->left != NULL)
    {
        prev = root;
        root = root->left;
        root->right = prev;
    }

    // The leftmost node is head of linked list, return it
    return (root);
}

// The main function that converts BST to DLL and returns head of DLL
struct node *BTToDLL(struct node *root)
{
    // Set the previous pointer
    fixPrevPtr(root);

    // Set the next pointer and return head of DLL
    return fixNextPtr(root);
}
```

Time Complexity: O(n) where n is the number of nodes in given Binary Tree. The solution simply does two traversals of all Binary Tree nodes.

Method 3

http://www.geeksforgeeks.org/convert-given-binary-tree-doubly-linked-list-set-3/

In this post, a third solution is discussed which seems to be the simplest of all. The idea is to do inorder traversal of the binary tree. While doing inorder traversal, keep track of the previously visited node in a variable say prev. For every visited node, make it next of prev and previous of this node as prev.

```
// A simple recursive function to convert a given Binary tree to Doubly
// Linked List
// root --> Root of Binary Tree
// head --> Pointer to head node of created doubly linked list
void BinaryTree2DoubleLinkedList(node *root, node **head)
{
    // Base case
    if (root == NULL) return;

    // Initialize previously visited node as NULL. This is
    // static so that the same value is accessible in all recursive
    // calls
    static node* prev = NULL;

    // Recursively convert left subtree
    BinaryTree2DoubleLinkedList(root->left, head);

    // Now convert this node
    if (prev == NULL)
```

```
        *head = root;
    else
    {
        root->left = prev;
        prev->right = root;
    }
    prev = root;

    // Finally convert right subtree
    BinaryTree2DoubleLinkedList(root->right, head);
}
```

Note that use of static variables like above is not a recommended practice (we have used static for simplicity). Imagine a situation where same function is called for two or more trees, the old value of *prev* would be used in next call for a different tree. To avoid such problems, we can use double pointer or reference to a pointer.
Time Complexity: The above program does a simple inorder traversal.
Time complexity is O(n) where n is the number of nodes in given binary tree.

Method 4
http://www.geeksforgeeks.org/convert-a-given-binary-tree-to-doubly-linked-list-set-4/

In the following implementation, we traverse the tree in inorder fashion.
We add nodes at the beginning of current linked list and update head of the list using pointer to head pointer. Since we insert at the beginning, we need to process leaves in reverse order.

For reverse order, we first traverse the right subtree before the left subtree. i.e. do a reverse inorder traversal

```
// A simple recursive function to convert a given
// Binary tree to Doubly Linked List
// root      --> Root of Binary Tree
// head_ref --> Pointer to head node of created
//              doubly linked list
void BToDLL(Node* root, Node** head_ref)
{
    // Base cases
    if (root == NULL)
        return;

    // Recursively convert right subtree
    BToDLL(root->right, head_ref);

    // insert root into DLL
    root->right = *head_ref;

    // Change left pointer of previous head
    if (*head_ref != NULL)
        (*head_ref)->left = root;

    // Change head of Doubly linked list
    *head_ref = root;

    // Recursively convert left subtree
    BToDLL(root->left, head_ref);
}
```

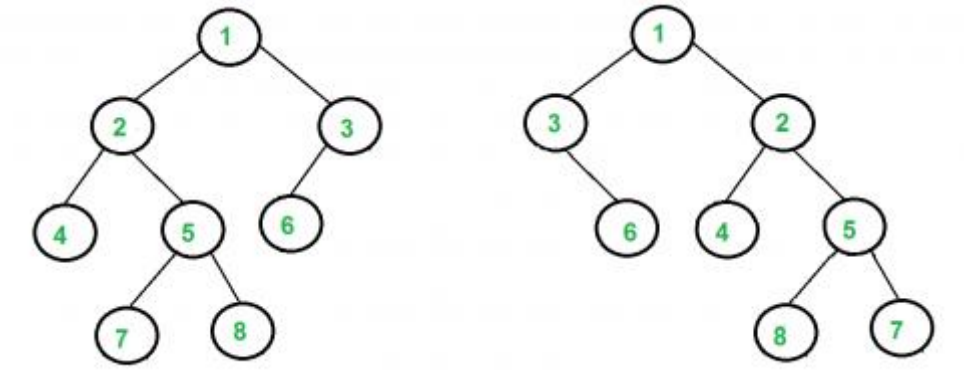
Time Complexity: O(n), as the solution does a single traversal of given Binary Tree.

55 Tree Isomorphism Problem

http://www.geeksforgeeks.org/tree-isomorphism-problem/

Write a function to detect if two trees are isomorphic. Two trees are called isomorphic if one of them can be obtained from other by a series of flips, i.e. by swapping left and right children of a number of nodes. Any number of nodes at any level can have their children swapped. Two empty trees are isomorphic.

For example, following two trees are isomorphic with following sub-trees flipped: 2 and 3, NULL and 6, 7 and 8.



We simultaneously traverse both trees. Let the current internal nodes of two trees being traversed be **n1** and **n2** respectively. There are following two conditions for subtrees rooted with n1 and n2 to be isomorphic.

1) Data of n1 and n2 is same.
2) One of the following two is true for children of n1 and n2
……**a)** Left child of n1 is isomorphic to left child of n2 and right child of n1 is isomorphic to right child of n2.
……**b)** Left child of n1 is isomorphic to right child of n2 and right child of n1 is isomorphic to left child of n2.

```
/* Given a binary tree, print its nodes in reverse level order */
bool isIsomorphic(node* n1, node *n2)
{
 // Both roots are NULL, trees isomorphic by definition
 if (n1 == NULL && n2 == NULL)
    return true;

 // Exactly one of the n1 and n2 is NULL, trees not isomorphic
 if (n1 == NULL || n2 == NULL)
    return false;
```

```
  if (n1->data != n2->data)
    return false;

 // There are two possible cases for n1 and n2 to be isomorphic
 // Case 1: The subtrees rooted at these nodes have NOT been "Flipped".
 // Both of these subtrees have to be isomorphic, hence the &&
 // Case 2: The subtrees rooted at these nodes have been "Flipped"
 return
 (isIsomorphic(n1->left,n2->left) && isIsomorphic(n1->right,n2->right))||
 (isIsomorphic(n1->left,n2->right) && isIsomorphic(n1->right,n2->left));
}
```

Time Complexity: The above solution does a traversal of both trees. So time complexity is O(m + n) where m and n are number of nodes in given trees.

56 Find all possible interpretations of an array of digits

http://www.geeksforgeeks.org/find-all-possible-interpretations/

Consider a coding system for alphabets to integers where 'a' is represented as 1, 'b' as 2, .. 'z' as 26. Given an array of digits (1 to 9) as input, write a function that prints all valid interpretations of input array.

Examples

Input: {1, 1}

Output: ("aa", 'k")

[2 interpretations: aa(1, 1), k(11)]

Input: {1, 2, 1}

Output: ("aba", "au", "la")

[3 interpretations: aba(1,2,1), au(1,21), la(12,1)]

Input: {9, 1, 8}

Output: {"iah", "ir"}

[2 interpretations: iah(9,1,8), ir(9,18)]

Please note we cannot change order of array. That means {1,2,1} cannot become {2,1,1}

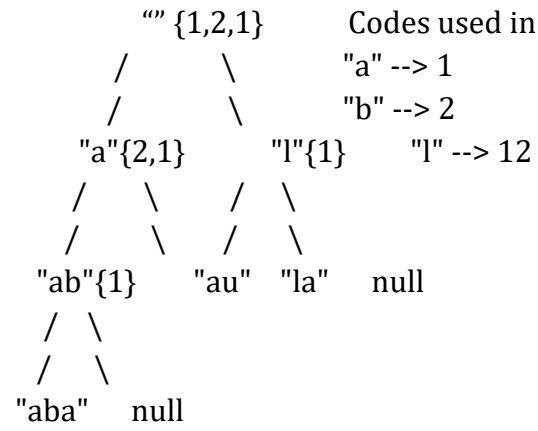
On first look it looks like a problem of permutation/combination. But on closer look you will figure out that this is an interesting tree problem.

The **idea** here is string can take at-most two paths:

1. Proces single digit
2. Process two digits

That means we can use binary tree here. Processing with single digit will be left child and two digits will be right child. If value two digits is greater than 26 then our right child will be null as we don't have alphabet for greater than 26.

Let's understand with an example .Array a = {1,2,1}. Below diagram shows that how our tree grows.



Braces {} contain array still pending for processing. Note that with every level, our array size decreases. If you will see carefully, it is not hard to find that tree height is always n (array size)

How to print all strings (interpretations)? Output strings are leaf node of tree. i.e for {1,2,1}, output is {aba au la}.

We can conclude that there are mainly two steps to print all interpretations of given integer array.

Step 1: Create a binary tree with all possible interpretations in leaf nodes.

Step 2: Print all leaf nodes from the binary tree created in step 1.

```
// Method to create a binary tree which stores all interpretations
// of arr[] in lead nodes
public static Node createTree(int data, String pString, int[] arr) {

    // Invalid input as alphabets maps from 1 to 26
    if (data > 26)
        return null;

    // Parent String + String for this node
    String dataToStr = pString + alphabet[data];

    Node root = new Node(dataToStr);

    // if arr.length is 0 means we are done
    if (arr.length != 0) {
        data = arr[0];
```

```
            // new array will be from index 1 to end as we are consuming
            // first index with this node
            int newArr[] = Arrays.copyOfRange(arr, 1, arr.length);

            // left child
            root.left = createTree(data, dataToStr, newArr);

            // right child will be null if size of array is 0 or 1
            if (arr.length > 1) {

                data = arr[0] * 10 + arr[1];

                // new array will be from index 2 to end as we
                // are consuming first two index with this node
                newArr = Arrays.copyOfRange(arr, 2, arr.length);

                root.right = createTree(data, dataToStr, newArr);
            }
        }
        return root;
    }

    // To print out leaf nodes
    public static void printleaf(Node root) {
        if (root == null)
            return;

        if (root.left == null && root.right == null)
            System.out.print(root.getDataString() + "   ");

        printleaf(root.left);
        printleaf(root.right);
    }

    // The main function that prints all interpretations of array
    static void printAllInterpretations(int[] arr) {

        // Step 1: Create Tree
        Node root = createTree(0, "", arr);

        // Step 2: Print Leaf nodes
        printleaf(root);

        System.out.println();  // Print new line
    }
```

Exercise:

1. What is the time complexity of this solution? [Hint : size of tree + finding leaf nodes]
2. Can we store leaf nodes at the time of tree creation so that no need to run loop again for leaf node fetching?
3. How can we reduce extra space?

57 Iterative Method to find Height of Binary Tree

http://www.geeksforgeeks.org/iterative-method-to-find-height-of-binary-tree/
There are two conventions to define height of Binary Tree
1) Number of nodes on longest path from root to the deepest node.
2) Number of edges on longest path from root to the deepest node.

Recursive method to find height of Binary Tree is discussed here. How to find height without recursion?

We can use level order traversal to find height without recursion.
The **idea** is to traverse level by level. Whenever move down to a level, increment height by 1 (height is initialized as 0). Count number of nodes at each level, stop traversing when count of nodes at next level is 0.
Time Complexity: O(n) where n is number of nodes in given binary tree.

58 Print ancestors of a given binary tree node without recursion

http://www.geeksforgeeks.org/print-ancestors-of-a-given-binary-tree-node-without-recursion/
Given a Binary Tree and a key, write a function that prints all the ancestors of the key in the given binary tree.

It is clear that we need to use a stack based iterative traversal of the Binary Tree. The idea is to have all ancestors in stack when we reach the node with given key. Once we reach the key, all we have to do is, print contents of stack.

How to get all ancestors in stack when we reach the given node? We can traverse all nodes in Postorder way. If we take a closer look at the recursive postorder traversal, we can easily observe that, when recursive function is called for a node, the recursion call stack contains ancestors of the node.

So **idea** is do **iterative Postorder traversal** and stop the traversal when we reach the desired node.

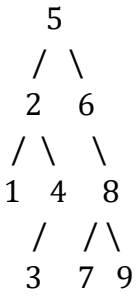
```
// Iterative Function to print all ancestors of a given key
void printAncestors(struct Node *root, int key)
{
    if (root == NULL) return;

    // Create a stack to hold ancestors
    struct Stack* stack = createStack(MAX_SIZE);

    // Traverse the complete tree in postorder way till we find the key
    while (1)
    {
        // Traverse the left side. While traversing, push the nodes into
        // the stack so that their right subtrees can be traversed later
        while (root && root->data != key)
        {
            push(stack, root);   // push current node
            root = root->left;  // move to next node
        }
         // If the node whose ancestors are to be printed is found,
        // then break the while loop.
        if (root && root->data == key)
            break;

        // Check if right sub-tree exists for the node at top
        // If not then pop that node because we don't need this
        // node any more.
        if (peek(stack)->right == NULL)
        {
            root = pop(stack);

          // If the popped node is right child of top, then remove the top
            // as well. Left child of the top must have processed before.
            // Consider the following tree for example and key = 3.  If we
            // remove the following loop, the program will go in an
            // infinite loop after reaching 5.
            //          1
            //        /   \
            //       2     3
            //         \
            //           4
            //             \
            //               5
            while (!isEmpty(stack) && peek(stack)->right == root)
               root = pop(stack);
        }
         // if stack is not empty then simply set the root as right child
        // of top and start traversing right sub-tree.
        root = isEmpty(stack)? NULL: peek(stack)->right;
    }
     // If stack is not empty, print contents of stack
    // Here assumption is that the key is there in tree
    while (!isEmpty(stack))
        printf("%d ", pop(stack)->data);
}
```

Exercise
Note that the above solution assumes that the given key is present in the given Binary Tree. It may go in infinite loop if key is not present. Extend the above solution to work even when the key is not present in tree.

59 Difference between sums of odd level and even level nodes of a Binary Tree

http://www.geeksforgeeks.org/difference-between-sums-of-odd-and-even-levels/
Given a a Binary Tree, find the difference between the sum of nodes at odd level and the sum of nodes at even level. Consider root as level 1, left and right children of root as level 2 and so on.

For example, in the following tree, sum of nodes at odd level is (5 + 1 + 4 + 8) which is 18. And sum of nodes at even level is (2 + 6 + 3 + 7 + 9) which is 27. The output for following tree should be 18 – 27 which is -9.



A **straightforward** method is to **use level order traversal**. In the traversal, check level of current node, if it is odd, increment odd sum by data of current node, otherwise increment even sum. Finally return difference between odd sum and even sum.

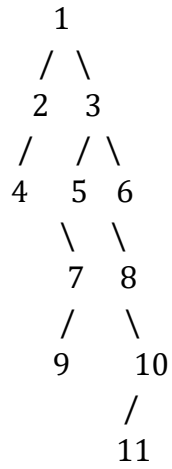
The problem can also be solved using simple **recursive traversal**. We can recursively calculate the required difference as, value of root's data subtracted by the difference for subtree under left child and the difference for subtree under right child.

```
The main function that return difference between odd and even levelnodes
int getLevelDiff(struct node *root)
{
   // Base case
   if (root == NULL)
         return 0;
    // Difference for root is root's data - difference for
   // left subtree - difference for right subtree
   return root->data - getLevelDiff(root->left) -
                                        getLevelDiff(root->right);}
```

Time complexity of both methods is O(n), but the second method is simple and easy to implement.

60 Find depth of the deepest odd level leaf node

http://www.geeksforgeeks.org/find-depth-of-the-deepest-odd-level-node/
Write a C code to get the depth of the deepest odd level leaf node in a binary tree. Consider that level starts with 1. Depth of a leaf node is number of nodes on the path from root to leaf (including both leaf and root).
For example, consider the following tree. The deepest odd level node is the node with value 9 and depth of this node is 5.



The **idea** is to recursively traverse the given binary tree and while traversing, maintain a variable "level" which will store the current node's level in the tree. If current node is leaf then check "level" is odd or not. If **level is odd** then return it. If current node is not leaf, then recursively find maximum depth in left and right subtrees, and return maximum of the two depths

```
// A recursive function to find depth of the deepest odd level leaf
int depthOfOddLeafUtil(Node *root,int level)
{
    // Base Case
    if (root == NULL)
        return 0;

    // If this node is a leaf and its level is odd, return its level
    if (root->left==NULL && root->right==NULL && level&1)
        return level;

    // If not leaf, return the maximum value from left and right subtrees
    return max(depthOfOddLeafUtil(root->left, level+1),
               depthOfOddLeafUtil(root->right, level+1));
}
```

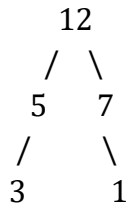
```
/* Main function which calculates the depth of deepest odd level leaf.
  This function mainly uses depthOfOddLeafUtil() */
int depthOfOddLeaf(struct Node *root)
{
    int level = 1, depth = 0;
    return depthOfOddLeafUtil(root, level);
}
```

Time Complexity: The function does a simple traversal of the tree, so the complexity is O(n).

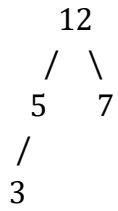
61 Check if all leaves are at same level

http://www.geeksforgeeks.org/check-leaves-level/

Given a Binary Tree, check if all leaves are at same level or not.



Leaves are at same level



Leaves are Not at same level

The **idea** is to first find level of the leftmost leaf and store it in a variable leafLevel. Then compare level of all other leaves with leafLevel, if same, return true, else return false.

We **traverse the given Binary Tree in Preorder fashion**. An argument leaflevel is passed to all calls. The value of leafLevel is initialized as 0 to indicate that the first leaf is not yet seen yet. The value is updated when we find first leaf. Level of subsequent leaves (in preorder) is compared with leafLevel

```
/* Recursive function which checks whether all leaves are at same level */
bool checkUtil(struct Node *root, int level, int *leafLevel)
{
    // Base case
    if (root == NULL)  return true;

    // If a leaf node is encountered
    if (root->left == NULL && root->right == NULL)
    {
        // When a leaf node is found first time
        if (*leafLevel == 0)
        {
            *leafLevel = level; // Set first found leaf's level
            return true;
        }

        // If this is not first leaf node, compare its level with
        // first leaf's level
        return (level == *leafLevel);
    }

    // If this node is not leaf, recursively check left and right subtrees
    return checkUtil(root->left, level+1, leafLevel) &&
           checkUtil(root->right, level+1, leafLevel);
}

/* The main function to check if all leafs are at same level.
   It mainly uses checkUtil() */
bool check(struct Node *root)
{
   int level = 0, leafLevel = 0;
   return checkUtil(root, level, &leafLevel);
}
```

Time Complexity: The function does a simple traversal of the tree, so the complexity is O(n).

62 Print Left View of a Binary Tree

http://www.geeksforgeeks.org/print-left-view-binary-tree/

Given a Binary Tree, print left view of it. Left view of a Binary Tree is set of nodes visible when tree is visited from left side. Left view of following tree is 12, 10, 25.

```
      12
    /   \
  10     30
     /  \
    25   40
```

The left view contains all nodes that are first nodes in their levels. A **simple solution** is to do level order traversal and print the first node in every level.

The problem can also be solved using simple recursive traversal. We can keep track of level of a node by passing a parameter to all recursive calls.

The **idea** is to **keep track of maximum level** also. Whenever we see a node whose level is more than maximum level so far, we print the node because this is the first node in its level (Note that we traverse the left subtree before right subtree).

```
// Recursive function to print left view of a binary tree.
void leftViewUtil(struct node *root, int level, int *max_level)
{
    // Base Case
    if (root==NULL)  return;

    // If this is the first node of its level
    if (*max_level < level)
    {
        printf("%d\t", root->data);
        *max_level = level;
    }

    // Recur for left and right subtrees
    leftViewUtil(root->left, level+1, max_level);
    leftViewUtil(root->right, level+1, max_level);
}

// A wrapper over leftViewUtil()
void leftView(struct node *root)
{
    int max_level = 0;
    leftViewUtil(root, 1, &max_level);
}
```

Time Complexity: The function does a simple traversal of the tree, so the complexity is O(n).

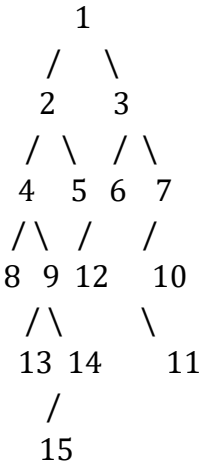
63 Remove all nodes which don't lie in any path with sum>= k

http://www.geeksforgeeks.org/remove-all-nodes-which-lie-on-a-path-having-sum-less-than-k/

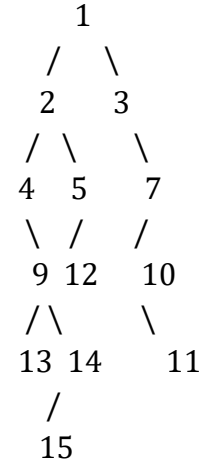
Given a binary tree, a complete path is defined as a path from root to a leaf. The sum of all nodes on that path is defined as the sum of that path. Given a number K, you have to remove (prune the tree) all nodes which don't lie in any path with sum>=k.

Note: A node can be part of multiple paths. So we have to delete it only in case when all paths from it have sum less than K.

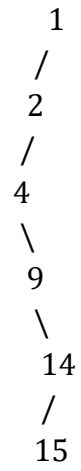
Consider the following Binary Tree



For input k = 20, the tree should be changed to following
(Nodes with values 6 and 8 are deleted)



For input k = 45, the tree should be changed to following.



the fact that nodes are deleted in bottom up manner. The **idea** is to **keep reducing the sum when traversing down**. When we reach a leaf and sum is greater than the leaf's data, then we delete the leaf. Note that deleting nodes may convert a non-leaf node to a leaf node and if the data for the converted leaf node is less than the current sum, then the converted leaf should also be deleted.

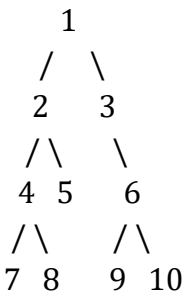
```
struct Node *pruneUtil(struct Node *root, int k, int *sum)
{
    // Base Case
    if (root == NULL)  return NULL;

    // Initialize left and right sums as sum from root to
    // this node (including this node)
    int lsum = *sum + (root->data);
    int rsum = lsum;
     // Recursively prune left and right subtrees
    root->left = pruneUtil(root->left, k, &lsum);
    root->right = pruneUtil(root->right, k, &rsum);
     // Get the maximum of left and right sums
    *sum = max(lsum, rsum);
    // If maximum is smaller than k, then this node
    // must be deleted
    if (*sum < k)
    {
        free(root);
        root = NULL;
    }
     return root;
}
// A wrapper over pruneUtil()
struct Node *prune(struct Node *root, int k){
    int sum = 0;
    return pruneUtil(root, k, &sum);
}
```

64 Extract Leaves of a Binary Tree in a Doubly Linked List

http://www.geeksforgeeks.org/connect-leaves-doubly-linked-list/

Given a Binary Tree, extract all leaves of it in a Doubly Linked List (DLL). Note that the DLL need to be created in-place. Assume that the node structure of DLL and Binary Tree is same, only the meaning of left and right pointers are different. In DLL, left means previous pointer and right means next pointer.

Let the following be input binary tree

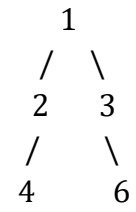


Output:

Doubly Linked List

7<->8<->5<->9<->10

Modified Tree:



We need to traverse all leaves and connect them by changing their left and right pointers. We also need to remove them from Binary Tree by changing left or right pointers in parent nodes.

There can be many ways to solve this. In the following implementation, we add leaves at the beginning of current linked list and update head of the list using pointer to head pointer.

Since we insert at the beginning, we need to process leaves in reverse order. For reverse order, we first traverse the right subtree then the left subtree. We use return values to update left or right pointers in parent nodes.

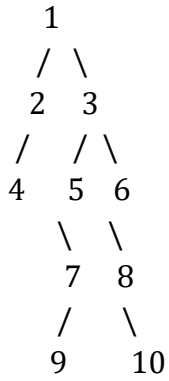
```
// Main function which extracts all leaves from given Binary Tree.
// The function returns new root of Binary Tree (Note that root may change
// if Binary Tree has only one node).  The function also sets *head_ref as
// head of doubly linked list.  left pointer of tree is used as prev in
DLL
// and right pointer is used as next
struct Node* extractLeafList(struct Node *root, struct Node **head_ref)
{
   // Base cases
   if (root == NULL)  return NULL;

   if (root->left == NULL && root->right == NULL)
   {
        // This node is going to be added to doubly linked list
        // of leaves, set right pointer of this node as previous
        // head of DLL. We don't need to set left pointer as left
        // is already NULL
        root->right = *head_ref;

        // Change left pointer of previous head
        if (*head_ref != NULL) (*head_ref)->left = root;

        // Change head of linked list
        *head_ref = root;

        return NULL;  // Return new root
   }

   // Recur for right and left subtrees
   root->right = extractLeafList(root->right, head_ref);
   root->left  = extractLeafList(root->left, head_ref);

   return root;
}
```

Time Complexity: O(n), the solution does a single traversal of given Binary Tree.

65 Deepest left leaf node in a binary tree

http://www.geeksforgeeks.org/deepest-left-leaf-node-in-a-binary-tree/

Given a Binary Tree, find the deepest leaf node that is left child of its parent. For example, consider the following tree. The deepest left leaf node is the node with value 9.



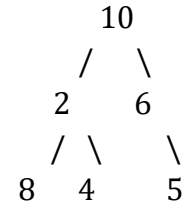
The idea is to recursively traverse the given binary tree and while traversing, maintain "level" which will store the current node's level in the tree. If current node is left leaf, then check if its level is more than the level of deepest left leaf seen so far. If level is more then update the result. If current node is not leaf, then recursively find maximum depth in left and right subtrees, and return maximum of the two depths

```
// A utility function to find deepest leaf node.
// lvl:  level of current node.
// maxlvl: pointer to the deepest left leaf node found so far
// isLeft: A bool indicate that this node is left child of its parent
// resPtr: Pointer to the result
void deepestLeftLeafUtil(Node *root, int lvl, int *maxlvl,
                         bool isLeft, Node **resPtr)
{
    // Base case
    if (root == NULL)
        return;
     // Update result if this node is left leaf and its level is more
    // than the maxl level of the current result
    if (isLeft && !root->left && !root->right && lvl > *maxlvl)
    {
        *resPtr = root;
        *maxlvl = lvl;
        return;
    }
     // Recur for left and right subtrees
    deepestLeftLeafUtil(root->left, lvl+1, maxlvl, true, resPtr);
    deepestLeftLeafUtil(root->right, lvl+1, maxlvl, false, resPtr);
}
 // A wrapper over deepestLeftLeafUtil().
Node* deepestLeftLeaf(Node *root)
{
    int maxlevel = 0;
    Node *result = NULL;
    deepestLeftLeafUtil(root, 0, &maxlevel, false, &result);
    return result;
}
```

Time Complexity: The function does a simple traversal of the tree, so the complexity is O(n).

66 Find next right node of a given key

http://www.geeksforgeeks.org/find-next-right-node-of-a-given-key/
Given a Binary tree and a key in the binary tree, find the node right to the given key. If there is no node on right side, then return NULL. Expected time complexity is O(n) where n is the number of nodes in the given binary tree.
For example, consider the following Binary Tree. Output for 2 is 6, output for 4 is 5. Output for 10, 6 and 5 is NULL.

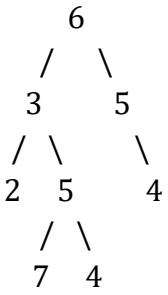


Solution: The **idea** is to do **level order traversal** of given Binary Tree. When we find the given key, we just check if the next node in level order traversal is of same level, if yes, we return the next node, otherwise return NULL.
Time Complexity: The above code is a simple BFS traversal code which visits every enqueue and dequeues a node at most once. Therefore, the time complexity is O(n) where n is the number of nodes in the given binary tree.

Exercise: Write a function to find left node of a given node. If there is no node on the left side, then return NULL.

67 Sum of all the numbers that are formed from root to leaf paths

http://www.geeksforgeeks.org/sum-numbers-formed-root-leaf-paths/
Given a binary tree, where every node value is a Digit from 1-9 .Find the sum of all the numbers which are formed from root to leaf paths.
For example consider the following Binary Tree.



There are 4 leaves, hence 4 root to leaf paths:

Path	Number
6->3->2	632
6->3->5->7	6357
6->3->5->4	6354
6->5>4	654

Answer = 632 + 6357 + 6354 + 654 = 13997

The **idea** is to do a preorder traversal of the tree. In the preorder traversal, keep track of the value calculated till the current node, let this value be val. For every node, we update the val as val*10 plus node's data.

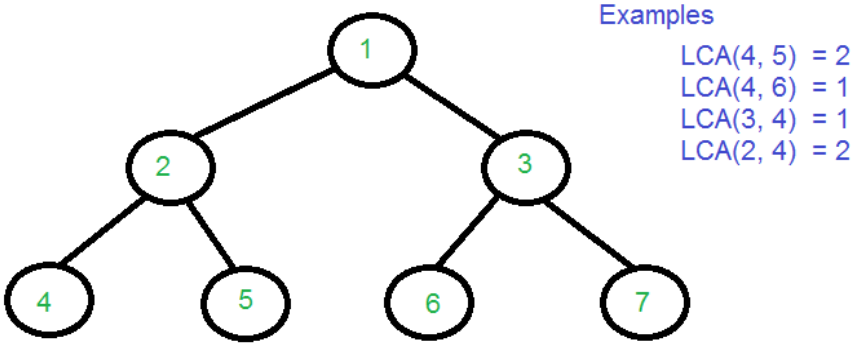
```
// Returns sum of all root to leaf paths. The first parameter is root
// of current subtree, the second parameter is value of the number formed
// by nodes from root to this node
int treePathsSumUtil(struct node *root, int val)
{
    // Base case
    if (root == NULL)  return 0;

    // Update val
    val = (val*10 + root->data);

    // if current node is leaf, return the current value of val
    if (root->left==NULL && root->right==NULL)
       return val;

    // recur sum of values for left and right subtree
    return treePathsSumUtil(root->left, val) +
           treePathsSumUtil(root->right, val);
}

// A wrapper function over treePathsSumUtil()
int treePathsSum(struct node *root)
{
    // Pass the initial value as 0 as there is nothing above root
    return treePathsSumUtil(root, 0);
}
```

Time Complexity: The above code is a simple preorder traversal code which visits every exactly once. Therefore, the time complexity is O(n) where n is the number of nodes in the given binary tree.

68 Lowest Common Ancestor in a Binary Tree | Set 1

http://www.geeksforgeeks.org/lowest-common-ancestor-binary-tree-set-1/
Given a binary tree (not a binary search tree) and two values say n1 and n2, write a program to find the least common ancestor.
Following is definition of LCA from Wikipedia:
Let T be a rooted tree. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself).

The LCA of n1 and n2 in T is the shared ancestor of n1 and n2 that is located farthest from the root. Computation of lowest common ancestors may be useful, for instance, as part of a procedure for determining the distance between pairs of nodes in a tree: the distance from n1 to n2 can be computed as the distance from the root to n1, plus the distance from the root to n2, minus twice the distance from the root to their lowest common ancestor. (Source Wiki)



We have discussed an efficient solution to find LCA in Binary Search Tree. In Binary Search Tree, using BST properties, we can find LCA in O(h) time where h is height of tree. Such an implementation is not possible in Binary Tree as keys Binary Tree nodes don't follow any order. Following are different approaches to find LCA in Binary Tree.

Method 1 (By Storing root to n1 and root to n2 paths):
Following is simple O(n) algorithm to find LCA of n1 and n2.
1) Find path from root to n1 and store it in a vector or array.
2) Find path from root to n2 and store it in another vector or array.
3) Traverse both paths till the values in arrays are same. Return the common element just before the mismatch.
Time Complexity: Time complexity of the above solution is O(n). The tree is traversed twice, and then path arrays are compared.

Method 2 (Using Single Traversal)

The method 1 finds LCA in O(n) time, but requires three tree traversals plus extra spaces for path arrays. If we assume that the keys n1 and n2 are present in Binary Tree, we can find LCA using single traversal of Binary Tree and without extra storage for path arrays.

The **idea** is to traverse the tree starting from root. If any of the given keys (n1 and n2) matches with root, then root is LCA (assuming that both keys are present). If root doesn't match with any of the keys, we recur for left and right subtree. The node which has one key present in its left subtree and the other key present in right subtree is the LCA. If both keys lie in left subtree, then left subtree has LCA also, otherwise LCA lies in right subtree.

```
// This function returns pointer to LCA of two given values n1 and n2.
// This function assumes that n1 and n2 are present in Binary Tree
struct Node *findLCA(struct Node* root, int n1, int n2)
{
    // Base case
    if (root == NULL) return NULL;

    // If either n1 or n2 matches with root's key, report
    // the presence by returning root (Note that if a key is
    // ancestor of other, then the ancestor key becomes LCA
    if (root->key == n1 || root->key == n2)
        return root;

    // Look for keys in left and right subtrees
    Node *left_lca  = findLCA(root->left, n1, n2);
    Node *right_lca = findLCA(root->right, n1, n2);

    // If both of the above calls return Non-NULL, then one key
    // is present in once subtree and other is present in other,
    // So this node is the LCA
    if (left_lca && right_lca)  return root;

    // Otherwise check if left subtree or right subtree is LCA
    return (left_lca != NULL)? left_lca: right_lca;
}
```

Time Complexity: Time complexity of the above solution is O(n) as the method does a simple tree traversal in bottom up fashion.

Note that the above method assumes that keys are present in Binary Tree. If one key is present and other is absent, then it returns the present key as LCA (Ideally should have returned NULL).

We can **extend** this method to handle all cases by passing two Boolean variables v1 and v2. v1 is set as true when n1 is present in tree and v2 is set as true if n2 is present in tree

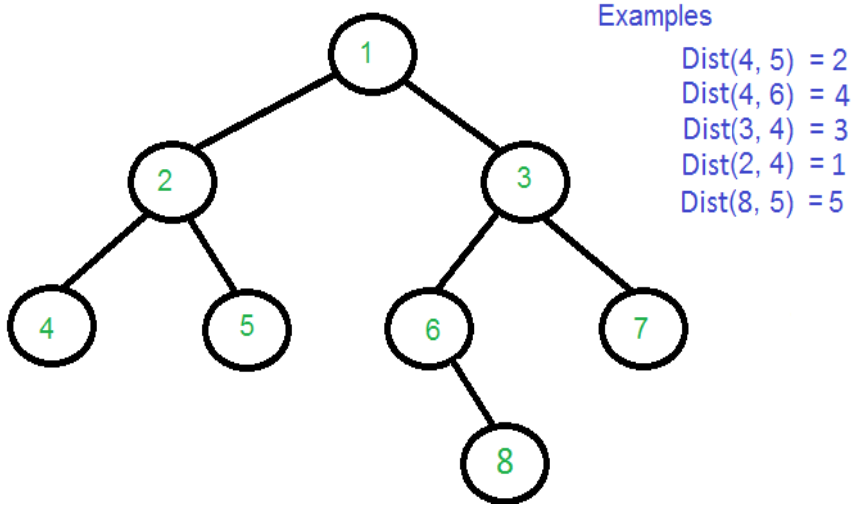
```
// This function returns pointer to LCA of two given values n1 and n2.
// v1 is set as true by this function if n1 is found
// v2 is set as true by this function if n2 is found
struct Node *findLCAUtil(struct Node* root, int n1, int n2, bool &v1, bool
&v2)
{
    // Base case
    if (root == NULL) return NULL;

    // If either n1 or n2 matches with root's key, report the presence
    // by setting v1 or v2 as true and return root (Note that if a key
    // is ancestor of other, then the ancestor key becomes LCA)
    if (root->key == n1)
    {
        v1 = true;
        return root;
    }
    if (root->key == n2)
    {
        v2 = true;
        return root;
    }

    // Look for keys in left and right subtrees
    Node *left_lca  = findLCAUtil(root->left, n1, n2, v1, v2);
    Node *right_lca = findLCAUtil(root->right, n1, n2, v1, v2);

    // If both of the above calls return Non-NULL, then one key
    // is present in once subtree and other is present in other,
    // So this node is the LCA
    if (left_lca && right_lca)  return root;

    // Otherwise check if left subtree or right subtree is LCA
    return (left_lca != NULL)? left_lca: right_lca;
}

// Returns true if key k is present in tree rooted with root
bool find(Node *root, int k)
{
    // Base Case
    if (root == NULL)
        return false;

    // If key is present at root, or in left subtree or right subtree,
    // return true;
    if (root->key == k || find(root->left, k) ||  find(root->right, k))
        return true;

    // Else return false
```

```
    return false;
}

// This function returns LCA of n1 and n2 only if both n1 and n2 are
present
// in tree, otherwise returns NULL;
Node *findLCA(Node *root, int n1, int n2)
{
    // Initialize n1 and n2 as not visited
    bool v1 = false, v2 = false;

    // Find lca of n1 and n2 using the technique discussed above
    Node *lca = findLCAUtil(root, n1, n2, v1, v2);

    // Return LCA only if both n1 and n2 are present in tree
    if (v1 && v2 || v1 && find(lca, n2) || v2 && find(lca, n1))
        return lca;

    // Else return NULL
    return NULL;
}
```

69 Find distance between two given keys of a Binary Tree

http://www.geeksforgeeks.org/find-distance-two-given-nodes/

Find the distance between two keys in a binary tree, no parent pointers are given. Distance between two nodes is the minimum number of edges to be traversed to reach one node from other.



The distance between two nodes can be obtained in terms of lowest common ancestor. Following is the formula.

Dist(n1, n2) = Dist(root, n1) + Dist(root, n2) - 2*Dist(root, lca)

'n1' and 'n2' are the two given keys
'root' is root of given Binary Tree.
'lca' is lowest common ancestor of n1 and n2
Dist(n1, n2) is the distance between n1 and n2.

Following is the implementation of above approach. The implementation is adopted from last code provided in Lowest Common Ancestor Post.

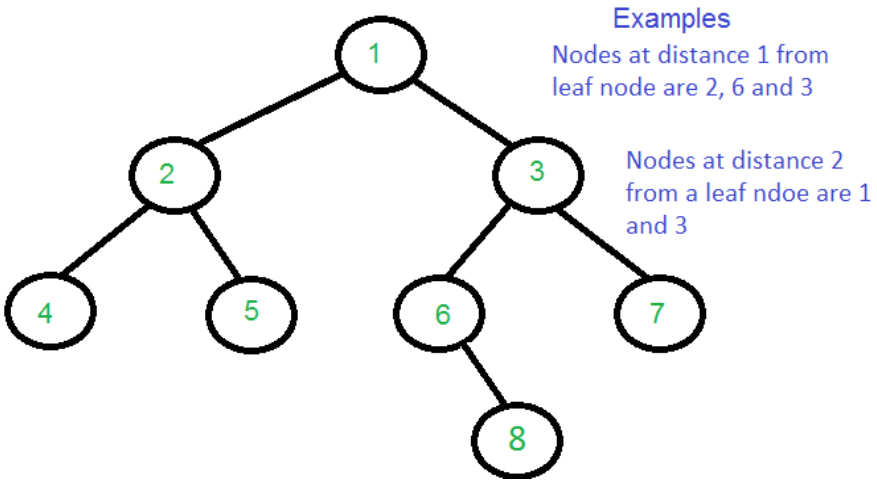
```
// This function returns pointer to LCA of two given values n1 and n2.
// It also sets d1, d2 and dist if one key is not ancestor of other
// d1 --> To store distance of n1 from root
// d2 --> To store distance of n2 from root
// lvl --> Level (or distance from root) of current node
// dist --> To store distance between n1 and n2
Node *findDistUtil(Node* root, int n1, int n2, int &d1, int &d2,
                   int &dist, int lvl)
{
    // Base case
    if (root == NULL) return NULL;

    // If either n1 or n2 matches with root's key, report
    // the presence by returning root (Note that if a key is
    // ancestor of other, then the ancestor key becomes LCA
    if (root->key == n1)
    {
         d1 = lvl;
         return root;
    }
    if (root->key == n2)
    {
         d2 = lvl;
         return root;
    }

    // Look for n1 and n2 in left and right subtrees
 Node *left_lca  = findDistUtil(root->left, n1, n2, d1, d2, dist, lvl+1);
 Node *right_lca = findDistUtil(root->right, n1, n2, d1, d2, dist, lvl+1);

    // If both of the above calls return Non-NULL, then one key
    // is present in once subtree and other is present in other,
    // So this node is the LCA
    if (left_lca && right_lca)
    {
        dist = d1 + d2 - 2*lvl;
        return root;
    }
```

```
     // Otherwise check if left subtree or right subtree is LCA
    return (left_lca != NULL)? left_lca: right_lca;
}

// The main function that returns distance between n1 and n2
// This function returns -1 if either n1 or n2 is not present in
// Binary Tree.
int findDistance(Node *root, int n1, int n2)
{
    // Initialize d1 (distance of n1 from root), d2 (distance of n2
    // from root) and dist(distance between n1 and n2)
    int d1 = -1, d2 = -1, dist;
    Node *lca = findDistUtil(root, n1, n2, d1, d2, dist, 1);

    // If both n1 and n2 were present in Binary Tree, return dist
    if (d1 != -1 && d2 != -1)
        return dist;

    // If n1 is ancestor of n2, consider n1 as root and find level
    // of n2 in subtree rooted with n1
    if (d1 != -1)
    {
        dist = findLevel(lca, n2, 0);
        return dist;
    }

    // If n2 is ancestor of n1, consider n2 as root and find level
    // of n1 in subtree rooted with n2
    if (d2 != -1)
    {
        dist = findLevel(lca, n1, 0);
        return dist;
    }

    return -1;
}
```

Time Complexity: Time complexity of the above solution is O(n) as the method does a single tree traversal.

70 Print all nodes that are at distance k from a leaf node

http://www.geeksforgeeks.org/print-nodes-distance-k-leaf-node/
Given a Binary Tree and a positive integer k, print all nodes that are distance k from a leaf node.
Here the meaning of distance is different from previous post. Here k distance from a leaf means k levels higher than a leaf node. For example if k is more than height of Binary Tree, then nothing should be printed. Expected time complexity is O(n) where n is the number nodes in the given Binary Tree.

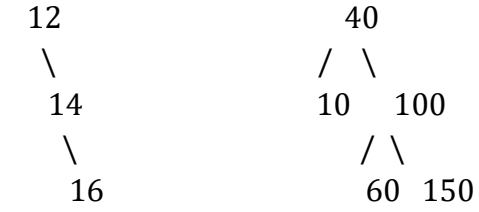


The **idea** is to traverse the tree. **Keep storing all ancestors till we hit a leaf node**. When we reach a leaf node, we print the ancestor at distance k. We also need to keep track of nodes that are already printed as output. For that we use a boolean array visited[].
Time Complexity: Time Complexity of above code is O(n) as the code does a simple tree traversal.

71 Check if a given Binary Tree is height balanced like a Red-Black Tree

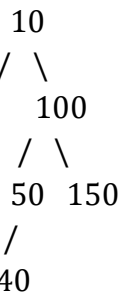
http://www.geeksforgeeks.org/check-given-binary-tree-follows-height-property-red-black-tree/
In a Red-Black Tree, the maximum height of a node is at most twice the minimum height (The four Red-Black tree properties make sure this is always followed). Given a

Binary Search Tree, we need to check for following property.
For every node, length of the longest leaf to node path has not more than twice the nodes on shortest path from node to leaf.



Cannot be a Red-Black Tree with any color assignment
Max height of 12 is 1
Min height of 12 is 3

It can be Red-Black Tree



It can also be Red-Black Tree
Expected **time complexity** is O(n). The tree should be traversed at-most once in the solution.

For every node, we need to get the maximum and minimum heights and compare them. The idea is to traverse the tree and for every node check if it's balanced.

We need to write a recursive function that returns three things, a boolean value to indicate the tree is balanced or not, minimum height and maximum height.

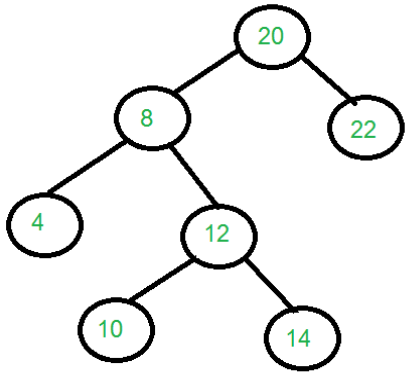
To return multiple values, we can either use a structure or pass variables by reference. We have passed maxh and minh by reference so that the values can be used in parent calls.

```
// Returns returns tree if the Binary tree is balanced like a Red-Black
// tree. This function also sets value in maxh and minh (passed by
// reference). maxh and minh are set as maximum and minimum heights of
root.
bool isBalancedUtil(Node *root, int &maxh, int &minh)
{
    // Base case
    if (root == NULL)
    {
        maxh = minh = 0;
        return true;
    }

    int lmxh, lmnh; // To store max and min heights of left subtree
    int rmxh, rmnh; // To store max and min heights of right subtree

    // Check if left subtree is balanced, also set lmxh and lmnh
    if (isBalancedUtil(root->left, lmxh, lmnh) == false)
        return false;

    // Check if right subtree is balanced, also set rmxh and rmnh
    if (isBalancedUtil(root->right, rmxh, rmnh) == false)
        return false;

    // Set the max and min heights of this node for the parent call
    maxh = max(lmxh, rmxh) + 1;
    minh = min(lmnh, rmnh) + 1;

    // See if this node is balanced
    if (maxh <= 2*minh)
        return true;

    return false;
}

// A wrapper over isBalancedUtil()
bool isBalanced(Node *root)
{
    int maxh, minh;
    return isBalancedUtil(root, maxh, minh);
}
```

Time Complexity: Time Complexity of above code is O(n) as the code does a simple tree traversal.

72 Print all nodes at distance k from a given node

http://www.geeksforgeeks.org/print-nodes-distance-k-given-node-binary-tree/
Given a binary tree, a target node in the binary tree, and an integer value k, print all the nodes that are at distance k from the given target node. No parent pointers are available.



Consider the tree shown in diagram

Input: target = pointer to node with data 8.
root = pointer to node with data 20.
k = 2.
Output : 10 14 22

If target is 14 and k is 3, then output
should be "4 20"

There are two types of nodes to be considered.
1) Nodes in the subtree rooted with target node. For example if the target node is 8 and k is 2, then such nodes are 10 and 14.

2) Other nodes, may be an ancestor of target, or a node in some other subtree. For target node 8 and k is 2, the node 22 comes in this category.
Finding the first type of nodes is easy to implement. Just traverse subtrees rooted with the target node and decrement k in recursive call. When the k becomes 0, print the node currently being traversed (See this for more details). Here we call the function as printkdistanceNodeDown().

How to find nodes of second type?

For the output nodes not lying in the subtree with the target node as the root, we must go through all ancestors. For every ancestor, we find its distance from target node, let the distance be d, now we go to other subtree (if target was found in left subtree, then we go to right subtree and vice versa) of the ancestor and find all nodes at k-d distance from the ancestor.

```
/* Recursive function to print all the nodes at distance k in the
   tree (or subtree) rooted with given root. See  */
void printkdistanceNodeDown(node *root, int k)
{
    // Base Case
    if (root == NULL || k < 0)  return;

    // If we reach a k distant node, print it
    if (k==0)
    {
        cout << root->data << endl;
        return;
    }

    // Recur for left and right subtrees
    printkdistanceNodeDown(root->left, k-1);
    printkdistanceNodeDown(root->right, k-1);
}

// Prints all nodes at distance k from a given target node.
// The k distant nodes may be upward or downward.  This function
// Returns distance of root from target node, it returns -1 if target
// node is not present in tree rooted with root.
int printkdistanceNode(node* root, node* target , int k)
{
    // Base Case 1: If tree is empty, return -1
    if (root == NULL) return -1;

    // If target is same as root.  Use the downward function
    // to print all nodes at distance k in subtree rooted with
    // target or root
    if (root == target)
    {
        printkdistanceNodeDown(root, k);
        return 0;
    }
     // Recur for left subtree
    int dl = printkdistanceNode(root->left, target, k);
     // Check if target node was found in left subtree
    if (dl != -1)
    {
         // If root is at distance k from target, print root
         // Note that dl is Distance of root's left child from target
         if (dl + 1 == k)
            cout << root->data << endl;
```

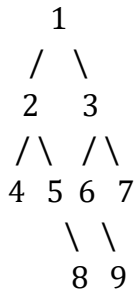
```
            // Else go to right subtree and print all k-dl-2 distant nodes
            // Note that the right child is 2 edges away from left child
            else
                printkdistanceNodeDown(root->right, k-dl-2);

            // Add 1 to the distance and return value for parent calls
            return 1 + dl;
    }

    // MIRROR OF ABOVE CODE FOR RIGHT SUBTREE
    // Note that we reach here only when node was not found in left
subtree
    int dr = printkdistanceNode(root->right, target, k);
    if (dr != -1)
    {
            if (dr + 1 == k)
                cout << root->data << endl;
            else
                printkdistanceNodeDown(root->left, k-dr-2);
            return 1 + dr;
    }

    // If target was neither present in left nor in right subtree
    return -1;
}
```

Time Complexity: At first look the time complexity looks more than O(n), but if we take a closer look, we can observe that no node is traversed more than twice. Therefore the time complexity is O(n).

73 Print a Binary Tree in Vertical Order | Set 1

http://www.geeksforgeeks.org/print-binary-tree-vertical-order/

Given a binary tree, print it vertically. The following example illustrates vertical order traversal.



The output of print this tree vertically will be:
4
2
1 5 6
3 8
7
9
The **idea** is to traverse the tree once and get the minimum and maximum horizontal distance with respect to root. For the tree shown above, minimum distance is -2 (for node with value 4) and maximum distance is 3 (For node with value 9).

Once we have maximum and minimum distances from root, we iterate for each vertical line at distance minimum to maximum from root, and for each vertical line traverse the tree and print the nodes which lie on that vertical line.
Time Complexity: Time complexity of above algorithm is O(w*n) where w is width of Binary Tree and n is number of nodes in Binary Tree. In worst case, the value of w can be O(n) (consider a complete tree for example) and time complexity can become O(n2).

Method – 2 (Hashing)
We need to check the Horizontal Distances from root for all nodes. If two nodes have the same Horizontal Distance (HD), then they are on same vertical line. The idea of HD is simple. HD for root is 0, a right edge (edge connecting to right subtree) is considered as +1 horizontal distance and a left edge is considered as -1 horizontal distance. For example, in the above tree, HD for Node 4 is at -2, HD for Node 2 is -1, HD for 5 and 6 is 0 and HD for node 7 is +2.

We can do **inorder traversal of the given Binary Tree**. While traversing the tree, we can recursively calculate HDs. We initially pass the horizontal distance as 0 for root. For left subtree, we pass the Horizontal Distance as Horizontal distance of root minus 1. For right subtree, we pass the Horizontal Distance as Horizontal Distance of root plus 1.

```
/ A wrapper over VerticalSumUtil()
    private void VerticalSum(TreeNode root) {

        // base case
        if (root == null) { return; }

        // Creates an empty hashMap hM
        HashMap<Integer, Integer> hM = new HashMap<Integer, Integer>();
```

```
     // Calls the VerticalSumUtil() to store the vertical sum values in hM
        VerticalSumUtil(root, 0, hM);

        // Prints the values stored by VerticalSumUtil()
        if (hM != null) {
            System.out.println(hM.entrySet());
        }
    }

    // Traverses the tree in Inoorder form and builds a hashMap hM that
    // contains the vertical sum
    private void VerticalSumUtil(TreeNode root, int hD,
                                        HashMap<Integer, Integer> hM) {

        // base case
        if (root == null) {  return; }

        // Store the values in hM for left subtree
        VerticalSumUtil(root.left(), hD - 1, hM);

        // Update vertical sum for hD of this node
        int prevSum = (hM.get(hD) == null) ? 0 : hM.get(hD);
        hM.put(hD, prevSum + root.key());

        // Store the values in hM for right subtree
        VerticalSumUtil(root.right(), hD + 1, hM);
    }
}
```

Time Complexity: O(n)

74 Vertical Sum in Binary Tree | Set 2 (Space Optimized)

http://www.geeksforgeeks.org/vertical-sum-in-binary-tree-set-space-optimized/

Given a Binary Tree, find vertical sum of the nodes that are in same vertical line. Print all sums through different vertical lines.

Examples:

```
    1
  /  \
 2    3
/\   /\
4  5 6  7
```

The tree has 5 vertical lines

Vertical-Line-1 has only one node 4 => vertical sum is 4

Vertical-Line-2: has only one node 2=> vertical sum is 2

Vertical-Line-3: has three nodes: 1,5,6 => vertical sum is 1+5+6 = 12

We have discussed **Hashing Based Solution** in Set 1. Hashing based solution requires a Hash Table to be maintained. We know that hashing requires more space than the number of entries in it. In this post, Doubly Linked List based solution is discussed. The solution discussed here requires only n nodes of linked list where n is total number of vertical lines in binary tree. Below is algorithm.

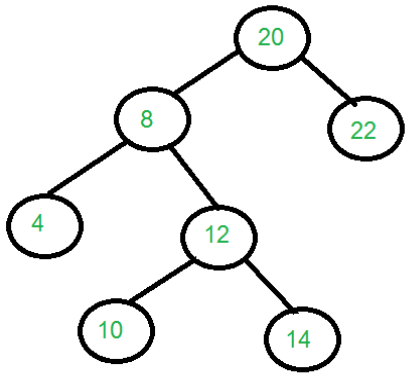
```
/* Recursive function to print all the nodes at distance k in the
   tree (or subtree) rooted with given root. See  */
void printkdistanceNodeDown(node *root, int k)
{
    // Base Case
    if (root == NULL || k < 0)  return;

    // If we reach a k distant node, print it
    if (k==0)
    {
        cout << root->data << endl;
        return;
    }

    // Recur for left and right subtrees
    printkdistanceNodeDown(root->left, k-1);
    printkdistanceNodeDown(root->right, k-1);
}

// Prints all nodes at distance k from a given target node.
// The k distant nodes may be upward or downward.  This function
// Returns distance of root from target node, it returns -1 if target
// node is not present in tree rooted with root.
int printkdistanceNode(node* root, node* target , int k)
{
    // Base Case 1: If tree is empty, return -1
    if (root == NULL) return -1;

    // If target is same as root.  Use the downward function
    // to print all nodes at distance k in subtree rooted with
    // target or root
    if (root == target)
    {
        printkdistanceNodeDown(root, k);
        return 0;
    }

    // Recur for left subtree
    int dl = printkdistanceNode(root->left, target, k);

    // Check if target node was found in left subtree
    if (dl != -1)
    {
        // If root is at distance k from target, print root
        // Note that dl is Distance of root's left child from target
        if (dl + 1 == k)
            cout << root->data << endl;
```

```
            // Else go to right subtree and print all k-dl-2 distant nodes
            // Note that the right child is 2 edges away from left child
            else
                printkdistanceNodeDown(root->right, k-dl-2);

            // Add 1 to the distance and return value for parent calls
            return 1 + dl;
        }
        // MIRROR OF ABOVE CODE FOR RIGHT SUBTREE
        // Note that we reach here only when node was not found in left
subtree
        int dr = printkdistanceNode(root->right, target, k);
        if (dr != -1)
        {
            if (dr + 1 == k)
                cout << root->data << endl;
            else
                printkdistanceNodeDown(root->left, k-dr-2);
            return 1 + dr;
        }

        // If target was neither present in left nor in right subtree
        return -1;
}
```

75 Construct a tree from Inorder and Level order traversals

http://www.geeksforgeeks.org/construct-tree-inorder-level-order-traversals/

Given inorder and level-order traversals of a Binary Tree, construct the Binary Tree. Following is an example to illustrate the problem.



Input: Two arrays that represent Inorder
and level order traversals of a
Binary Tree

in[] = {4, 8, 10, 12, 14, 20, 22};

level[] = {20, 8, 22, 4, 12, 10, 14};

Output: Construct the tree represented
by the two arrays.
For the above two arrays, the
constructed tree is shown in
the diagram on right side

Let us consider the above example.
in[] = {4, 8, 10, 12, 14, 20, 22};
level[] = {20, 8, 22, 4, 12, 10, 14};

In a **Level order sequence**, the first element is the root of the tree. So we know '20' is root for given sequences. By searching '20' in Inorder sequence, we can find out all elements on left side of '20' are in left subtree and elements on right are in right subtree. So we know below structure now.

```
        20
       /   \
      /     \
{4,8,10,12,14}  {22}
```

Let us call {4,8,10,12,14} as left subarray in Inorder traversal and {22} as right subarray in Inorder traversal.

In level order traversal, keys of left and right subtrees are not consecutive. So we extract all nodes from level order traversal which are in left subarray of Inorder traversal. To construct the left subtree of root, we recur for the extracted elements from level order traversal and left subarray of inorder traversal. In the above example, we recur for following two arrays.

// Recur for following arrays to construct the left subtree
In[] = {4, 8, 10, 12, 14}
level[] = {8, 4, 12, 10, 14}

Similarly, we recur for following two arrays and construct the right subtree.
// Recur for following arrays to construct the right subtree
In[] = {22}
level[] = {22}

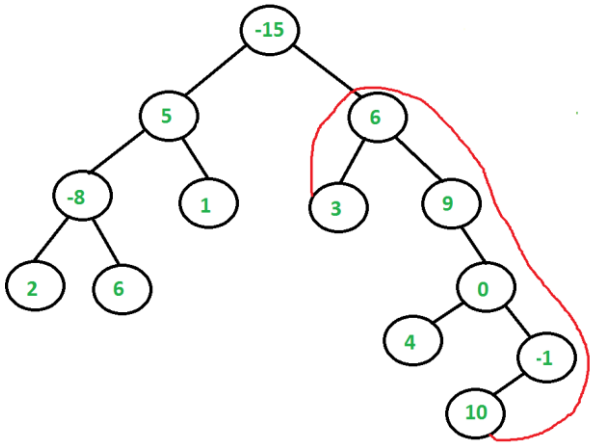
```
/* Recursive function to construct binary tree of size n from
   Inorder traversal in[] and Level Order traversal level[].
   inSrt and inEnd are start and end indexes of array in[]
   Initial values of inStrt and inEnd should be 0 and n -1.
   The function doesn't do any error checking for cases
   where inorder and levelorder do not form a tree */
Node* buildTree(int in[], int level[], int inStrt, int inEnd, int n)
{

    // If start index is more than the end index
    if (inStrt > inEnd)
        return NULL;

    /* The first node in level order traversal is root */
    Node *root = newNode(level[0]);

    /* If this node has no children then return */
    if (inStrt == inEnd)
        return root;

    /* Else find the index of this node in Inorder traversal */
    int inIndex = search(in, inStrt, inEnd, root->key);

    // Extract left subtree keys from level order traversal
    int *llevel  = extrackKeys(in, level, inIndex, n);

    // Extract right subtree keys from level order traversal
    int *rlevel  = extrackKeys(in + inIndex + 1, level, n-inIndex-1, n);

    /* construct left and right subtress */
    root->left = buildTree(in, llevel, inStrt, inIndex-1, n);
    root->right = buildTree(in, rlevel, inIndex+1, inEnd, n);

    // Free memory to avoid memory leak
    delete [] llevel;
    delete [] rlevel;

    return root;
}
```

An upper bound on **time complexity** of above method is O(n3). In the main recursive function, extractNodes() is called which takes O(n2) time.

The code can be **optimized** in many ways and there may be better solutions. Looking for improvements and other optimized approaches to solve this problem.

76 Find the maximum path sum between two leaves of a binary tree

http://www.geeksforgeeks.org/find-maximum-path-sum-two-leaves-binary-tree/
Given a binary tree in which each node element contains a number. Find the maximum possible sum from one leaf node to another.
The maximum sum path may or may not go through root. For example, in the following binary tree, the maximum sum is 27(3 + 6 + 9 + 0 – 1 + 10). Expected time complexity is O(n).
If one side of root is empty, then function should return minus infinite (INT_MIN in case of C/C++)



A **simple solution** is to traverse the tree and do following for every traversed node X.
1) Find maximum sum from leaf to root in left subtree of X (we can use this post for this and next steps)
2) Find maximum sum from leaf to root in right subtree of X.
3) Add the above two calculated values and X->data and compare the sum with the maximum value obtained so far and update the maximum value.
4) Return the maximum value.

The **time complexity** of above solution is O(n2)

We can find the maximum sum using **single traversal** of binary tree. The idea is to maintain two values in recursive calls
1) Maximum root to leaf path sum for the subtree rooted under current node.
2) The maximum path sum between leaves (desired output).

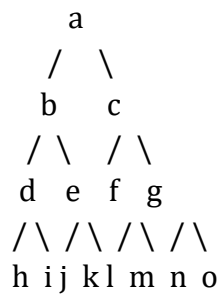
For every visited node X, we find the maximum root to leaf sum in left and right subtrees of X. We add the two values with X->data, and compare the sum with maximum path sum found so far.
Following is the implementation of the above O(n) solution.

```
// A utility function to find the maximum sum between any
// two leaves.This function calculates two values:
// 1) Maximum path sum between two leaves which is stored
//    in res.
// 2) The maximum root to leaf path sum which is returned.
// If one side of root is empty, then it returns INT_MIN
int maxPathSumUtil(struct Node *root, int &res)
{
    // Base cases
    if (root==NULL) return 0;
    if (!root->left && !root->right) return root->data;

    // Find maximum sum in left and right subtree. Also
    // find maximum root to leaf sums in left and right
    // subtrees and store them in ls and rs
    int ls = maxPathSumUtil(root->left, res);
    int rs = maxPathSumUtil(root->right, res);


    // If both left and right children exist
    if (root->left && root->right)
    {
        // Update result if needed
        res = max(res, ls + rs + root->data);

        // Return maxium possible value for root being
        // on one side
        return max(ls, rs) + root->data;
    }

    // If any of the two children is empty, return
    // root sum for root being on one side
    return (!root->left)? rs + root->data:
                          ls + root->data;
}
```

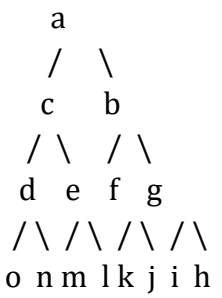
77 Reverse alternate levels of a perfect binary tree

http://www.geeksforgeeks.org/reverse-alternate-levels-binary-tree/
Given a Perfect Binary Tree, reverse the alternate level nodes of the binary tree.

Given tree:



Modified tree:



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.

```
// 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);
}
```

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

Method 3 (Using One Traversal)

```
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);
}
```

78 Check if two nodes are cousins in a Binary Tree

http://www.geeksforgeeks.org/check-two-nodes-cousins-binary-tree/
Given the binary Tree and the two nodes say 'a' and 'b', determine whether the two nodes are cousins of each other or not.
Two nodes are cousins of each other if they are at same level and have different parents.
Example

```
   6
 /  \
 3   5
/\  /\
7  8 1  3
```

Say two node be 7 and 1, result is TRUE.
Say two nodes are 3 and 5, result is FALSE.
Say two nodes are 7 and 5, result is FALSE.

The **idea** is to find level of one of the nodes. Using the found level, **check if 'a' and 'b' are at this level. If 'a' and 'b' are at given level**, then finally check if they are not children of same parent.

```
// Returns 1 if a and b are cousins, otherwise 0
int isCousin(struct Node *root, struct Node *a, struct Node *b)
{
    //1. The two Nodes should be on the same level in the binary tree.
    //2. The two Nodes should not be siblings (means that they should
    // not have the same parent Node).
    if ((level(root,a,1) == level(root,b,1)) && !(isSibling(root,a,b)))
        return 1;
    else return 0;
}
```

Time Complexity of the above solution is O(n) as it does at most three traversals of binary tree.

79 Serialize and Deserialize a Binary Tree

http://www.geeksforgeeks.org/serialize-deserialize-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.

If given Tree is Binary Search Tree?
If the given Binary Tree is Binary Search Tree, we can store it by either storing preorder or postorder traversal. In case of Binary Search Trees, only preorder or postorder traversal is sufficient to store structure information.

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 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

Input:

```
   20
  /  \
 8    22
```

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

```
// This function stores a tree in a file pointed by fp
void serialize(Node *root, FILE *fp)
{
    // If current node is NULL, store marker
    if (root == NULL)
    {
        fprintf(fp, "%d ", MARKER);
        return;
    }

    // Else, store current node and recur for its children
    fprintf(fp, "%d ", root->key);
    serialize(root->left, fp);
    serialize(root->right, fp);
}

// This function constructs a tree from a file pointed by 'fp'
void deSerialize(Node *&root, FILE *fp)
{
    // Read next item from file. If theere are no more items or next
    // item is marker, then return
    int val;
    if ( !fscanf(fp, "%d ", &val) || val == MARKER)
       return;

    // Else create node with this item and recur for children
    root = newNode(val);
    deSerialize(root->left, fp);
    deSerialize(root->right, fp);
}
```

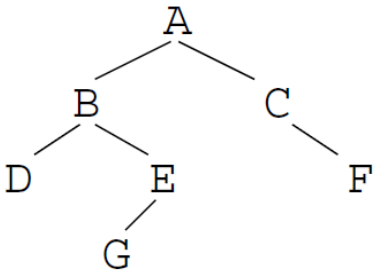
How much extra space is required in above solution?

If there are n keys, then the above solution requires n+1 markers which may be better than simple solution (storing keys twice) in situations where keys are big or keys have big data items associated with them.

Can we optimize it further?

The above solution can be optimized in many ways. If we take a closer look at above serialized trees, we can observer that all leaf nodes require two markers. One simple optimization is to store a separate bit with every node to indicate that the node is internal or external. This way we don't have to store two markers with every leaf node as leaves can be identified by extra bit. We still need marker for internal nodes with

one child. For example in the following diagram ' is used to indicate an internal node set bit, and '/' is used as NULL marker. The diagram is taken from here.



A'B'DE'G/C'/F

Please note that there are always more leaf nodes than internal nodes in a Binary Tree (Number of leaf nodes is number of internal nodes plus 1, so this optimization makes sense.

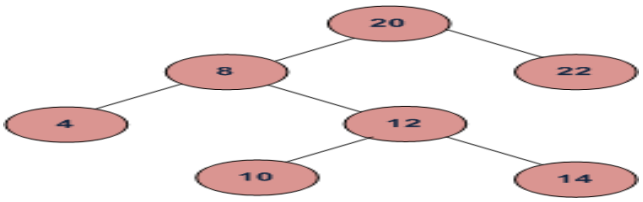
How to serialize n-ary tree?
In an n-ary tree, there is no designated left or right child. We can store an 'end of children' marker with every node. The following diagram shows serialization where ')' is used as end of children marker. We will soon be covering implementation for n-ary tree. The diagram is taken from here.

80 Print nodes between two given level numbers of a binary tree

http://www.geeksforgeeks.org/given-binary-tree-print-nodes-two-given-level-numbers/
Given a binary tree and two level numbers 'low' and 'high', print nodes from level low to level high.
For example consider the binary tree given in below diagram.

Input: Root of below tree, low = 2, high = 4

Output:
8 22
4 12
10 14



A **Simple Method** is to first write a recursive function that prints nodes of a given level number. Then call recursive function in a loop from low to high.
Time complexity of this method is O(n2)

We can print nodes in **O(n) time** using **queue based iterative level order traversal**. The idea is to do simple queue based level order traversal. While doing inorder traversal, add a marker node at the end. Whenever we see a marker node, we increase level number. If level number is between low and high, then print nodes.
Time complexity of above method is O(n) as it does a simple level order traversal.

81 Find the closest leaf in a Binary Tree

http://www.geeksforgeeks.org/find-closest-leaf-binary-tree/
Given a Binary Tree and a key 'k', find distance of the closest leaf from 'k'.
Examples:

```
        A
      /   \
     B     C
         /   \
        E     F
       /       \
      G         H
     / \       /
    I   J     K
```

Closest leaf to 'H' is 'K', so distance is 1 for 'H'
Closest leaf to 'C' is 'B', so distance is 2 for 'C'
Closest leaf to 'E' is either 'I' or 'J', so distance is 2 for 'E'
Closest leaf to 'B' is 'B' itself, so distance is 0 for 'B'

The main point to note here is that a closest key can either be a descendent of given key or can be reached through one of the ancestors.

The **idea** is **to traverse the given tree in preorder and keep track of ancestors in an array**. When we reach the given key, we evaluate distance of the closest leaf in subtree rooted with given key. We also traverse all ancestors one by one and find distance of the closest leaf in the subtree rooted with ancestor. We compare all distances and return minimum.

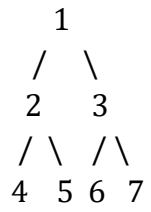
The above code can be **optimized** by storing the left/right information also in ancestor array. The idea is, if given key is in left subtree of an ancestors, then there is no point to call closestDown(). Also, the loop can that traverses ancestors array can be optimized to not traverse ancestors which are at more distance than current result.

Exercise:
Extend the above solution to print not only distance, but the key of closest leaf also.

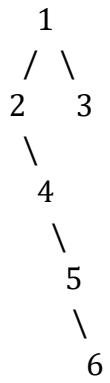
82 Print Nodes in Top View of Binary Tree

http://www.geeksforgeeks.org/print-nodes-top-view-binary-tree/
Top view of a binary tree is the set of nodes visible when the tree is viewed from the top. Given a binary tree, print the top view of it. The output nodes can be printed in any order. **Expected time complexity** is O(n)

A node x is there in output if x is the topmost node at its horizontal distance. Horizontal distance of left child of a node x is equal to horizontal distance of x minus 1, and that of right child is horizontal distance of x plus 1.



Top view of the above binary tree is
4 2 1 3 7



Top view of the above binary tree is
2 1 3 6

The **idea** is to do **something similar to vertical Order Traversal**. Like vertical Order Traversal, we need to nodes of same horizontal distance together. We do a level order traversal so that the topmost node at a horizontal node is visited before any other node of same horizontal distance below it. Hashing is used to check if a node at given horizontal distance is seen or not.

```
public void printTopView()
    {
        // base case
        if (root == null) {  return;  }

        // Creates an empty hashset
        HashSet<Integer> set = new HashSet<>();

        // Create a queue and add root to it
        Queue<QItem> Q = new LinkedList<QItem>();
        Q.add(new QItem(root, 0)); // Horizontal distance of root is 0

        // Standard BFS or level order traversal loop
        while (!Q.isEmpty())
        {
            // Remove the front item and get its details
            QItem qi = Q.remove();
            int hd = qi.hd;
            TreeNode n = qi.node;

            // If this is the first node at its horizontal distance,
            // then this node is in top view
            if (!set.contains(hd))
            {
                set.add(hd);
                System.out.print(n.key + " ");
            }
```

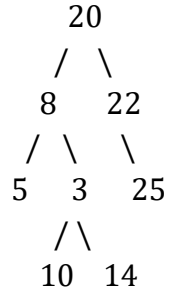
```
            // Enqueue left and right children of current node
            if (n.left != null)
                Q.add(new QItem(n.left, hd-1));
            if (n.right != null)
                Q.add(new QItem(n.right, hd+1));
        }
    }
}
```

Time Complexity of the above implementation is O(n) where n is number of nodes in given binary tree. The assumption here is that add() and contains() methods of HashSet work in O(1) time.

83 Bottom View of a Binary Tree

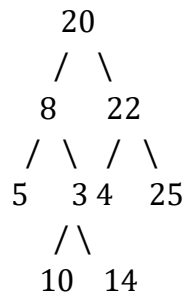
http://www.geeksforgeeks.org/bottom-view-binary-tree/

Given a Binary Tree, we need to print the bottom view from left to right. A node x is there in output if x is the bottommost node at its horizontal distance. Horizontal distance of left child of a node x is equal to horizontal distance of x minus 1, and that of right child is horizontal distance of x plus 1.



For the above tree the output should be 5, 10, 3, 14, 25.

If there are multiple bottom-most nodes for a horizontal distance from root, then print the later one in level traversal. For example, in the below diagram, 3 and 4 are both the bottom-most nodes at horizontal distance 0, we need to print 4.



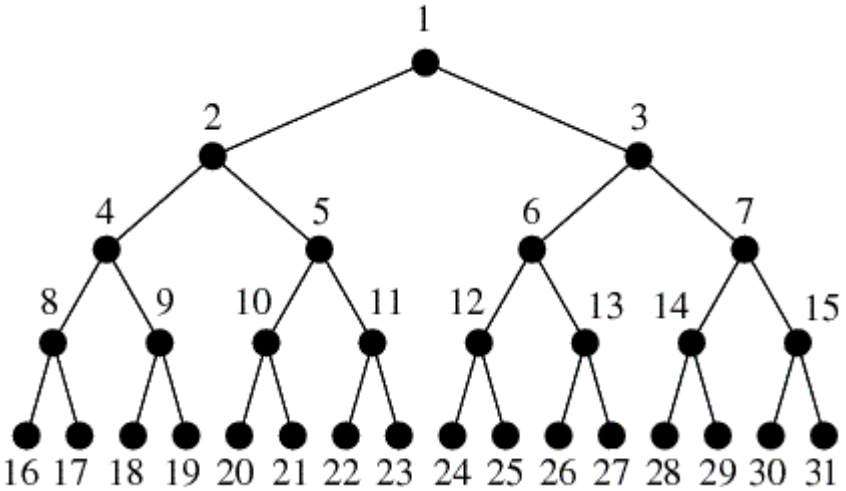
For the above tree the output should be 5, 10, 4, 14, 25.

The following are steps to print Bottom View of Bianry Tree.
1. We put tree nodes in a queue for the level order traversal.
2. Start with the horizontal distance(hd) 0 of the root node, keep on adding left child to queue along with the horizontal distance as hd-1 and right child as hd+1.
3. Also, use a TreeMap which stores key value pair sorted on key.
4. Every time, we encounter a new horizontal distance or an existing horizontal distance put the node data for the horizontal distance as key. For the first time it will add to the map, next time it will replace the value. This will make sure that the bottom most element for that horizontal distance is present in the map and if you see the tree from beneath that you will see that element.

Exercise: Extend the above solution to print all bottommost nodes at a horizontal distance if there are multiple bottommost nodes. For the above second example, the output should be 5 10 3 4 14 25.

84 Perfect Binary Tree Specific Level Order Traversal

http://www.geeksforgeeks.org/perfect-binary-tree-specific-level-order-traversal/
Given a Perfect Binary Tree like below:
(click on image to get a clear view)



Print the level order of nodes in following specific manner:
1 2 3 4 7 5 6 8 15 9 14 10 13 11 12 16 31 17 30 18 29 19 28 20 27 21 26 22 25 23 24
i.e. print nodes in level order but nodes should be from left and right side alternatively.

Here 1st and 2nd levels are trivial.
While 3rd level: 4(left), 7(right), 5(left), 6(right) are printed.
While 4th level: 8(left), 15(right), 9(left), 14(right), .. are printed.
While 5th level: 16(left), 31(right), 17(left), 30(right), .. are printed.

In **standard Level Order Traversal**, we enqueue root into a queue 1st, then we dequeue ONE node from queue, process (print) it, enqueue its children into queue. We keep doing this until queue is empty.

Approach 1:

We can do standard level order traversal here too but instead of printing nodes directly, we have to store nodes in current level in a temporary array or list 1st and then take nodes from alternate ends (left and right) and print nodes. Keep repeating this for all levels.
This approach takes more memory than standard traversal.

Approach 2:

The standard level order traversal idea will slightly change here. Instead of processing ONE node at a time, we will process TWO nodes at a time. And while pushing children into queue, the enqueue order will be: 1st node's left child, 2nd node's right child, 1st node's right child and 2nd node's left child.

```
/* Given a perfect binary tree, print its nodes in specific
   level order */
void printSpecificLevelOrder(Node *root)
{
    if (root == NULL)
        return;

    // Let us print root and next level first
    cout << root->data;

    // / Since it is perfect Binary Tree, right is not checked
    if (root->left != NULL)
      cout << " " << root->left->data << " " << root->right->data;

    // Do anything more if there are nodes at next level in
    // given perfect Binary Tree
    if (root->left->left == NULL)
        return;

    // Create a queue and enqueue left and right children of root
    queue <Node *> q;
    q.push(root->left);
    q.push(root->right);
```

```
    // We process two nodes at a time, so we need two variables
    // to store two front items of queue
    Node *first = NULL, *second = NULL;

    // traversal loop
    while (!q.empty())
    {
        // Pop two items from queue
        first = q.front();
        q.pop();
        second = q.front();
        q.pop();

        // Print children of first and second in reverse order
        cout << " " << first->left->data << " " << second->right->data;
        cout << " " << first->right->data << " " << second->left->data;

        // If first and second have grandchildren, enqueue them
        // in reverse order
        if (first->left->left != NULL)
        {
            q.push(first->left);
            q.push(second->right);
            q.push(first->right);
            q.push(second->left);
        }
    }
}
```

Followup Questions:

The above code prints specific level order from TOP to BOTTOM. How will you do specific level order traversal from BOTTOM to TOP (Amazon Interview | Set 120 – Round 1 Last Problem)

What if tree is not perfect, but complete.

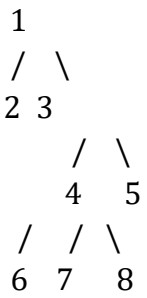
What if tree is neither perfect, nor complete. It can be any general binary tree.

85 Convert left-right representation of a binary tree to down-right

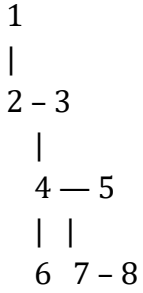
http://quiz.geeksforgeeks.org/convert-left-right-representation-bianry-tree-right/
Left-Right representation of a binary tree is standard representation where every node has a pointer to left child and another pointer to right child.

Down-Right representation is an alternate representation where every node has a pointer to left (or first) child and another pointer to next sibling. So siblings at every level are connected from left to right.

Given a binary tree in left-right representation as below



Convert the structure of the tree to down-right representation like the below tree.



The conversion should happen in-place, i.e., left child pointer should be used as down pointer and right child pointer should be used as right sibling pointer.

The **idea** is to **first convert left and right children**, then **convert the root**. Following is C++ implementation of the idea.

```
// An Iterative level order traversal based function to
// convert left-right to down-right representation.
void convert(node *root)
{
    // Base Case
    if (root == NULL)  return;

    // Recursively convert left an right subtrees
    convert(root->left);
    convert(root->right);

    // If left child is NULL, make right child as left
    // as it is the first child.
    if (root->left == NULL)
       root->left = root->right;

    // If left child is NOT NULL, then make right child
    // as right of left child
    else
       root->left->right = root->right;

    // Set root's right as NULL
    root->right = NULL;
}
```

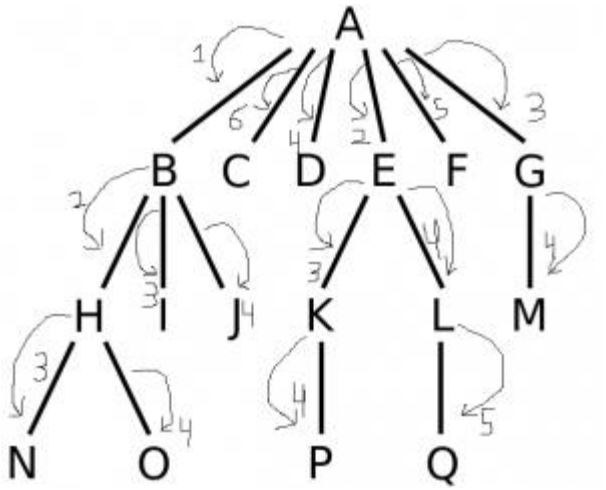
Time complexity of the above program is O(n).

86 Minimum no. of iterations to pass information to all nodes in the tree

http://www.geeksforgeeks.org/minimum-iterations-pass-information-nodes-tree/

Given a very large n-ary tree. Where the root node has some information which it wants to pass to all of its children down to the leaves with the constraint that it can only pass the information to one of its children at a time (take it as one iteration). Now in the next iteration the child node can transfer that information to only one of its children and at the same time instance the child's parent i.e. root can pass the info to one of its remaining children. Continuing in this way we have to find the minimum no of iterations required to pass the information to all nodes in the tree.

Minimum no of iterations for tree below is 6. The root A first passes information to B. In next iteration, A passes information to E and B passes information to H and so on.



This can be done using Post Order Traversal. The idea is to consider height and children count on each and every node.

If a child node i takes ci iterations to pass info below its subtree, then its parent will take (ci + 1) iterations to pass info to subtree rooted at that child i.

If parent has more children, it will pass info to them in subsequent iterations. Let's say children of a parent takes c1, c2, c3, c4, …, cn iterations to pass info in their own subtree, Now parent has to pass info to these n children one by one in n iterations. If parent picks child i in ith iteration, then parent will take (i + ci) iterations to pass info to child i and all it's subtree.

In any iteration, when parent passes info a child i+1, children (1 to i) which got info from parent already in previous iterations, will pass info to further down in subsequent iterations, if any child (1 to i) has its own child further down.

To pass info to whole tree in minimum iterations, it needs to be made sure that bandwidth is utilized as efficiently as possible (i.e. maximum passable no of nodes should pass info further down in any iteration)
The best possible scenario would be that in nth iteration, n different nodes pass info to their child.

Nodes with height = 0: (Trivial case) Leaf node has no children (no information passing needed), so no of iterations would be ZERO.

Nodes with height = 1: Here node has to pass info to all the children one by one (all children are leaf node, so no more information passing further down). Since all children are leaf, node can pass info to any child in any order (pick any child who didn't receive the info yet). One iteration needed for each child and so no of iterations would be no of children.So node with height 1 with n children will take n iterations.
Take a counter initialized with ZERO, loop through all children and keep incrementing counter.

Nodes with height > 1: Let's assume that there are n children (1 to n) of a node and minimum no iterations for all n children are c1, c2, …., cn.
To make sure maximum no of nodes participate in info passing in any iteration, parent should 1st pass info to that child who will take maximum iteration to pass info further down in subsequent iterations. i.e. in any iteration, parent should choose the child who takes maximum iteration later on. It can be thought of as a greedy approach where parent choose that child 1st, who needs maximum no of iterations so that all subsequent iterations can be utilized efficiently.

If parent goes in any other fashion, then in the end, there could be some nodes which are done quite early, sitting idle and so bandwidth is not utilized efficiently in further iterations.

If there are two children i and j with minimum iterations ci and cj where ci > cj, then If parent picks child j 1st then no of iterations needed by parent to pass info to both children and their subtree would be:max (1 + cj, 2 + ci) = 2 + ci

If parent picks child i 1st then no of iterations needed by parent to pass info to both children and their subtree would be: max(1 + ci, 2 + cj) = 1 + ci (So picking ci gives better result than picking cj)
This tells that parent should always choose child i with max ci value in any iteration.

SO here greedy approach is:
sort all ci values decreasing order,
let's say after sorting, values are c1 > c2 > c3 > …. > cn
take a counter c, set c = 1 + c1 (for child with maximum no of iterations)
for all children i from 2 to n, c = c + 1 + ci
then total no of iterations needed by parent is max(n, c)

Let minItr(A) be the minimum iteration needed to pass info from node A to it's all the sub-tree. Let child(A) be the count of all children for node A. So recursive relation would be:

```
1. Get minItr(B) of all children (B) of a node (A)
2. Sort all minItr(B) in descending order
3. Get minItr of A based on all minItr(B)
   minItr(A) = child(A)
   For children B from i = 0 to child(A)
         minItr(A) = max ( minItr(A), minItr(B) + i + 1)
```

Base cases would be:
If node is leaf, minItr = 0
If node's height is 1, minItr = children count

87 Clone a Binary Tree with Random Pointers

http://www.geeksforgeeks.org/clone-binary-tree-random-pointers/
Given a Binary Tree where every node has following structure.

```
struct node {
   int key;
   struct node *left,*right,*random;
}
```

The random pointer points to any random node of the binary tree and can even point to NULL, clone the given binary tree.

Method 1 (Use Hashing)
The idea is to store mapping from given tree nodes to clone tre node in hashtable. Following are detailed steps.

1) Recursively traverse the given Binary and copy key value, left pointer and right pointer to clone tree. While copying, store the mapping from given tree node to clone tree node in a hashtable. In the following pseudo code, 'cloneNode' is currently visited node of clone tree and 'treeNode' is currently visited node of given tree.

```
cloneNode->key  = treeNode->key
cloneNode->left = treeNode->left
cloneNode->right = treeNode->right
map[treeNode] = cloneNode
```

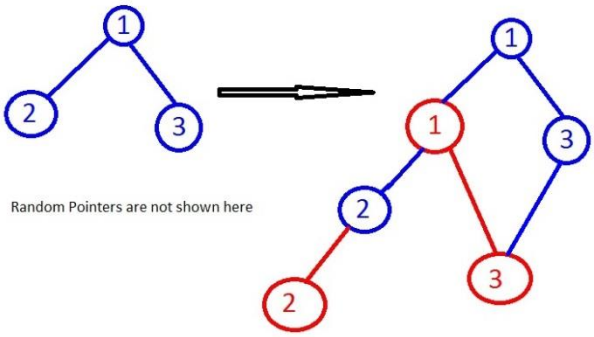
2) Recursively traverse both trees and set random pointers using entries from hash table.
cloneNode->random = map[treeNode->random]

Following is C++ implementation of above idea. The following implementation uses map from C++ STL. Note that map doesn't implement hash table, it actually is based on self-balancing binary search tree.

```
// This function creates clone by copying key and left and right pointers
// This function also stores mapping from given tree node to clone.
Node* copyLeftRightNode(Node* treeNode, map<Node *, Node *> *mymap)
{
    if (treeNode == NULL)
        return NULL;
    Node* cloneNode = newNode(treeNode->key);
    (*mymap)[treeNode] = cloneNode;
    cloneNode->left  = copyLeftRightNode(treeNode->left, mymap);
    cloneNode->right = copyLeftRightNode(treeNode->right, mymap);
    return cloneNode;
}

// This function copies random node by using the hashmap built by
// copyLeftRightNode()
void copyRandom(Node* treeNode,  Node* cloneNode, map<Node *, Node *>
*mymap)
{
    if (cloneNode == NULL)
        return;
    cloneNode->random =  (*mymap)[treeNode->random];
    copyRandom(treeNode->left, cloneNode->left, mymap);
    copyRandom(treeNode->right, cloneNode->right, mymap);
}

// This function makes the clone of given tree. It mainly uses
// copyLeftRightNode() and copyRandom()
Node* cloneTree(Node* tree)
{
    if (tree == NULL)
        return NULL;
    map<Node *, Node *> *mymap = new  map<Node *, Node *>;
    Node* newTree = copyLeftRightNode(tree, mymap);
    copyRandom(tree, newTree, mymap);
    return newTree;
}
```

Method 2 (Temporarily Modify the Given Binary Tree)

1. Create new nodes in cloned tree and insert each new node in original tree between the left pointer edge of corresponding node in the original tree (See the below image). i.e. if current node is A and it's left child is B (A — >> B), then new cloned node with key A wil be created (say cA) and it will be put as A — >> cA — >> B (B can be a NULL or a non-NULL left child). Right child pointer will be set correctly i.e. if for current node A, right child is C in original tree (A — >> C) then corresponding cloned nodes cA and cC will like cA —- >> cC



2. Set random pointer in cloned tree as per original tree
i.e. if node A's random pointer points to node B, then in cloned tree, cA will point to cB (cA and cB are new node in cloned tree corresponding to node A and B in original tree)

3. Restore left pointers correctly in both original and cloned tree

```
// This function creates new nodes cloned tree and puts new cloned node
// in between current node and it's left child
// i.e. if current node is A and it's left child is B ( A --- >> B ),
//      then new cloned node with key A wil be created (say cA) and
//      it will be put as
//      A --- >> cA --- >> B
// Here B can be a NULL or a non-NULL left child
// Right child pointer will be set correctly
// i.e. if for current node A, right child is C in original tree
// (A --- >> C) then corresponding cloned nodes cA and cC will like
// cA ---- >> cC
```

```
// This function creates new nodes cloned tree and puts new cloned node
// in between current node and it's left child
// i.e. if current node is A and it's left child is B ( A --- >> B ),
//      then new cloned node with key A wil be created (say cA) and
//      it will be put as
//      A --- >> cA --- >> B
// Here B can be a NULL or a non-NULL left child
// Right child pointer will be set correctly
// i.e. if for current node A, right child is C in original tree
// (A --- >> C) then corresponding cloned nodes cA and cC will like
// cA ---- >> cC
Node* copyLeftRightNode(Node* treeNode)
{
    if (treeNode == NULL)
        return NULL;

    Node* left = treeNode->left;
    treeNode->left = newNode(treeNode->key);
    treeNode->left->left = left;
    if(left != NULL)
        left->left = copyLeftRightNode(left);

    treeNode->left->right = copyLeftRightNode(treeNode->right);
    return treeNode->left;
}

// This function sets random pointer in cloned tree as per original tree
// i.e. if node A's random pointer points to node B, then
// in cloned tree, cA wil point to cB (cA and cB are new node in cloned
// tree corresponding to node A and B in original tree)
void copyRandomNode(Node* treeNode, Node* cloneNode)
{
    if (treeNode == NULL)
        return;
    if(treeNode->random != NULL)
        cloneNode->random = treeNode->random->left;
    else
        cloneNode->random = NULL;

    if(treeNode->left != NULL && cloneNode->left != NULL)
        copyRandomNode(treeNode->left->left, cloneNode->left->left);
    copyRandomNode(treeNode->right, cloneNode->right);
}

// This function will restore left pointers correctly in
// both original and cloned tree
void restoreTreeLeftNode(Node* treeNode, Node* cloneNode)
{
    if (treeNode == NULL)
        return;
    if (cloneNode->left != NULL)
    {
        Node* cloneLeft = cloneNode->left->left;
        treeNode->left = treeNode->left->left;
```

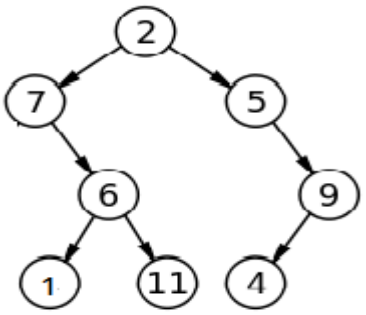
```
        cloneNode->left = cloneLeft;
    }
    else
        treeNode->left = NULL;

    restoreTreeLeftNode(treeNode->left, cloneNode->left);
    restoreTreeLeftNode(treeNode->right, cloneNode->right);
}

//This function makes the clone of given tree
Node* cloneTree(Node* treeNode)
{
    if (treeNode == NULL)
        return NULL;
    Node* cloneNode = copyLeftRightNode(treeNode);
    copyRandomNode(treeNode, cloneNode);
    restoreTreeLeftNode(treeNode, cloneNode);
    return cloneNode;

}
```

88 Given a binary tree, how do you remove all the half nodes?

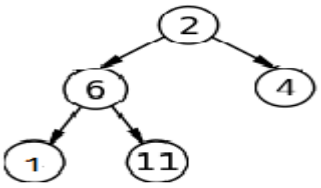
http://www.geeksforgeeks.org/given-a-binary-tree-how-do-you-remove-all-the-half-nodes/

Given A binary Tree, how do you remove all the half nodes (which has only one child)? Note leaves should not be touched as they have both children as NULL.

For example consider the below tree.



Nodes 7, 5 and 9 are half nodes as one of their child is Null. We need to remove all such half nodes and return the root pointer of following new tree.



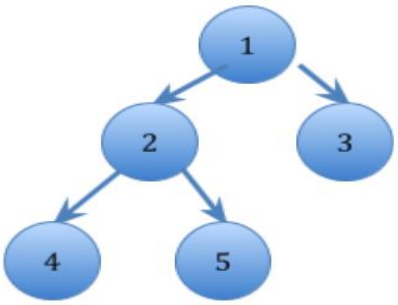
The **idea** is to **use post-order traversal** to solve this problem efficiently. We first process the left children, then right children, and finally the node itself. So we form the new tree bottom up, starting from the leaves towards the root. By the time we process the current node, both its left and right subtrees were already processed. Below is the implementation of this idea.

```
// Removes all nodes with only one child and returns
// new root (note that root may change)
struct node* RemoveHalfNodes(struct node* root)
{
    if (root==NULL)
        return NULL;

    root->left  = RemoveHalfNodes(root->left);
    root->right = RemoveHalfNodes(root->right);

    if (root->left==NULL && root->right==NULL)
        return root;

    /* if current nodes is a half node with left
       child NULL left, then it's right child is
       returned and replaces it in the given tree */
    if (root->left==NULL)
    {
        struct node *new_root = root->right;
        free(root); // To avoid memory leak
        return new_root;
    }


    /* if current nodes is a half node with right
       child NULL right, then it's right child is
       returned and replaces it in the given tree  */
    if (root->right==NULL)
    {
        struct node *new_root = root->left;
        free(root); // To avoid memory leak
        return new_root;
    }

    return root;
}
```

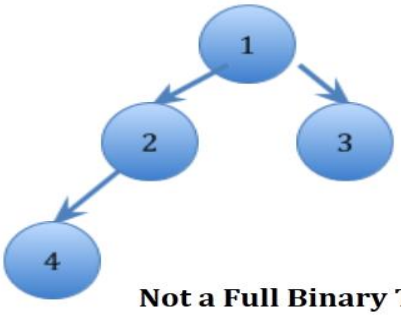
Time complexity of the above solution is O(n) as it does a simple traversal of binary tree.

89 Check whether a binary tree is a full binary tree or not

http://www.geeksforgeeks.org/check-whether-binary-tree-full-binary-tree-not/
A full binary tree is defined as a binary tree in which all nodes have either zero or two child nodes. Conversely, there is no node in a full binary tree, which has one child node. More information about full binary trees can be found here.
For Example:



Full Binary Tree



Not a Full Binary Tree

To check whether a binary tree is a full binary tree we need to test the following cases:-

1) If a binary tree node is NULL then it is a full binary tree.
2) If a binary tree node does have empty left and right sub-trees, then it is a full binary tree by definition
3) If a binary tree node has left and right sub-trees, then it is a part of a full binary tree by definition. In this case recursively check if the left and right sub-trees are also binary trees themselves.
4) In all other combinations of right and left sub-trees, the binary tree is not a full binary tree.

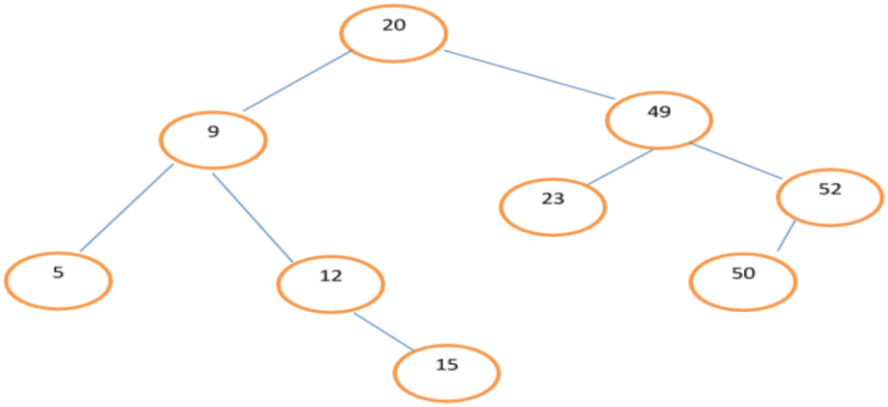
```
/* This function tests if a binary tree is a full binary tree. */
bool isFullTree (struct Node* root)
{
    // If empty tree
    if (root == NULL)
        return true;

    // If leaf node
    if (root->left == NULL && root->right == NULL)
        return true;

    // If both left and right are not NULL, and left & right subtrees
    // are full
    if ((root->left) && (root->right))
        return (isFullTree(root->left) && isFullTree(root->right));

    // We reach here when none of the above if conditions work
    return false;
}
```

Time complexity of the above code is O(n) where n is number of nodes in given binary tree.

90 Find sum of all left leaves in a given Binary Tree

http://www.geeksforgeeks.org/find-sum-left-leaves-given-binary-tree/

Given a Binary Tree, find sum of all left leaves in it. For example, sum of all left leaves in below Binary Tree is 5+23+50 = 78.



The **idea** is to traverse the tree, starting from root. For every node, check if its left subtree is a leaf. If it is, then add it to the result.

```
This function returns sum of all left leaves in a given binary tree
int leftLeavesSum(Node *root)
{
    // Initialize result
    int res = 0;
     // Update result if root is not NULL
    if (root != NULL)
    {
       // If left of root is NULL, then add key of
       // left child
       if (isLeaf(root->left))
            res += root->left->key;
       else // Else recur for left child of root
            res += leftLeavesSum(root->left);
        // Recur for right child of root and update res
       res += leftLeavesSum(root->right);
    }
     // return result
    return res;
}
```

Time complexity of the above solution is O(n) where n is number of nodes in Binary Tree.

Following is **Another Method** to solve the above problem. This **solution passes in a sum variable as an accumulator**. When a left leaf is encountered, the leaf's data is added to sum.
Time complexity of this method is also O(n).

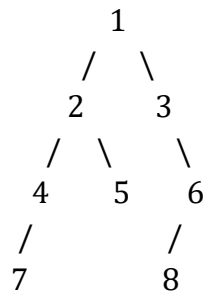
```
/* Pass in a sum variable as an accumulator */
void leftLeavesSumRec(Node *root, bool isleft, int *sum)
{
    if (!root) return;

    // Check whether this node is a leaf node and is left.
    if (!root->left && !root->right && isleft)
        *sum += root->key;
    // Pass 1 for left and 0 for right
    leftLeavesSumRec(root->left,  1, sum);
    leftLeavesSumRec(root->right, 0, sum);
}
// A wrapper over above recursive function
int leftLeavesSum(Node *root)
{
    int sum = 0; //Initialize result

    // use the above recursive function to evaluate sum
    leftLeavesSumRec(root, 0, &sum);

    return sum;
}
```

91 Remove nodes on root to leaf paths of length < K

http://www.geeksforgeeks.org/remove-nodes-root-leaf-paths-length-k/

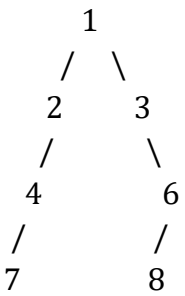
Given a Binary Tree and a number k, remove all nodes that lie only on root to leaf path(s) of length smaller than k. If a node X lies on multiple root-to-leaf paths and if any of the paths has path length >= k, then X is not deleted from Binary Tree. In other words a node is deleted if all paths going through it have lengths smaller than k.

Consider the following example Binary Tree



Input: Root of above Binary Tree
k = 4

Output: The tree should be changed to following



There are 3 paths
i) 1->2->4->7 path length = 4
ii) 1->2->5 path length = 3
iii) 1->3->6->8 path length = 4
There is only one path " 1->2->5 " of length smaller than 4.
The node 5 is the only node that lies only on this path, so
node 5 is removed.
Nodes 2 and 1 are not removed as they are parts of other paths
of length 4 as well.

If k is 5 or greater than 5, then whole tree is deleted.
If k is 3 or less than 3, then nothing is deleted.

The **idea** here is to use post order traversal of the tree. Before removing a node we need to check that all the children of that node in the shorter path are already removed.

There are 2 cases:
i) This node becomes a leaf node in which case it needs to be deleted.
ii) This node has other child on a path with path length >= k. In that case it needs not to be deleted.

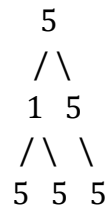
```
// Utility method that actually removes the nodes which are not
// on the pathLen >= k. This method can change the root as well.
Node *removeShortPathNodesUtil(Node *root, int level, int k)
{
    //Base condition
    if (root == NULL)
        return NULL;

    // Traverse the tree in postorder fashion so that if a leaf
    // node path length is shorter than k, then that node and
    // all of its descendants till the node which are not
    // on some other path are removed.
    root->left = removeShortPathNodesUtil(root->left, level + 1, k);
    root->right = removeShortPathNodesUtil(root->right, level + 1, k);

    // If root is a leaf node and it's level is less than k then
    // remove this node.
    // This goes up and check for the ancestor nodes also for the
    // same condition till it finds a node which is a part of other
    // path(s) too.
    if (root->left == NULL && root->right == NULL && level < k)
    {
        delete root;
        return NULL;
    }

    // Return root;
    return root;
}

// Method which calls the utitlity method to remove the short path
// nodes.
Node *removeShortPathNodes(Node *root, int k)
{
    int pathLen = 0;
    return removeShortPathNodesUtil(root, 1, k);
}
```

Time complexity of the above solution is O(n) where n is number of nodes in given Binary Tree.

92 Find Count of Single Valued Subtrees

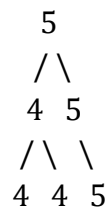
http://www.geeksforgeeks.org/find-count-of-singly-subtrees/
Given a binary tree, write a program to count the number of Single Valued Subtrees. A Single Valued Subtree is one in which all the nodes have same value.
Expected time complexity is O(n).

Example:
Input: root of below tree



Output: 4

There are 4 subtrees with single values.
Input: root of below tree



Output: 5
There are five subtrees with single values.

A **Simple Solution** is to traverse the tree. For every traversed node, check if all values under this node are same or not. If same, then increment count.
Time complexity of this solution is O(n2).

An **Efficient Solution** is to **traverse the tree in bottom up manner**. For every subtree visited, return true if subtree rooted under it is single valued and increment count. So the idea is to use count as a reference parameter in recursive calls and use returned values to find out if left and right subtrees are single valued or not.

```
// This function increments count by number of single
// valued subtrees under root. It returns true if subtree
// under root is Singly, else false.
bool countSingleRec(Node* root, int &count)
{
    // Return false to indicate NULL
    if (root == NULL)
       return true;

    // Recursively count in left and right subtrees also
    bool left = countSingleRec(root->left, count);
    bool right = countSingleRec(root->right, count);

    // If any of the subtrees is not singly, then this
    // cannot be singly.
    if (left == false || right == false)
       return false;

    // If left subtree is singly and non-empty, but data
    // doesn't match
    if (root->left && root->data != root->left->data)
        return false;

    // Same for right subtree
    if (root->right && root->data != root->right->data)
        return false;

    // If none of the above conditions is true, then
    // tree rooted under root is single valued, increment
    // count and return true.
    count++;
    return true;
}
```

Time complexity of this solution is O(n) where n is number of nodes in given binary tree.

93 Check if a given array can represent Preorder Traversal of Binary Search Tree

http://www.geeksforgeeks.org/check-if-a-given-array-can-represent-preorder-traversal-of-binary-search-tree/

Given an array of numbers, return true if given array can represent preorder traversal of a Binary Search Tree, else return false. Expected time complexity is O(n).

Examples:

Input: pre[] = {2, 4, 3}

Output: true

Given array can represent preorder traversal
of below tree

```
  2
   \
    4
   /
  3
```

Input: pre[] = {2, 4, 1}

Output: false

Given array cannot represent preorder traversal
of a Binary Search Tree.

Input: pre[] = {40, 30, 35, 80, 100}

Output: true

Given array can represent preorder traversal
of below tree

```
    40
  /   \
 30    80
  \     \
  35     100
```

Input: pre[] = {40, 30, 35, 20, 80, 100}

Output: false

Given array cannot represent preorder traversal
of a Binary Search Tree.

A **Simple Solution** is to do following for every node pre[i] starting from first one.

1) Find the first greater value on right side of current node.
Let the index of this node be j. Return true if following
conditions hold. Else return false
(i) All values after the above found greater value are
greater than current node.
(ii) Recursive calls for the subarrays pre[i+1..j-1] and
pre[j+1..n-1] also return true.

Time Complexity of the above solution is O(n2)

An **Efficient Solution** can solve this problem in **O(n) time**. The idea is to use a stack. This problem is similar to Next (or closest) Greater Element problem. Here we find next greater element and after finding next greater, if we find a smaller element, then return false.

1) Create an empty stack.
2) Initialize root as INT_MIN.
3) Do following for every element pre[i]
a) If pre[i] is smaller than current root, return false.
b) Keep removing elements from stack while pre[i] is greater
then stack top. Make the last removed item as new root (to
be compared next).
At this point, pre[i] is greater than the removed root
(That is why if we see a smaller element in step a), we
return false)
c) push pre[i] to stack (All elements in stack are in decreasing
order)

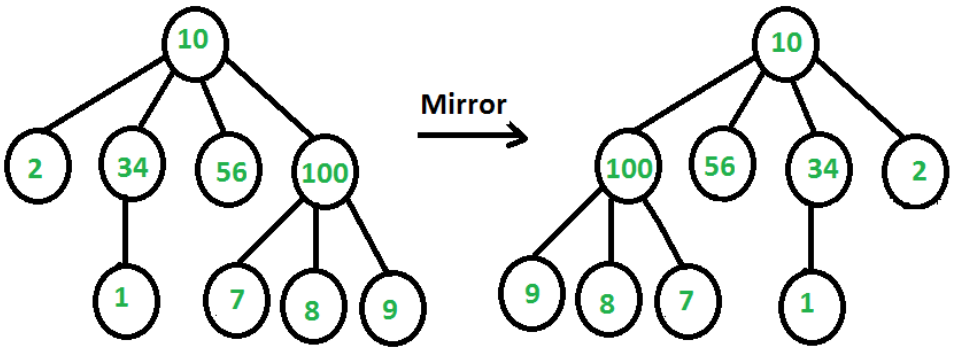
```
bool canRepresentBST(int pre[], int n)
{
    // Create an empty stack
    stack<int> s;

    // Initialize current root as minimum possible
    // value
    int root = INT_MIN;

    // Traverse given array
    for (int i=0; i<n; i++)
    {
        // If we find a node who is on right side
        // and smaller than root, return false
```

```
        if (pre[i] < root)
            return false;

        // If pre[i] is in right subtree of stack top,
        // Keep removing items smaller than pre[i]
        // and make the last removed item as new
        // root.
        while (!s.empty() && s.top()<pre[i])
        {
            root = s.top();
            s.pop();
        }

        // At this point either stack is empty or
        // pre[i] is smaller than root, push pre[i]
        s.push(pre[i]);
    }
    return true;
}
```

94 Mirror of n-ary Tree

http://www.geeksforgeeks.org/mirror-of-n-ary-tree/

Given a Tree where every node contains variable number of children, convert the tree to its mirror. Below diagram shows an example.



Node of tree is represented as a key and a variable sized array of children pointers. The **idea is similar to mirror of Binary Tree**. For every node, we first recur for all of its children and then reverse array of children pointers. We can also do these steps in other way, i.e., **reverse array of children pointers first and then recur for children**.

```
// Function to convert a tree to its mirror
void mirrorTree(Node * root)
{
    // Base case: Nothing to do if root is NULL
    if (root==NULL)
        return;

    // Number of children of root
    int n = root->child.size();

    // If number of child is less than 2 i.e.
    // 0 or 1 we do not need to do anything
    if (n < 2)
        return;

    // Calling mirror function for each child
    for (int i=0; i<n; i++)
        mirrorTree(root->child[i]);

    // Reverse vector (variable sized array) of child
    // pointers
    reverse(root->child.begin(), root->child.end());

}
```

95 Find multiplication of sums of data of leaves at same levels

http://www.geeksforgeeks.org/find-multiplication-of-sums-of-data-of-all-leaves-at-sane-levels/

Given a Binary Tree, return following value for it.

1) For every level, compute sum of all leaves if there are leaves at this level. Otherwise ignore it.

2) Return multiplication of all sums.

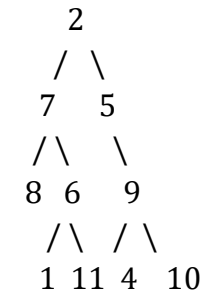
Input: Root of below tree

```
     2
   /  \
  7    5
        \
         9
```

Output: 63

First levels doesn't have leaves. Second level has one leaf 7 and third level also has one leaf 9. Therefore result is 7*9 = 63

Input: Root of below tree



Output: 208
First two levels don't have leaves. Third level has single leaf 8. Last level has four leaves 1, 11, 4 and 10. Therefore result is 8 * (1 + 11 + 4 + 10)

One **Simple Solution** is to recursively compute leaf sum for all level starting from top to bottom. Then multiply sums of levels which have leaves.
Time complexity of this solution would be O(n2).

An **Efficient Solution** is to **use Queue based level order traversal**. While doing the traversal, process all different levels separately. For every processed level, check if it has a leaves. If it has then compute sum of leaf nodes. Finally return product of all sums.

96 Succinct Encoding of Binary Tree

http://www.geeksforgeeks.org/succinct-encoding-of-binary-tree/
A succinct encoding of Binary Tree takes close to minimum possible space. The number of structurally different binary trees on n nodes is n'th Catalan number. For large n, this is about 4n; thus we need at least about log2 4 n= 2n bits to encode it. A succinct binary tree therefore would occupy 2n+o(n) bits.

One simple representation which meets this bound is to visit the nodes of the tree in preorder, outputting "1" for an internal node and "0" for a leaf. If the tree contains data, we can simply simultaneously store it in a consecutive array in preorder.

Below is algorithm for encoding:

```
function EncodeSuccinct(node n, bitstring structure, array data) {
   if n = nil then
      append 0 to structure;
   else
      append 1 to structure;
      append n.data to data;
      EncodeSuccinct(n.left, structure, data);
      EncodeSuccinct(n.right, structure, data);
}
```

And below is algorithm for decoding

```
function DecodeSuccinct(bitstring structure, array data) {
   remove first bit of structure and put it in b
   if b = 1 then
      create a new node n
      remove first element of data and put it in n.data
      n.left = DecodeSuccinct(structure, data)
      n.right = DecodeSuccinct(structure, data)
      return n
   else
      return nil
}
```

Example:
Input:


```
      10
    /    \
  20      30
 / \       \
40  50      70
```

Data Array (Contains preorder traversal)
10 20 40 50 30 70

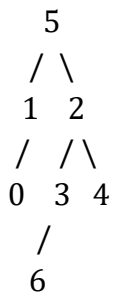
Structure Array
1 1 1 0 0 1 0 0 1 0 1 0 0
1 indicates data and 0 indicates NULL

97 Construct Binary Tree from given Parent Array representation

http://www.geeksforgeeks.org/construct-a-binary-tree-from-parent-array-representation/

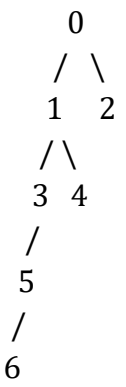
Given an array that represents a tree in such a way that array indexes are values in tree nodes and array values give the parent node of that particular index (or node). The value of the root node index would always be -1 as there is no parent for root. Construct the standard linked representation of given Binary Tree from this given representation.

Input: parent[] = {1, 5, 5, 2, 2, -1, 3}
Output: root of below tree



Explanation:
Index of -1 is 5. So 5 is root.
5 is present at indexes 1 and 2. So 1 and 2 are children of 5.
1 is present at index 0, so 0 is child of 1.
2 is present at indexes 3 and 4. So 3 and 4 are children of 2.
3 is present at index 6, so 6 is child of 3.

Input: parent[] = {-1, 0, 0, 1, 1, 3, 5};
Output: root of below tree



Expected **time complexity** is O(n) where n is number of elements in given array.

A **Simple Solution** to **recursively construct by first searching the current root**, then recurring for the found indexes (there can be at most two indexes) and making them left and right subtrees of root. This solution takes O(n2) as we have to linearly search for every node.

An **Efficient Solution** can solve the above problem in O(n) time. The idea is to **use extra space**. An array created[0..n-1] is used to keep track of created nodes.
createTree(parent[], n)
Create an array of pointers say created[0..n-1]. The value of created[i] is NULL if node for index i is not created, else value is pointer to the created node.

Do following for every index i of given array
createNode(parent, i, created)
createNode(parent[], i, crated[])

If created[i] is not NULL, then node is already created. So return.
Create a new node with value 'i'.

If parent[i] is -1 (i is root), make created node as root and return.
Check if parent of 'i' is created (We can check this by checking if created[parent[i]] is NULL or not.

If parent is not created, recur for parent and create the parent first.
Let the pointer to parent be p. If p->left is NULL, then make the new node as left child. Else make the new node as right child of parent.

```
// Creates a node with key as 'i'.  If i is root, then it changes
// root.  If parent of i is not created, then it creates parent first
void createNode(int parent[], int i, Node *created[], Node **root)
{
    // If this node is already created
    if (created[i] != NULL)
        return;

    // Create a new node and set created[i]
    created[i] = newNode(i);

    // If 'i' is root, change root pointer and return
    if (parent[i] == -1)
    {
        *root = created[i];
        return;
    }

    // If parent is not created, then create parent first
    if (created[parent[i]] == NULL)
        createNode(parent, parent[i], created, root);

    // Find parent pointer
    Node *p = created[parent[i]];

    // If this is first child of parent
    if (p->left == NULL)
        p->left = created[i];
    else // If second child
        p->right = created[i];
}

// Creates tree from parent[0..n-1] and returns root of the created tree
Node *createTree(int parent[], int n)
{
    // Create an array created[] to keep track
    // of created nodes, initialize all entries
    // as NULL
    Node *created[n];
    for (int i=0; i<n; i++)
        created[i] = NULL;

    Node *root = NULL;
    for (int i=0; i<n; i++)
        createNode(parent, i, created, &root);

    return root;

}
```

98 Symmetric Tree (Mirror Image of itself)

http://www.geeksforgeeks.org/symmetric-tree-tree-which-is-mirror-image-of-itself/

Given a binary tree, check whether it is a mirror of itself.

For example, this binary tree is symmetric:

```
    1
   / \
  2   2
 /\   /\
3  4 4  3
```

But the following is not:

```
    1
   /\
  2  2
   \  \
   3   3
```

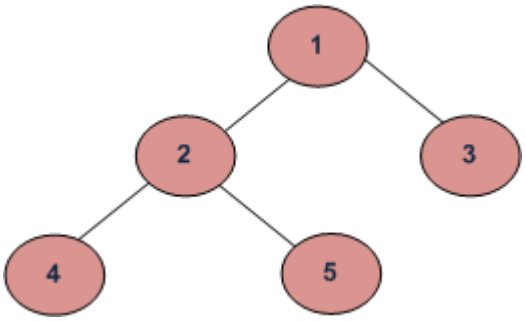
The **idea** is to write a **recursive function isMirror() that takes two trees as argument and returns true if trees are mirror and false if trees are not mirror**.

The isMirror() function recursively checks two roots and subtrees under the root.

```
bool isMirror(struct Node *root1, struct Node *root2)
{
    // If both trees are emptu, then they are mirror images
    if (root1 == NULL && root2 == NULL)
        return true;
     // For two trees to be mirror images, the following three
    // conditions must be true
    // 1 - Their root node's key must be same
    // 2 - left subtree of left tree and right subtree
    //      of right tree have to be mirror images
    // 3 - right subtree of left tree and left subtree
    //      of right tree have to be mirror images
    if (root1 && root2 && root1->key == root2->key)
        return isMirror(root1->left, root2->right) &&
               isMirror(root1->right, root2->left);
    // if neither of above conditions is true then root1
    // and root2 are not mirror images
    return false;
}
// Returns true if a tree is symmetric i.e. mirror image of itself
bool isSymmetric(struct Node* root)
{
    // Check if tre is mirror of itself
    return isMirror(root, root);
}
```

99 Find Minimum Depth of a Binary Tree

http://www.geeksforgeeks.org/find-minimum-depth-of-a-binary-tree/
Given a binary tree, find its minimum depth. The minimum depth is the number of nodes along the shortest path from root node down to the nearest leaf node.
For example, minimum height of below Binary Tree is 2.



Note that the path must end on a leaf node. For example, minimum height of below Binary Tree is also 2.

```
      10
     /
    5
```

The **idea** is to traverse the given Binary Tree. **For every node, check if it is a leaf node**. If yes, then return 1. If not leaf node then if left subtree is NULL, then recur for right subtree. And if right subtree is NULL, then recur for left subtree. If both left and right subtrees are not NULL, then take the minimum of two heights.
Time complexity of above solution is O(n) as it traverses the tree only once.

The above method may end up with complete traversal of Binary Tree even when the topmost leaf is close to root. A Better Solution is to do Level Order Traversal. While doing traversal, returns depth of the first encountered leaf node.

100 Maximum Path Sum in a Binary Tree

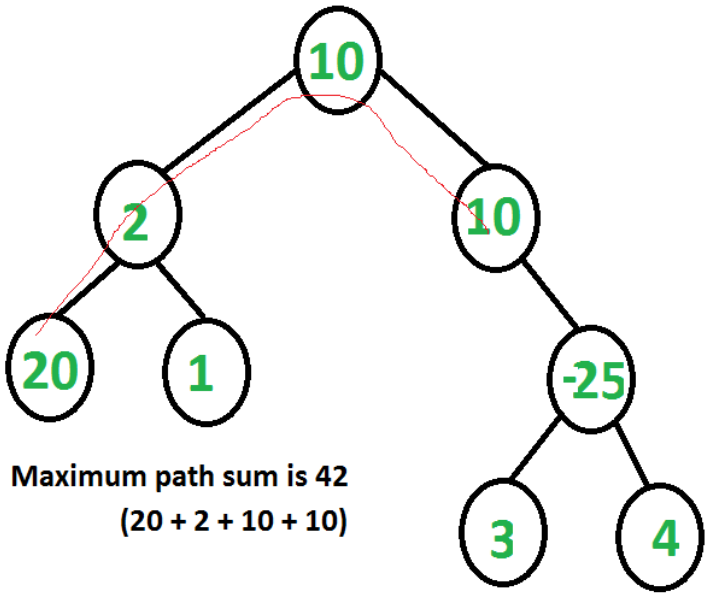
http://www.geeksforgeeks.org/find-maximum-path-sum-in-a-binary-tree/
Given a binary tree, find the maximum path sum. The path may start and end at any node in the tree.

Example:
Input: Root of below tree

```
    1
   / \
  2   3
```

Output: 6

See below diagram for another example.
1+2+3



For each node there can be four ways that the max path goes through the node:

1. Node only
2. Max path through Left Child + Node
3. Max path through Right Child + Node
4. Max path through Left Child + Node + Max path through Right Child

The **idea** is to keep trace of four paths and pick up the max one in the end. **An important thing to note is, root of every subtree need to return maximum path sum such that at most one child of root is involved**. This is needed for parent function call. In below code, this sum is stored in 'max_single' and returned by the recursive function.

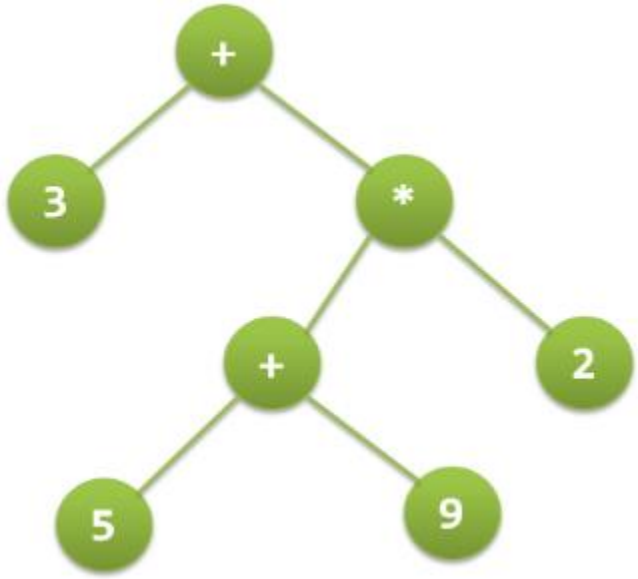
```
// This function returns overall maximum path sum in 'res'
// And returns max path sum going through root.
int findMaxUtil(Node* root, int &res)
{
    //Base Case
    if (root == NULL)
        return 0;

    // l and r store maximum path sum going through left and
    // right child of root respectively
    int l = findMaxUtil(root->left,res);
    int r = findMaxUtil(root->right,res);

    // Max path for parent call of root. This path must
    // include at-most one child of root
    int max_single = max(max(l, r) + root->data, root->data);

    // Max Top represents the sum when the Node under
    // consideration is the root of the maxsum path and no
    // ancestors of root are there in max sum path
    int max_top = max(max_single, l + r + root->data);

    res = max(res, max_top); // Store the Maximum Result.

    return max_single;
}

// Returns maximum path sum in tree with given root
int findMaxSum(Node *root)
{
    // Initialize result
    int res = INT_MIN;

    // Compute and return result
    findMaxUtil(root, res);
    return res;
}
```

Time Complexity: O(n) where n is number of nodes in Binary Tree.

101 Expression Tree

http://www.geeksforgeeks.org/expression-tree/

Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand so for example expression tree for 3 + ((5+9)*2) would be:



Inorder traversal of expression tree produces infix version of given postfix expression (same with preorder traversal it gives prefix expression)

Evaluating the expression represented by expression tree:

```
Let t be the expression tree
If  t is not null then
    If t.value is operand then
          Return  t.value
    A = solve(t.left)
    B = solve(t.right)

    // calculate applies operator 't.value'
    // on A and B, and returns value
    Return calculate(A, B, t.value)
```

Construction of Expression Tree:
Now For constructing expression tree we use a stack. We loop through input expression and do following for every character.
1) If character is operand push that into stack
2) If character is operator pop two values from stack make them its child and push current node again.
At the end only element of stack will be root of expression tree.

```
// Returns root of constructed tree for given
// postfix expression
et* constructTree(char postfix[])
{
    stack<et *> st;
    et *t, *t1, *t2;

    // Traverse through every character of
    // input expression
    for (int i=0; i<strlen(postfix); i++)
    {
        // If operand, simply push into stack
        if (!isOperator(postfix[i]))
        {
            t = newNode(postfix[i]);
            st.push(t);
        }
        else // operator
        {
            t = newNode(postfix[i]);
             // Pop two top nodes
            t1 = st.top(); // Store top
            st.pop();      // Remove top
            t2 = st.top();
            st.pop();

            //  make them children
            t->right = t1;
            t->left = t2;

            // Add this subexpression to stack
            st.push(t);
        }
    }
     //  only element will be root of expression
    // tree
    t = st.top();
    st.pop();

    return t;
}
```

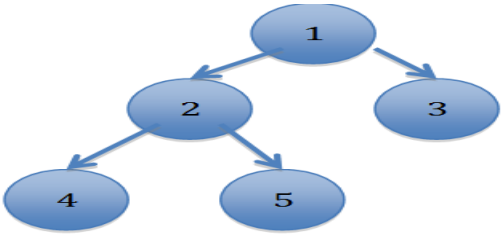
102 Check whether a binary tree is a complete tree or not | Set 2 (Recursive Solution)

http://www.geeksforgeeks.org/check-whether-binary-tree-complete-not-set-2-recursive-solution/

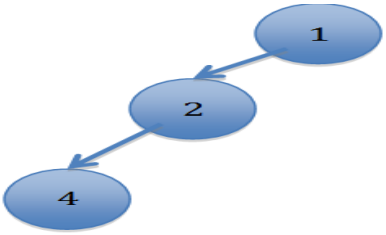
A complete binary tree is a binary tree whose all levels except the last level are completely filled and all the leaves in the last level are all to the left side. More information about complete binary trees can be found here.

For Example:-

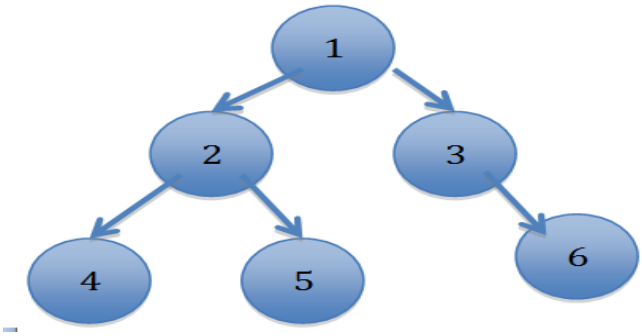
Below tree is a Complete Binary Tree (All nodes till the second last nodes are filled and all leaves are to the left side)



Below tree is not a Complete Binary Tree (The second level is not completely filled)

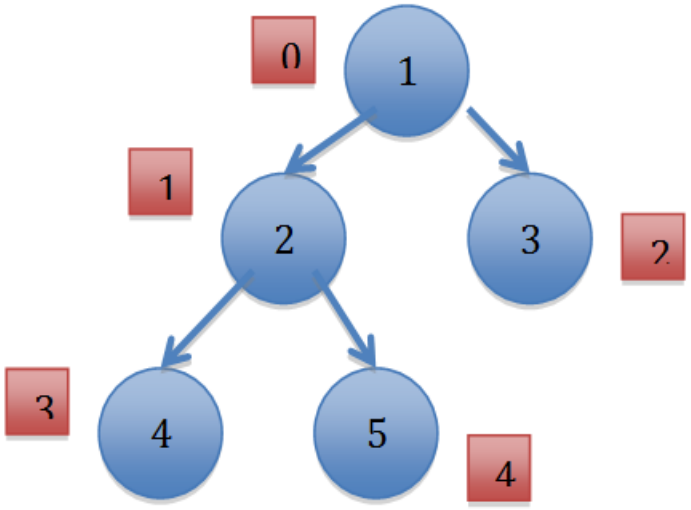


Below tree is not a Complete Binary Tree (All the leaves are not aligned to the left. The left child node of node with data 3 is empty while the right child node is non-empty).

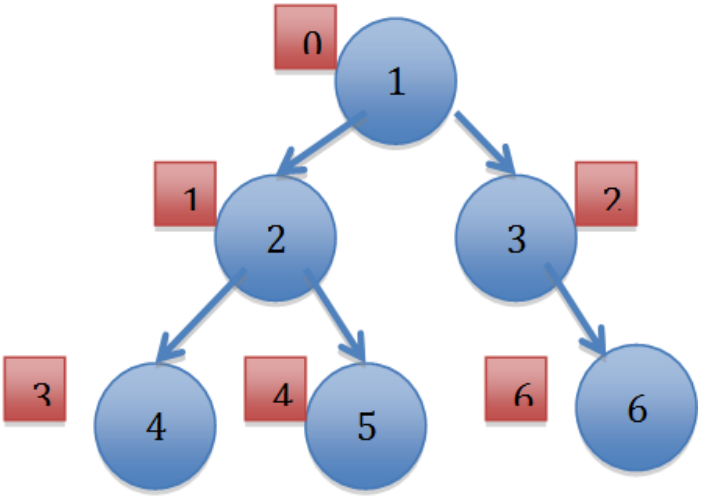


An **iterative solution** for this problem is discussed in below post.
Check whether a given Binary Tree is Complete or not | Set 1 (Using Level Order Traversal)

In the **array representation** of a binary tree, if the parent node is assigned an index of 'i' and left child gets assigned an index of '2*i + 1' while the right child is assigned an index of '2*i + 2'. If we represent the binary tree below as an array with the respective indices assigned to the different nodes of the tree below are shown below:-



As can be seen from the above figure, the assigned indices in case of a complete binary tree will strictly less be than the number of nodes in the complete binary tree. Below is the example of non-complete binary tree with the assigned array indices. As can be seen the assigned indices are equal to the number of nodes in the binary tree. Hence this tree is not a complete binary tree.



Hence we proceed in the following manner in order to check if the binary tree is complete binary tree.

Calculate the number of nodes (count) in the binary tree.
Start recursion of the binary tree from the root node of the binary tree with index (i) being set as 0 and the number of nodes in the binary (count).
If the current node under examination is NULL, then the tree is a complete binary tree. Return true.

If index (i) of the current node is greater than or equal to the number of nodes in the binary tree (count) i.e. (i>= count), then the tree is not a complete binary. Return false.
Recursively check the left and right sub-trees of the binary tree for same condition. For the left sub-tree use the index as (2*i + 1) while for the right sub-tree use the index as (2*i + 2).

The time complexity of the above algorithm is O(n). Following is the code for checking if a binary tree is a complete binary tree.

```
/* This function checks if the binary tree is complete or not */
bool isComplete (struct Node* root, unsigned int index,
                 unsigned int number_nodes)
{
    // An empty tree is complete
    if (root == NULL)
        return (true);

    // If index assigned to current node is more than
    // number of nodes in tree, then tree is not complete
    if (index >= number_nodes)
        return (false);

    // Recur for left and right subtrees
    return (isComplete(root->left, 2*index + 1, number_nodes) &&
            isComplete(root->right, 2*index + 2, number_nodes));

}
```

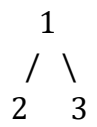
103 Change a Binary Tree so that every node stores sum of all nodes in left subtree

http://quiz.geeksforgeeks.org/change-a-binary-tree-so-that-every-node-stores-sum-of-all-nodes-in-left-subtree/

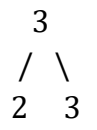
Given a Binary Tree, change the value in each node to sum of all the values in the nodes in the left subtree including its own.

Example

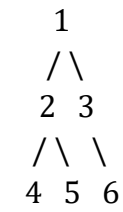
Input :



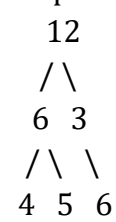
Output :



Input



Output:



The **idea** is to **traverse the given tree in bottom up manner**. For every node, recursively compute sum of nodes in left and right subtrees. Add sum of nodes in left subtree to current node and return sum of nodes under current subtree.

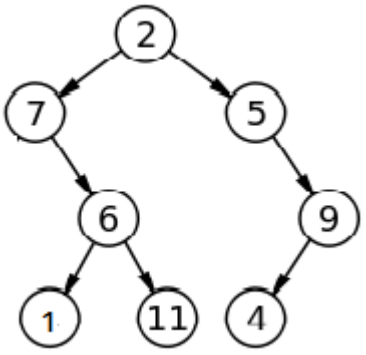
```
// Function to modify a Binary Tree so that every node
// stores sum of values in its left child including its
// own value
int updatetree(node *root)
{
    // Base cases
    if (!root)
        return 0;
    if (root->left == NULL && root->right == NULL)
        return root->data;

    // Update left and right subtrees
    int leftsum  = updatetree(root->left);
    int rightsum = updatetree(root->right);

    // Add leftsum to current node
    root->data += leftsum;

    // Return sum of values under root
    return root->data + rightsum;
}
```

Time Complexity: O(n)

104 Iterative Search for a key 'x' in Binary Tree

http://quiz.geeksforgeeks.org/iterative-search-for-a-key-x-in-binary-tree/
Given a Binary Tree and a key to be searched in it, write an iterative method that returns true if key is present in Binary Tree, else false.

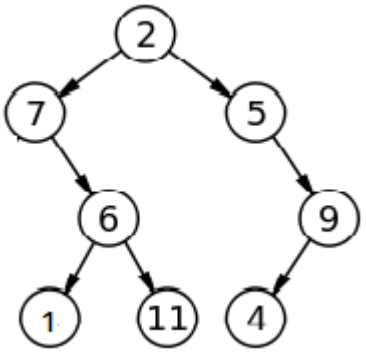
For example, in the following tree, if the searched key is 6, then function should return true and if the searched key is 12, then function should return false.



One thing is sure that we need to traverse complete tree to decide whether key is present or not. **We can use any of the following traversals to iteratively search a key in a given binary tree.**
1) Iterative Level Order Traversal.
2) Iterative Inorder Traversal
3) Iterative Preorder Traversal
4) Iterative Postorder Traversal

105 Find maximum (or minimum) in Binary Tree

http://quiz.geeksforgeeks.org/find-maximum-or-minimum-in-binary-tree/
Given a Binary Tree, find minimum elements in it. For example, maximum in the following Binary Tree is 11.



In Binary Search Tree, we can find maximum by traversing right pointers until we reach rightmost node. But in Binary Tree, we must visit every node to figure out maximum.

So the **idea** is to traverse the given tree and for every node return maximum of 3 values.
1) Node's data.
2) Maximum in node's left subtree.
3) Maximum in node's right subtree.

106 Custom Tree Problem

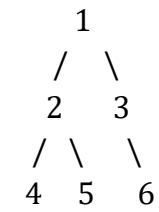
http://www.geeksforgeeks.org/custom-tree-problem/

107 Print Postorder traversal from given Inorder and Preorder traversals

http://www.geeksforgeeks.org/print-postorder-from-given-inorder-and-preorder-traversals/

Given Inorder and Preorder traversals of a binary tree, print Postorder traversal.
Example:
Input:
Inorder traversal in[] = {4, 2, 5, 1, 3, 6}
Preorder traversal pre[] = {1, 2, 4, 5, 3, 6}

Output:
Postorder traversal is {4, 5, 2, 6, 3, 1}
Trversals in the above example represents following tree



A naive method is to first construct the tree, then use simple recursive method to print postorder traversal of the constructed tree.

We can print postorder traversal without constructing the tree. The idea is, root is always the first item in preorder traversal and it must be the last item in postorder traversal.
We first recursively print left subtree, then recursively print right subtree. Finally, print root. To find boundaries of left and right subtrees in pre[] and in[], we search root in in[], all elements before root in in[] are elements of left subtree and all elements after root are elements of right subtree.
In pre[], all elements after index of root in in[] are elements of right subtree. And elements before index (including the element at index and excluding the first element) are elements of left subtree.

```
// Prints postorder traversal from given inorder and preorder traversals
void printPostOrder(int in[], int pre[], int n)
{
   // The first element in pre[] is always root, search it
   // in in[] to find left and right subtrees
   int root = search(in, pre[0], n);

   // If left subtree is not empty, print left subtree
   if (root != 0)
      printPostOrder(in, pre+1, root);

   // If right subtree is not empty, print right subtree
   if (root != n-1)
      printPostOrder(in+root+1, pre+root+1, n-root-1);

   // Print root
   cout << pre[0] << " ";
}
```

Time Complexity: The above function visits every node in array. For every visit, it calls search which takes O(n) time. Therefore, overall time complexity of the function is $O(n^2)$

108 Convert a Binary Tree to Threaded binary tree | Set 1 (Using Queue)

http://www.geeksforgeeks.org/convert-binary-tree-threaded-binary-tree/

We have discussed Threaded Binary Tree. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. In a simple threaded binary tree, the NULL right pointers are used to store inorder successor. Where-ever a right pointer is NULL, it is used to store inorder successor.
Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.

Following is structure of single threaded binary tree.

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

  // Used to indicate whether the right pointer is a normal right
  // pointer or a pointer to inorder successor.
  bool isThreaded;
};
```

How to convert a Given Binary Tree to Threaded Binary Tree?
We basically need to set NULL right pointers to inorder successor. We first do an inorder traversal of the tree and store it in a queue (we can use a simple array also) so that the inorder successor becomes the next node. We again do an inorder traversal and whenever we find a node whose right is NULL, we take the front item from queuue and make it the right of current node. We also set isThreaded to true to indicate that the right pointer is a threaded link.

Methode 2
http://www.geeksforgeeks.org/convert-binary-tree-threaded-binary-tree-set-2-efficient/
Idea of Threaded Binary Tree is to make inorder traversal faster and do it without stack and without recursion. In a simple threaded binary tree, the NULL right pointers are used to store inorder successor. Where-ever a right pointer is NULL, it is used to store inorder successor.

How to convert a Given Binary Tree to Threaded Binary Tree?
In Method 1, a space efficient solution is discussed that doesn't require queue.

The idea is based on the fact that we link from inorder predecessor to a node. We link those inorder predecessor which lie in subtree of node. So we find inorder predecessor of a node if its left is not NULL. Inorder predecessor of a node (whose left is NULL) is rightmost node in left child. Once we find the predecessor, we link a thread from it to current node.
Following is the implementation of the above idea.

```
// Converts tree with given root to threaded
// binary tree.
// This function returns rightmost child of
// root.
Node *createThreaded(Node *root)
{
    // Base cases : Tree is empty or has single
    //              node
    if (root == NULL)
        return NULL;
    if (root->left == NULL &&
        root->right == NULL)
        return root;

    // Find predecessor if it exists
    if (root->left != NULL)
    {
        // Find predecessor of root (Rightmost
        // child in left subtree)
```

```
        Node* l = createThreaded(root->left);

        // Link a thread from predecessor to
        // root.
        l->right = root;
        l->isThreaded = true;
    }

    // If current node is rightmost child
    if (root->right == NULL)
        return root;

    // Recur for right subtree.
    return createThreaded(root->right);
}
```

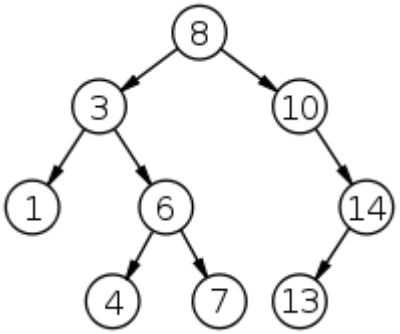
This algorithm works in O(n) **time complexity** and O(1) **space** other than function call stack

More Article on Binary Tree

http://www.geeksforgeeks.org/category/data-structures/tree/page/7/

109 Binary Search Tree | Set 1 (Search and Insertion)

http://quiz.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/
The following is definition of Binary Search Tree(BST) according to Wikipedia
Binary Search Tree, is a node-based binary tree data structure which has the following properties:
The left subtree of a node contains only nodes with keys less than the node's key.
The right subtree of a node contains only nodes with keys greater than the node's key.
The left and right subtree each must also be a binary search tree.
There must be no duplicate nodes.



The above properties of Binary Search Tree provide an ordering among keys so that the operations like search, minimum and maximum can be done fast. If there is no ordering, then we may have to compare every key to search a given key.

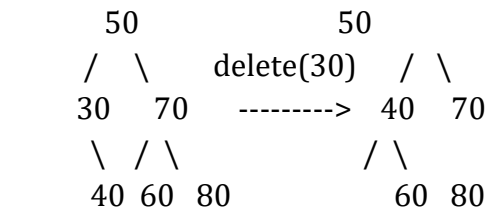
Searching a key
To search a given key in Bianry Search Tree, we first compare it with root, if the key is present at root, we return root. If key is greater than root's key, we recur for right subtree of root node. Otherwise we recur for left subtree

110 Binary Search Tree | Set 2 (Delete)

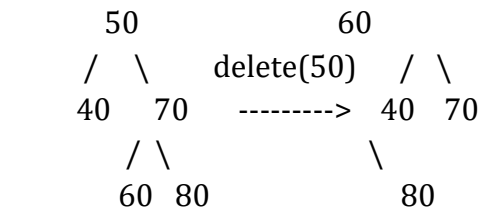
http://quiz.geeksforgeeks.org/binary-search-tree-set-2-delete/
We have discussed BST search and insert operations. In this post, delete operation is discussed. When we delete a node, there possibilities arise.
1) Node to be deleted is leaf: Simply remove from the tree.

```
          50                            50
       /     \         delete(20)      /   \
      30      70       --------->    30     70
     / \     / \                     \    / \
    20  40  60  80                    40  60  80
```

2) Node to be deleted has only one child: Copy the child to the node and delete the child



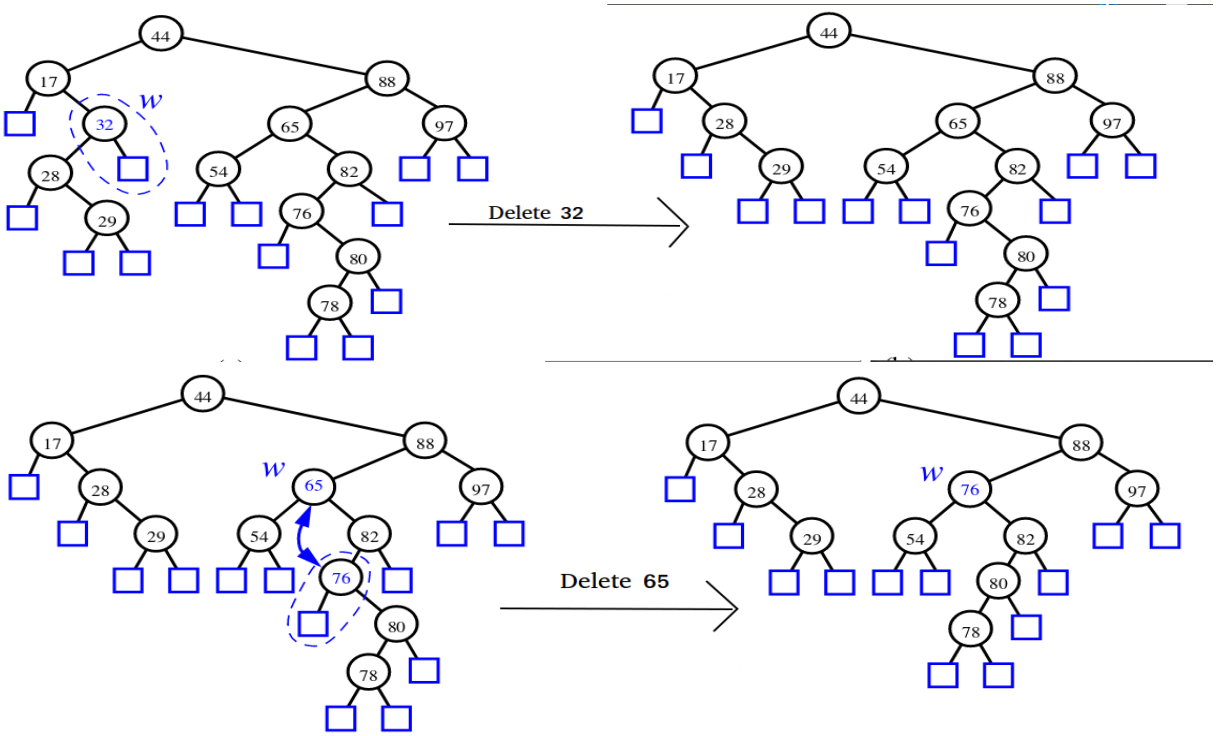
3) Node to be deleted has two children: Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.



The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

```
/* Given a binary search tree and a key, this function deletes the key
   and returns the new root */
struct node* deleteNode(struct node* root, int key)
{
    // base case
    if (root == NULL) return root;

    // If the key to be deleted is smaller than the root's key,
    // then it lies in left subtree
    if (key < root->key)
        root->left = deleteNode(root->left, key);

    // If the key to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
     // if key is same as root's key, then This is the node
    // to be deleted
    else
    {
        // node with only one child or no child
        if (root->left == NULL)
        {
            struct node *temp = root->right;
            free(root);
            return temp;
        }
```

```
        else if (root->right == NULL)
        {
            struct node *temp = root->left;
            free(root);
            return temp;
        }

        // node with two children: Get the inorder successor (smallest
        // in the right subtree)
        struct node* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}
```



Time Complexity: The worst case time complexity of delete operation is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of delete operation may become O(n)

111 Data Structure for a single resource reservations

http://www.geeksforgeeks.org/data-structure-for-future-reservations-for-a-single-resource/

Design a data structure to do reservations of future jobs on a single machine under following constraints.
1) Every job requires exactly k time units of the machine.
2) The machine can do only one job at a time.
3) Time is part of the system. Future Jobs keep coming at different times. Reservation of a future job is done only if there is no existing reservation within k time frame (after and before)
4) Whenever a job finishes (or its reservation time plus k becomes equal to current time), it is removed from system.

Example:
Let time taken by a job (or k) be = 4

At time 0: Reservation request for a job at time 2 in
future comes in, reservation is done as machine
will be available (no conflicting reservations)
Reservations {2}

At time 3: Reservation requests at times 15, 7, 20 and 3.
Job at 7, 15 and 20 can be reserved, but at 3
cannot be reserved as it conflicts with a
reserved at 2.
Reservations {2, 7, 15, 20}

At time 6: Reservation requests at times 30, 17, 35 and 45
Jobs at 30, 35 and 45 are reserved, but at 17
cannot be reserved as it conflicts with a reserved
at 15.
Reservations {7, 15, 30, 35, 45}.
Note that job at 2 is removed as it must be finished by 6.
Let us consider different data structures for this task.

One **solution is to keep all future reservations sorted in array**. Time complexity of checking for conflicts can be done in O(Logn) using Binary Search, but insertions and deletions take O(n) time.

Hashing cannot be used here as the search is not exact search, but a search within k time frame.

The **idea** is to **use Binary Search Tree** to maintain set of reserved jobs. For every reservation request, insert it only when there is no conflicting reservation. While inserting job, do "within k time frame check". If there is a k distant node on insertion path from root, then reject the reservation request, otherwise do the reservation.

```
/* BST insert to process a new reservation request at
   a given time (future time).  This function does
   reservation only if there is no existing job within
   k time frame of new job  */
struct node* insert(struct node* root, int time, int k)
{
    /* If the tree is empty, return a new node */
    if (root == NULL) return newNode(time);

    // Check if this job conflicts with existing
    // reservations
    if ((time-k < root->time) && (time+k > root->time))
        return root;

    /* Otherwise, recur down the tree */
    if (time < root->time)
        root->left  = insert(root->left, time, k);
    else
        root->right = insert(root->right, time, k);

    /* return the (unchanged) node pointer */
    return root;
}
```

Deletion of job is simple BST delete operation.
A normal BST takes O(h) time for insert and delete operations. We can use self-balancing binary search trees like AVL, Red-Black, .. to do both operations in O(Log n) time.

112 Advantages of BST over Hash Table

http://www.geeksforgeeks.org/advantages-of-bst-over-hash-table/
Hash Table supports following operations in Θ(1) time.
1) Search
2) Insert
3) Delete
The **time complexity** of above operations in a self-balancing Binary Search Tree (BST) (like Red-Black Tree, AVL Tree, Splay Tree, etc) is O(Logn).

So Hash Table seems to beating BST in all common operations. When should we prefer BST over Hash Tables, what are advantages. Following are some important points in favor of BSTs.

We can get all keys in sorted order by just doing Inorder Traversal of BST. This is not a natural operation in Hash Tables and requires extra efforts.
Doing order statistics, finding closest lower and greater elements, doing range queries are easy to do with BSTs. Like sorting, these operations are not a natural operation with Hash Tables.

BSTs are easy to implement compared to hashing, we can easily implement our own customized BST. To implement Hashing, we generally rely on libraries provided by programming languages.
With BSTs, all operations are guaranteed to work in O(Logn) time. But with Hashing, Θ(1) is average time and some particular operations may be costly, especially when table resizing happens.

113 Find the node with minimum value in a Binary Search Tree

http://www.geeksforgeeks.org/find-the-minimum-element-in-a-binary-search-tree/
This is quite simple. Just traverse the node from root to left recursively until left is NULL. The node whose left is NULL is the node with minimum value.
Time Complexity: O(n) Worst case happens for left skewed trees.

114 Inorder predecessor and successor for a given key in BST

http://www.geeksforgeeks.org/inorder-predecessor-successor-given-key-bst/

You need to find the inorder successor and predecessor of a given key. In case the given key is not found in BST, then return the two values within which this key will lie.
Following is the algorithm to reach the desired result. Its a recursive method:
Input: root node, key
output: predecessor node, successor node

1. If root is NULL
   then return
2. if key is found then
   a. If its left subtree is not null
      Then predecessor will be the right most
      child of left subtree or left child itself.
   b. If its right subtree is not null
      The successor will be the left most child
      of right subtree or right child itself.
   return
3. If key is smaller then root node
      set the successor as root
      search recursively into left subtree
   else
      set the predecessor as root
      search recursively into right subtree

```
// This function finds predecessor and successor of key in BST.
// It sets pre and suc as predecessor and successor respectively
void findPreSuc(Node* root, Node*& pre, Node*& suc, int key)
{
    // Base case
    if (root == NULL)  return ;

    // If key is present at root
    if (root->key == key)
    {
        // the maximum value in left subtree is predecessor
        if (root->left != NULL)
        {
            Node* tmp = root->left;
```

```
            while (tmp->right)
                tmp = tmp->right;
            pre = tmp ;
        }

        // the minimum value in right subtree is successor
        if (root->right != NULL)
        {
            Node* tmp = root->right ;
            while (tmp->left)
                tmp = tmp->left ;
            suc = tmp ;
        }
        return ;
    }

    // If key is smaller than root's key, go to left subtree
    if (root->key > key)
    {
        suc = root ;
        findPreSuc(root->left, pre, suc, key) ;
    }
    else // go to right subtree
    {
        pre = root ;
        findPreSuc(root->right, pre, suc, key) ;
    }
}
```

115 Lowest Common Ancestor in a Binary Search Tree.

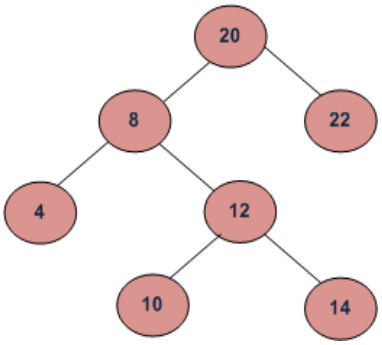
http://www.geeksforgeeks.org/lowest-common-ancestor-in-a-binary-search-tree/

Given values of two nodes in a Binary Search Tree, write a c program to find the Lowest Common Ancestor (LCA). You may assume that both the values exist in the tree.

The function prototype should be as follows:

struct node *lca(node* root, int n1, int n2)

n1 and n2 are two given values in the tree with given root.



For example, consider the BST in diagram, LCA of 10 and 14 is 12 and LCA of 8 and 14 is 8.
Following is definition of LCA from Wikipedia:
Let T be a rooted tree. The lowest common ancestor between two nodes n1 and n2 is defined as the lowest node in T that has both n1 and n2 as descendants (where we allow a node to be a descendant of itself).

The LCA of n1 and n2 in T is the shared ancestor of n1 and n2 that is located farthest from the root. Computation of lowest common ancestors may be useful, for instance, as part of a procedure for determining the distance between pairs of nodes in a tree: the distance from n1 to n2 can be computed as the distance from the root to n1, plus the distance from the root to n2, minus twice the distance from the root to their lowest common ancestor. (Source Wiki)

Solutions:
If we are given a BST where every node has parent pointer, then LCA can be easily determined by traversing up using parent pointer and printing the first intersecting node.

We can solve this problem using BST properties. We can recursively traverse the BST from root. The main idea of the solution is, while traversing from top to bottom, the first node n we encounter with value between n1 and n2, i.e., n1 < n < n2 or same as one of the n1 or n2, is LCA of n1 and n2 (assuming that n1 < n2).
So just recursively traverse the BST in, if node's value is greater than both n1 and n2 then our LCA lies in left side of the node, if it's is smaller than both n1 and n2, then LCA lies on right side. Otherwise root is LCA (assuming that both n1 and n2 are present in BST)

```
/* Function to find LCA of n1 and n2. The function assumes that both
   n1 and n2 are present in BST */
struct node *lca(struct node* root, int n1, int n2)
{
    if (root == NULL) return NULL;

    // If both n1 and n2 are smaller than root, then LCA lies in left
    if (root->data > n1 && root->data > n2)
        return lca(root->left, n1, n2);

    // If both n1 and n2 are greater than root, then LCA lies in right
    if (root->data < n1 && root->data < n2)
        return lca(root->right, n1, n2);

    return root;
}
```

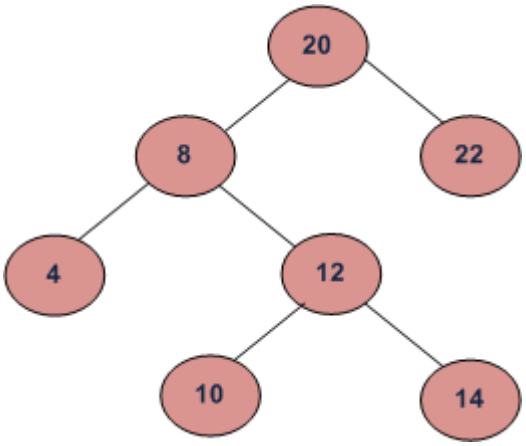
Time complexity of above solution is O(h) where h is height of tree. Also, the above solution requires O(h) extra space in function call stack for recursive function calls. **We can avoid extra space using** iterative solution**.**

Exercise
The above functions assume that n1 and n2 both are in BST. If n1 and n2 are not present, then they may return incorrect result. Extend the above solutions to return NULL if n1 or n2 or both not present in BST.

Find LCA in Binary Tree using RMQ
http://www.geeksforgeeks.org/find-lca-in-binary-tree-using-rmq/

116 Inorder Successor in Binary Search Tree

http://www.geeksforgeeks.org/inorder-successor-in-binary-search-tree/
In Binary Tree, Inorder successor of a node is the next node in Inorder traversal of the Binary Tree. Inorder Successor is NULL for the last node in Inoorder traversal.
In Binary Search Tree, Inorder Successor of an input node can also be defined as the node with the smallest key greater than the key of input node. So, it is sometimes important to find next node in sorted order.



In the above diagram, inorder successor of 8 is 10, inorder successor of 10 is 12 and inorder successor of 14 is 20.

Method 1 (Uses Parent Pointer)
In this method, we assume that every node has parent pointer.
The Algorithm is divided into two cases on the basis of right subtree of the input node being empty or not.

Input: node, root // node is the node whose Inorder successor is needed.
output: succ // succ is Inorder successor of node.

1) If right subtree of node is not NULL, then succ lies in right subtree. Do following.
Go to right subtree and return the node with minimum key value in right subtree.

2) If right sbtree of node is NULL, then succ is one of the ancestors. Do following.
Travel up using the parent pointer until you see a node which is left child of it's parent.
The parent of such a node is the succ.

Implementation
Note that the function to find InOrder Successor is highlighted (with gray background) in below code.

```
struct node * inOrderSuccessor(struct node *root, struct node *n)
{
  // step 1 of the above algorithm
  if( n->right != NULL )
    return minValue(n->right);

  // step 2 of the above algorithm
  struct node *p = n->parent;
  while(p != NULL && n == p->right)
  {
     n = p;
     p = p->parent;
  }
  return p;
}
```

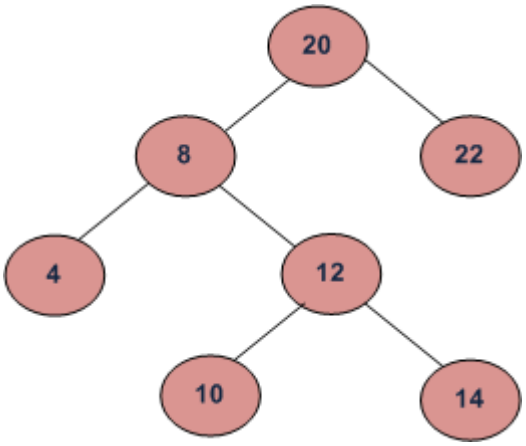
Time Complexity: O(h) where h is height of tree.

Method 2 (Search from root)
Parent pointer is NOT needed in this algorithm. The Algorithm is divided into two cases on the basis of right subtree of the input node being empty or not.
Input: node, root // node is the node whose Inorder successor is needed.
output: succ // succ is Inorder successor of node.
1) If right subtree of node is not NULL, then succ lies in right subtree. Do following.
Go to right subtree and return the node with minimum key value in right subtree.
2) If right sbtree of node is NULL, then start from root and us search like technique. Do following.

Travel down the tree, if a node's data is greater than root's data then go right side, otherwise go to left side.
Time Complexity: O(h) where h is height of tree.

117 Find k-th smallest element in BST (Order Statistics in BST)

http://www.geeksforgeeks.org/find-k-th-smallest-element-in-bst-order-statistics-in-bst/
Given root of binary search tree and K as input, find K-th smallest element in BST.
For example, in the following BST, if k = 3, then output should be 10, and if k = 5, then output should be 14.



Method 1: Using Inorder Traversal.
Inorder traversal of BST retrieves elements of tree in the sorted order. The inorder traversal uses stack to store to be explored nodes of tree (threaded tree avoids stack and recursion for traversal, see this post). The idea is to keep track of popped elements which participate in the order statics. Hypothetical algorithm is provided below,
Time complexity: O(n) where n is total nodes in tree..

Algorithm:

```
/* initialization */
pCrawl = root
set initial stack element as NULL (sentinal)
/* traverse upto left extreme */
while(pCrawl is valid )
  stack.push(pCrawl)
  pCrawl = pCrawl.left
```

```
/* process other nodes */
while( pCrawl = stack.pop() is valid )
  stop if sufficient number of elements are popped.
  if( pCrawl.right is valid )
    pCrawl = pCrawl.right
    while( pCrawl is valid )
      stack.push(pCrawl)
      pCrawl = pCrawl.left
```

Method 2: Augmented Tree Data Structure.

The idea is to maintain rank of each node. We can keep track of elements in a subtree of any node while building the tree. Since we need K-th smallest element, we can maintain number of elements of left subtree in every node.

Assume that the root is having N nodes in its left subtree. If K = N + 1, root is K-th node. If K < N, we will continue our search (recursion) for the Kth smallest element in the left subtree of root. If K > N + 1, we continue our search in the right subtree for the (K – N – 1)-th smallest element. Note that we need the count of elements in left subtree only.
Time complexity: O(h) where h is height of tree.

Algorithm:

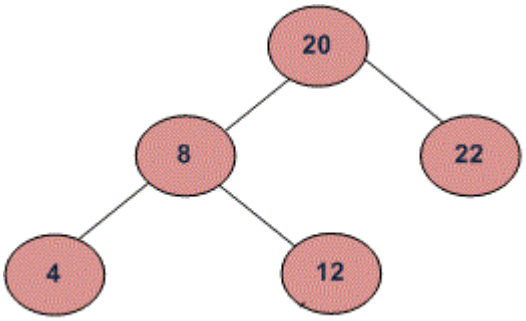
```
start:
if K = root.leftElement + 1
  root node is the K th node.
  goto stop
else if K > root.leftElements
  K = K - (root.leftElements + 1)
  root = root.right
  goto start
else
  root = root.left
  goto srart

stop:
```

118 K'th smallest element in BST using O(1) Extra Space

http://www.geeksforgeeks.org/kth-smallest-element-in-bst-using-o1-extra-space/
The **idea** is to use **Morris Traversal.** In this traversal, we first create links to Inorder successor and print the data using these links, and finally revert the changes to restore original tree.

119 Print BST keys in the given range

http://www.geeksforgeeks.org/print-bst-keys-in-the-given-range/
Given two values k1 and k2 (where k1 < k2) and a root pointer to a Binary Search Tree. Print all the keys of tree in range k1 to k2. i.e. print all x such that k1<=x<=k2 and x is a key of given BST. Print all the keys in increasing order.
For example, if k1 = 10 and k2 = 22, then your function should print 12, 20 and 22.



Algorithm:
1) If value of root's key is greater than k1, then recursively call in left subtree.
2) If value of root's key is in range, then print the root's key.
3) If value of root's key is smaller than k2, then recursively call in right subtree.

Time Complexity: O(n) where n is the total number of keys in tree.

120 Sorted Array to Balanced BST

http://www.geeksforgeeks.org/sorted-array-to-balanced-bst/
Given a sorted array. Write a function that creates a Balanced Binary Search Tree using array elements.
Examples:
Input: Array {1, 2, 3}
Output: A Balanced BST

```
  2
 / \
1   3
```

Input: Array {1, 2, 3, 4}
Output: A Balanced BST

```
    3
   / \
  2   4
 /
1
```

Algorithm

Constructing from sorted array in O(n) time is simpler as we can get the middle element in O(1) time. Following is a simple algorithm where we first find the middle node of list and make it root of the tree to be constructed.

1) Get the Middle of the array and make it root.
2) Recursively do same for left half and right half.
   a) Get the middle of left half and make it left child of the root created in step 1.
   b) Get the middle of right half and make it right child of the root created in step 1.

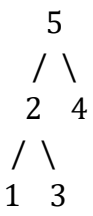
Time Complexity: O(n)
Following is the recurrance relation for sortedArrayToBST().

T(n) = 2T(n/2) + C
T(n) --> Time taken for an array of size n
C --> Constant (Finding middle of array and linking root to left and right subtrees take constant time)

121 Find the largest BST subtree in a given Binary Tree

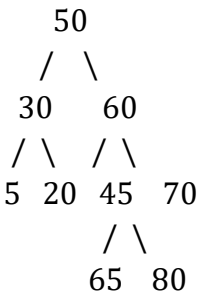
http://www.geeksforgeeks.org/find-the-largest-subtree-in-a-tree-that-is-also-a-bst/
Given a Binary Tree, write a function that returns the size of the largest subtree which is also a Binary Search Tree (BST). If the complete Binary Tree is BST, then return the size of whole tree.

Examples:
Input:

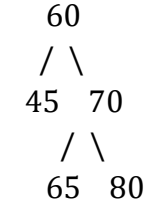


Output: 3
The following subtree is the maximum size BST subtree

Input:



Output: 5
The following subtree is the maximum size BST subtree



Method 1 (Simple but inefficient)
Start from root and do an inorder traversal of the tree. For each node N, check whether the subtree rooted with N is BST or not. If BST, then return size of the subtree rooted with N. Else, recur down the left and right subtrees and return the maximum of values returned by left and right subtrees.
Time Complexity: The worst case time complexity of this method will be O(n^2). Consider a skewed tree for worst case analysis.

Method 2 (Tricky and Efficient)

In method 1, we traverse the tree in top down manner and do BST test for every node. If we traverse the tree in bottom up manner, then we can pass information about subtrees to the parent. The passed information can be used by the parent to do BST test (for parent node) only in constant time (or O(1) time). A left subtree need to tell the parent whether it is BST or not and also need to pass maximum value in it. So that we can compare the maximum value with the parent's data to check the BST property. Similarly, the right subtree need to pass the minimum value up the tree. The subtrees need to pass the following information up the tree for the finding the largest BST.

1) Whether the subtree itself is BST or not (In the following code, is_bst_ref is used for this purpose)
2) If the subtree is left subtree of its parent, then maximum value in it. And if it is right subtree then minimum value in it.
3) Size of this subtree if this subtree is BST (In the following code, return value of largestBSTtil() is used for this purpose)
max_ref is used for passing the maximum value up the tree and min_ptr is used for passing minimum value up the tree.

```
/* Returns size of the largest BST subtree in a Binary Tree
  (efficient version). */
int largestBST(struct node* node)
{
  // Set the initial values for calling largestBSTUtil()
  int min = INT_MAX;  // For minimum value in right subtree
  int max = INT_MIN;  // For maximum value in left subtree

  int max_size = 0;  // For size of the largest BST
  bool is_bst = 0;

  largestBSTUtil(node, &min, &max, &max_size, &is_bst);

  return max_size;
}

/* largestBSTUtil() updates *max_size_ref for the size of the largest BST
   subtree.   Also, if the tree rooted with node is non-empty and a BST,
   then returns size of the tree. Otherwise returns 0.*/
int largestBSTUtil(struct node* node, int *min_ref, int *max_ref,
                             int *max_size_ref, bool *is_bst_ref)
{

  /* Base Case */
  if (node == NULL)
  {
     *is_bst_ref = 1; // An empty tree is BST
```

```
    return 0;    // Size of the BST is 0
  }

  int min = INT_MAX;

  /* A flag variable for left subtree property
     i.e., max(root->left) < root->data */
  bool left_flag = false;

  /* A flag variable for right subtree property
     i.e., min(root->right) > root->data */
  bool right_flag = false;

  int ls, rs; // To store sizes of left and right subtrees

  /* Following tasks are done by recursive call for left subtree
    a) Get the maximum value in left subtree (Stored in *max_ref)
    b) Check whether Left Subtree is BST or not (Stored in *is_bst_ref)
    c) Get the size of maximum size BST in left subtree (updates
*max_size) */
  *max_ref = INT_MIN;
  ls = largestBSTUtil(node->left, min_ref, max_ref, max_size_ref,
is_bst_ref);
  if (*is_bst_ref == 1 && node->data > *max_ref)
     left_flag = true;

  /* Before updating *min_ref, store the min value in left subtree. So
that we
     have the correct minimum value for this subtree */
  min = *min_ref;

  /* The following recursive call does similar (similar to left subtree)
    task for right subtree */
  *min_ref =  INT_MAX;
  rs = largestBSTUtil(node->right, min_ref, max_ref, max_size_ref,
is_bst_ref);
  if (*is_bst_ref == 1 && node->data < *min_ref)
     right_flag = true;

  // Update min and max values for the parent recursive calls
  if (min < *min_ref)
     *min_ref = min;
  if (node->data < *min_ref) // For leaf nodes
     *min_ref = node->data;
  if (node->data > *max_ref)
     *max_ref = node->data;

  /* If both left and right subtrees are BST. And left and right
     subtree properties hold for this node, then this tree is BST.
     So return the size of this tree */
  if(left_flag && right_flag)
  {
     if (ls + rs + 1 > *max_size_ref)
         *max_size_ref = ls + rs + 1;
     return ls + rs + 1;
```

```
  }
  else
  {
    //Since this subtree is not BST, set is_bst flag for parent calls
     *is_bst_ref = 0;
     return 0;
  }
}
```

Time Complexity: O(n) where n is the number of nodes in the given Binary Tree.

122 Check for Identical BSTs without building the trees

http://www.geeksforgeeks.org/check-for-identical-bsts-without-building-the-trees/

Given two arrays which represent a sequence of keys. Imagine we make a Binary Search Tree (BST) from each array. We need to tell whether two BSTs will be identical or not without actually constructing the tree.

Examples

For example, the input arrays are {2, 4, 3, 1} and {2, 1, 4, 3} will construct the same tree

Let the input arrays be a[] and b[]

Example 1:

a[] = {2, 4, 1, 3} will construct following tree.

```
  2
 / \
1   4
   /
  3
```

b[] = {2, 4, 3, 1} will also also construct the same tree.

```
  2
 / \
1   4
   /
  3
```

So the output is "True"

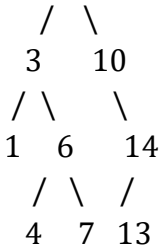
Example 2:

a[] = {8, 3, 6, 1, 4, 7, 10, 14, 13}

b[] = {8, 10, 14, 3, 6, 4, 1, 7, 13}

They both construct the same following BST, so output is "True"

```
        8
```



Solution:

According to BST property, elements of left subtree must be smaller and elements of right subtree must be greater than root.

Two arrays represent sane BST if for every element x, the elements in left and right subtrees of x appear after it in both arrays. And same is true for roots of left and right subtrees.

The **idea** is to **check of if next smaller and greater elements are same in both arrays**. Same properties are recursively checked for left and right subtrees. The idea looks simple, but implementation requires checking all conditions for all elements. Following is an interesting recursive implementation of the idea.

```
/* The main function that checks if two arrays a[] and b[] of size n
construct
  same BST. The two values 'min' and 'max' decide whether the call is made
for
  left subtree or right subtree of a parent element. The indexes i1 and i2
are
  the indexes in (a[] and b[]) after which we search the left or right
child.
  Initially, the call is made for INT_MIN and INT_MAX as 'min' and 'max'
  respectively, because root has no parent.
  i1 and i2 are just after the indexes of the parent element in a[] and
b[]. */
bool isSameBSTUtil(int a[], int b[], int n, int i1, int i2, int min, int
max)
{
   int j, k;

   /* Search for a value satisfying the constraints of min and max in a[]
and
      b[]. If the parent element is a leaf node then there must be some
      elements in a[] and b[] satisfying constraint. */
   for (j=i1; j<n; j++)
       if (a[j]>min && a[j]<max)
           break;
   for (k=i2; k<n; k++)
       if (b[k]>min && b[k]<max)
           break;
   /* If the parent element is leaf in both arrays */
   if (j==n && k==n)
```

```
        return true;

    /* Return false if any of the following is true
       a) If the parent element is leaf in one array, but non-leaf in
other.
       b) The elements satisfying constraints are not same. We either
search
          for left child or right child of the parent element (decinded by
min
          and max values). The child found must be same in both arrays */
    if (((j==n)^(k==n)) || a[j]!=b[k])
        return false;

    /* Make the current child as parent and recursively check for left and
right
       subtrees of it. Note that we can also pass a[k] in place of a[j] as
they
       are both are same */
    return isSameBSTUtil(a, b, n, j+1, k+1, a[j], max) &&  // Right Subtree
           isSameBSTUtil(a, b, n, j+1, k+1, min, a[j]);    // Left Subtree
}
```

123 Add all greater values to every node in a given BST

http://www.geeksforgeeks.org/add-greater-values-every-node-given-bst/

Given a Binary Search Tree (BST), modify it so that all greater values in the given BST are added to every node. For example, consider the following BST.

```
       50
     /   \
    30    70
   / \   / \
  20  40 60  80
```

The above tree should be modified to following

```
       260
     /   \
    330   150
   / \   / \
 350 300 210  80
```

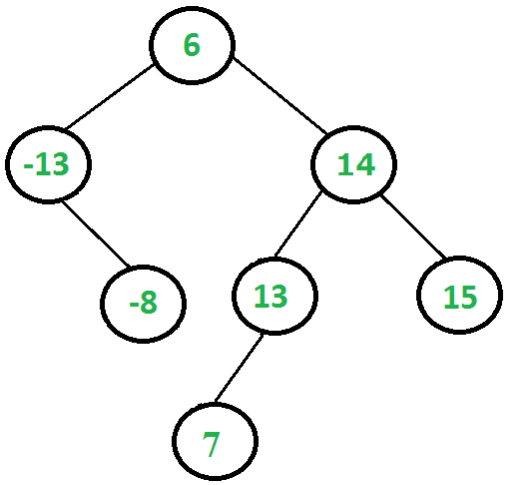
A **simple method** for solving this is to **find sum of all greater values for every node.** This method would take **O(n^2) time.**

We can do it **using a single traversal**. The **idea** is to use following BST property. If we do reverse Inorder traversal of BST, we get all nodes in decreasing order. We do

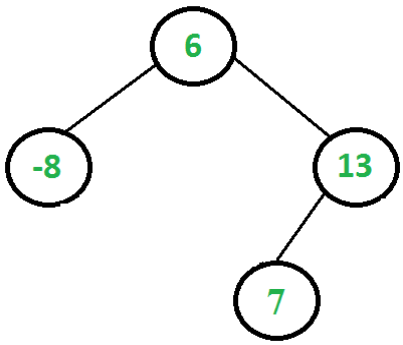
reverse Inorder traversal and keep track of the sum of all nodes visited so far, we add this sum to every node.
Time Complexity: O(n) where n is number of nodes in the given BST.
As a side note, we can also use reverse Inorder traversal to find kth largest element in a BST.

124 Remove BST keys outside the given range

http://www.geeksforgeeks.org/remove-bst-keys-outside-the-given-range/
Given a Binary Search Tree (BST) and a range [min, max], remove all keys which are outside the given range. The modified tree should also be BST. For example, consider the following BST and range [-10, 13].



The given tree should be changed to following. Note that all keys outside the range [-10, 13] are removed and modified tree is BST.



There are two possible cases for every node.
1) Node's key is outside the given range. This case has two sub-cases.
…….a) Node's key is smaller than the min value.
…….b) Node's key is greater that the max value.
2) Node's key is in range.
We don't need to do anything for case 2. In case 1, we need to remove the node and change root of sub-tree rooted with this node.

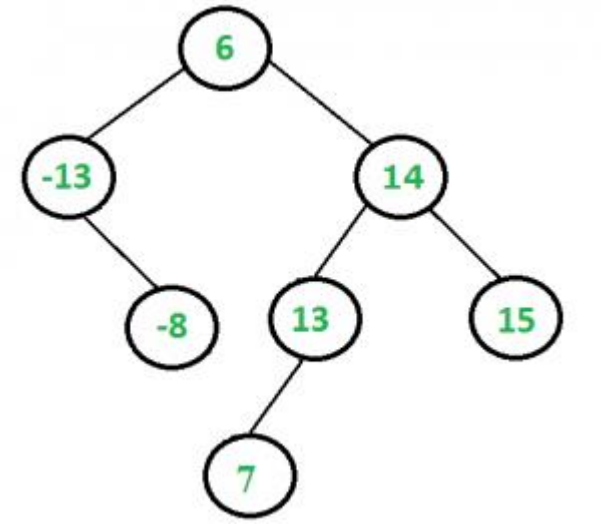
The **idea** is to **fix the tree in Postorder fashion**. When we visit a node, we make sure that its left and right sub-trees are already fixed. In case 1.a), we simply remove root and return right sub-tree as new root. In case 1.b), we remove root and return left sub-tree as new root.

```
// Resmoves all nodes having value outside the given range and returns the
root
// of modified tree
node* removeOutsideRange(node *root, int min, int max)
{
   // Base Case
   if (root == NULL)
      return NULL;

   // First fix the left and right subtrees of root
   root->left =  removeOutsideRange(root->left, min, max);
   root->right =  removeOutsideRange(root->right, min, max);

   // Now fix the root.  There are 2 possible cases for toot
   // 1.a) Root's key is smaller than min value (root is not in range)
   if (root->key < min)
   {
       node *rChild = root->right;
       delete root;
       return rChild;
   }
   // 1.b) Root's key is greater than max value (root is not in range)
   if (root->key > max)
   {
       node *lChild = root->left;
       delete root;
       return lChild;
   }
   // 2. Root is in range
   return root;
}
```

Time Complexity: O(n) where n is the number of nodes in given BST.

125 Find if there is a triplet in a Balanced BST that adds to zero

http://www.geeksforgeeks.org/find-if-there-is-a-triplet-in-bst-that-adds-to-0/
Given a Balanced Binary Search Tree (BST), write a function isTripletPresent() that returns true if there is a triplet in given BST with sum equals to 0, otherwise returns false. Expected time complexity is O(n^2) and only O(Logn) extra space can be used. You can modify given Binary Search Tree. Note that height of a Balanced BST is always O(Logn)
For example, isTripletPresent() should return true for following BST because there is a triplet with sum 0, the triplet is {-13, 6, 7}.



The **Brute Force Solution** is to consider each triplet in BST and check whether the sum adds upto zero.
The **time complexity** of this solution will be O(n^3).

A **Better Solution** is to **create an auxiliary array and store Inorder traversal of BST in the array.** The array will be sorted as Inorder traversal of BST always produces sorted data. Once we have the Inorder traversal, we can use method 2 of this post to find the triplet with sum equals to 0.
This solution works in **O(n^2) time**, but requires **O(n) auxiliary space**.

Following is the solution that works in O(n^2) time and uses O(Logn) extra space:
1) Convert given BST to Doubly Linked List (DLL)
2) Now iterate through every node of DLL and if the key of node is negative, then find a pair in DLL with sum equal to key of current node multiplied by -1.

```
// The main function that returns true if there is a 0 sum triplet in
// BST otherwise returns false
bool isTripletPresent(node *root)
{
    // Check if the given  BST is empty
    if (root == NULL)
       return false;

    // Convert given BST to doubly linked list.  head and tail store the
    // pointers to first and last nodes in DLLL
    node* head = NULL;
    node* tail = NULL;
    convertBSTtoDLL(root, &head, &tail);

    // Now iterate through every node and find if there is a pair with sum
    // equal to -1 * heaf->key where head is current node
    while ((head->right != tail) && (head->key < 0))
    {
        // If there is a pair with sum equal to  -1*head->key, then return
        // true else move forward
        if (isPresentInDLL(head->right, tail, -1*head->key))
            return true;
        else
            head = head->right;
    }

    // If we reach here, then there was no 0 sum triplet
    return false;
}
```

Note that the above solution modifies given BST.
Time Complexity: Time taken to convert BST to DLL is O(n) and time taken to find triplet in DLL is O(n^2).
Auxiliary Space: The auxiliary space is needed only for function call stack in recursive function convertBSTtoDLL(). Since given tree is balanced (height is O(Logn)), the number of functions in call stack will never be more than O(Logn).
We can also find triplet in same time and extra space without modifying the tree.

126 Check if each internal node of a BST has exactly one child

http://www.geeksforgeeks.org/check-if-each-internal-node-of-a-bst-has-exactly-one-child/

Given Preorder traversal of a BST, check if each non-leaf node has only one child. Assume that the BST contains unique entries.

Examples

Input: pre[] = {20, 10, 11, 13, 12}

Output: Yes

The give array represents following BST. In the following BST, every internal node has exactly 1 child. Therefor, the output is true.

```
    20
   /
  10
   \
    11
     \
      13
     /
    12
```

In Preorder traversal, descendants (or Preorder successors) of every node appear after the node. In the above example, 20 is the first node in preorder and all descendants of 20 appear after it. All descendants of 20 are smaller than it. For 10, all descendants are greater than it. In general, we can say, if all internal nodes have only one child in a BST, then all the descendants of every node are either smaller or larger than the node. The reason is simple, since the tree is BST and every node has only one child, all descendants of a node will either be on left side or right side, means all descendants will either be smaller or greater.

Approach 1 (Naive)

This approach simply follows the above idea that all values on right side are either smaller or larger. Use two loops, the outer loop picks an element one by one, starting from the leftmost element. The inner loop checks if all elements on the right side of the picked element are either smaller or greater.

The **time complexity** of this method will be O(n^2).

Approach 2

Since all the descendants of a node must either be larger or smaller than the node. We can do following for every node in a loop.

1. Find the next preorder successor (or descendant) of the node.
2. Find the last preorder successor (last element in pre[]) of the node.
3. If both successors are less than the current node, or both successors are greater than the current node, then continue. Else, return false.

```
bool hasOnlyOneChild(int pre[], int size)
{
    int nextDiff, lastDiff;

    for (int i=0; i<size-1; i++)
    {
        nextDiff = pre[i] - pre[i+1];
        lastDiff = pre[i] - pre[size-1];
        if (nextDiff*lastDiff < 0)
            return false;;
    }
    return true;
}
```

Approach 3

1. Scan the last two nodes of preorder & mark them as min & max.
2. Scan every node down the preorder array. Each node must be either smaller than the min node or larger than the max node. Update min & max accordingly.

127 Merge Two Balanced Binary Search Trees

http://www.geeksforgeeks.org/merge-two-balanced-binary-search-trees/

You are given two balanced binary search trees e.g., AVL or Red Black Tree. Write a function that merges the two given balanced BSTs into a balanced binary search tree. Let there be m elements in first tree and n elements in the other tree. Your merge function should take O(m+n) time.

Method 1 (Insert elements of first tree to second)

Take all elements of first BST one by one, and insert them into the second BST. Inserting an element to a self balancing BST takes Logn time (See this) where n is size of the BST. So time complexity of this method is Log(n) + Log(n+1) … Log(m+n-1). The value of this expression will be between mLogn and mLog(m+n-1). As an optimization, we can pick the smaller tree as first tree.

Method 2 (Merge Inorder Traversals)
1) Do inorder traversal of first tree and store the traversal in one temp array arr1[]. This step takes O(m) time.
2) Do inorder traversal of second tree and store the traversal in another temp array arr2[]. This step takes O(n) time.
3) The arrays created in step 1 and 2 are sorted arrays. Merge the two sorted arrays into one array of size m + n. This step takes O(m+n) time.
4) Construct a balanced tree from the merged array.This step takes O(m+n) time.
Time complexity of this method is O(m+n) which is better than method 1. This method takes O(m+n) time even if the input BSTs are not balanced.

Method 3 (In-Place Merge using DLL)
We can use a Doubly Linked List to merge trees in place. Following are the steps.
1) Convert the given two Binary Search Trees into doubly linked list in place for this step).
2) Merge the two sorted Linked Lists
3) Build a Balanced Binary Search Tree from the merged list created in step 2.
Time complexity of this method is also O(m+n) and this method does conversion in place.

128 Merge two BSTs with limited extra space

http://www.geeksforgeeks.org/merge-two-bsts-with-limited-extra-space/
Given two Binary Search Trees(BST), print the elements of both BSTs in sorted form. The expected time complexity is O(m+n) where m is the number of nodes in first tree and n is the number of nodes in second tree. Maximum allowed auxiliary space is O(height of the first tree + height of the second tree).
Examples:
First BST

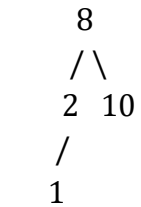
```
    3
  /   \
 1     5
```

Second BST

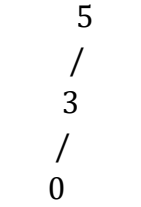
```
   4
 /  \
2    6
```

Output: 1 2 3 4 5 6

First BST



Second BST



Output: 0 1 2 3 5 8 10

A similar question has been discussed earlier. Let us first discuss already discussed methods of the previous post which was for Balanced BSTs. The method 1 can be applied here also, but the **time complexity** will be O(n^2) in worst case.

The method 2 can also be applied here, but the extra space required will be O(n) which violates the constraint given in this question. Method 3 can be applied here but the step 3 of method 3 can't be done in O(n) for an unbalanced BST.

The **idea** is to use **iterative inorder traversal**. We **use two auxiliary stacks for two BSTs**. Since we need to print the elements in sorted form, whenever we get a smaller element from any of the trees, we print it. If the element is greater, then we push it back to stack for the next iteration.

```
// The function to print data of two BSTs in sorted order
void  merge(struct node *root1, struct node *root2)
{
    // s1 is stack to hold nodes of first BST
    struct snode *s1 = NULL;

    // Current node of first BST
    struct node  *current1 = root1;

    // s2 is stack to hold nodes of second BST
    struct snode *s2 = NULL;

    // Current node of second BST
    struct node  *current2 = root2;

    // If first BST is empty, then output is inorder
    // traversal of second BST
```

```
if (root1 == NULL)
{
    inorder(root2);
    return;
}
// If second BST is empty, then output is inorder
// traversal of first BST
if (root2 == NULL)
{
    inorder(root1);
    return ;
}

// Run the loop while there are nodes not yet printed.
// The nodes may be in stack(explored, but not printed)
// or may be not yet explored
while (current1 != NULL || !isEmpty(s1) ||
      current2 != NULL || !isEmpty(s2))
{
    // Following steps follow iterative Inorder Traversal
    if (current1 != NULL || current2 != NULL )
    {
        // Reach the leftmost node of both BSTs and push ancestors of
        // leftmost nodes to stack s1 and s2 respectively
        if (current1 != NULL)
        {
            push(&s1, current1);
            current1 = current1->left;
        }
        if (current2 != NULL)
        {
            push(&s2, current2);
            current2 = current2->left;
        }

    }
    else
    {
        // If we reach a NULL node and either of the stacks is empty,
        // then one tree is exhausted, ptint the other tree
        if (isEmpty(s1))
        {
            while (!isEmpty(s2))
            {
                current2 = pop (&s2);
                current2->left = NULL;
                inorder(current2);
            }
            return ;
        }
        if (isEmpty(s2))
        {
```

```
while (!isEmpty(s1))
                {
                    current1 = pop (&s1);
                    current1->left = NULL;
                    inorder(current1);
                }
                return ;
            }

            // Pop an element from both stacks and compare the
            // popped elements
            current1 = pop(&s1);
            current2 = pop(&s2);

            // If element of first tree is smaller, then print it
            // and push the right subtree. If the element is larger,
            // then we push it back to the corresponding stack.
            if (current1->data < current2->data)
            {
                printf("%d ", current1->data);
                current1 = current1->right;
                push(&s2, current2);
                current2 = NULL;
            }
            else
            {
                printf("%d ", current2->data);
                current2 = current2->right;
                push(&s1, current1);
                current1 = NULL;
            }
        }
    }
```

Time Complexity: O(m+n)
Auxiliary Space: O(height of the first tree + height of the second tree)

129 Two nodes of a BST are swapped, correct the BST

http://www.geeksforgeeks.org/fix-two-swapped-nodes-of-bst/
Two of the nodes of a Binary Search Tree (BST) are swapped. Fix (or correct) the BST.
Input Tree:

```
     10
    / \
   5   8
  / \
 2   20
```

In the above tree, nodes 20 and 8 must be swapped to fix the tree.
Following is the output tree

```
      10
     /  \
    5   20
   / \
  2   8
```

The **inorder traversal of a BST produces a sorted array**. So a simple method is to store inorder traversal of the input tree in an auxiliary array. Sort the auxiliary array. Finally, insert the auxiilary array elements back to the BST, keeping the structure of the BST same.
Time complexity of this method is O(nLogn) and auxiliary space needed is O(n).

We can **solve** this in **O(n) time** and with a **single traversal of the given BST**. Since inorder traversal of BST is always a sorted array, the problem can be reduced to a problem where two elements of a sorted array are swapped. There are two cases that we need to handle:

1. The swapped nodes are not adjacent in the inorder traversal of the BST.
For example, Nodes 5 and 25 are swapped in {3 5 7 8 10 15 20 25}.
The inorder traversal of the given tree is 3 25 7 8 10 15 20 5

If we observe carefully, during inorder traversal, we find node 7 is smaller than the previous visited node 25. Here save the context of node 25 (previous node). Again, we find that node 5 is smaller than the previous node 20. This time, we save the context of node 5 (current node). Finally swap the two node's values.

2. The swapped nodes are adjacent in the inorder traversal of BST.
For example, Nodes 7 and 8 are swapped in {3 5 7 8 10 15 20 25}.
The inorder traversal of the given tree is 3 5 8 7 10 15 20 25

Unlike case #1, here only one point exists where a node value is smaller than previous node value. e.g. node 7 is smaller than node 8.

How to Solve?
We will maintain three pointers, first, middle and last. When we find the first point where current node value is smaller than previous node value, we update the first with the previous node & middle with the current node. When we find the second point where current node value is smaller than previous node value, we update the last with the current node.
In case #2, we will never find the second point. So, last pointer will not be updated. After processing, if the last node value is null, then two swapped nodes of BST are adjacent.
Time Complexity: O(n)

130 Construct BST from given preorder traversal

http://www.geeksforgeeks.org/construct-bst-from-given-preorder-traversa/
Given preorder traversal of a binary search tree, construct the BST.
For example, if the given traversal is {10, 5, 1, 7, 40, 50}, then the output should be root of following tree.

```
   10
  /  \
 5    40
/ \    \
1   7    50
```

Method 1 (O(n^2) time complexity)
The first element of preorder traversal is always root. We first construct the root. Then we find the index of first element which is greater than root. Let the index be 'i'. The values between root and 'i' will be part of left subtree, and the values between 'i+1' and 'n-1' will be part of right subtree. Divide given pre[] at index "i" and recur for left and right sub-trees.
For example in {10, 5, 1, 7, 40, 50}, 10 is the first element, so we make it root. Now we look for the first element greater than 10, we find 40. So we know the structure of BST is as following.

```
         10
       /   \
      /     \
 {5, 1, 7}     {40, 50}
```

We recursively follow above steps for subarrays {5, 1, 7} and {40, 50}, and get the complete tree.
Time Complexity: O(n^2)

Method 2 (O(n) time complexity)

The idea used here is inspired from method 3 of this post. The trick is to set a range {min .. max} for every node. Initialize the range as {INT_MIN .. INT_MAX}. The first node will definitely be in range, so create root node. To construct the left subtree, set the range as {INT_MIN ...root->data}. If a values is in the range {INT_MIN .. root->data}, the values is part part of left subtree. To construct the right subtree, set the range as {root->data..max .. INT_MAX}.

```
// A recursive function to construct BST from pre[]. preIndex is used
// to keep track of index in pre[].
struct node* constructTreeUtil( int pre[], int* preIndex, int key,
                                int min, int max, int size )
{
    // Base case
    if( *preIndex >= size )
        return NULL;
      struct node* root = NULL;
      // If current element of pre[] is in range, then
    // only it is part of current subtree
    if( key > min && key < max )
    {
       // Allocate memory for root of this subtree and increment *preIndex
        root = newNode ( key );
        *preIndex = *preIndex + 1;

        if (*preIndex < size)
        {
            // Contruct the subtree under root
            // All nodes which are in range {min .. key} will go in left
            // subtree, and first such node will be root of left subtree.
    root->left = constructTreeUtil(pre,preIndex,pre[*preIndex],
min,key,size);

            // All nodes which are in range {key..max} will go in right
            // subtree, and first such node will be root of right subtree.
           root->right = constructTreeUtil( pre, preIndex, pre[*preIndex],
                                        key, max, size );
        }
    }
      return root;
}
 // The main function to construct BST from given preorder traversal.
// This function mainly uses constructTreeUtil()
struct node *constructTree (int pre[], int size)
{
    int preIndex = 0;
    return constructTreeUtil ( pre, &preIndex, pre[0], INT_MIN, INT_MAX,
size );
}
```

Time Complexity: O(n)

Methode 3 (Using Stack)

Following is a stack based iterative solution that works in O(n) time.

1. Create an empty stack.
2. Make the first value as root. Push it to the stack.
3. Keep on popping while the stack is not empty and the next value is greater than stack's top value. Make this value as the right child of the last popped node. Push the new node to the stack.
4. If the next value is less than the stack's top value, make this value as the left child of the stack's top node. Push the new node to the stack.
5. Repeat steps 2 and 3 until there are items remaining in pre[].

```
Node* constructTree ( int pre[], int size )
{
    // Create a stack of capacity equal to size
    Stack* stack = createStack( size );
     // The first element of pre[] is always root
    Node* root = newNode( pre[0] );
     // Push root
    push( stack, root );
     int i;
    Node* temp;
     // Iterate through rest of the size-1 items of given preorder array
    for ( i = 1; i < size; ++i )
    {
        temp = NULL;

        /* Keep on popping while the next value is greater than
           stack's top value. */
        while ( !isEmpty( stack ) && pre[i] > peek( stack )->data )
            temp = pop( stack );
    // Make this greater value as the right child and push it to the stack
        if ( temp != NULL)
        {
            temp->right = newNode( pre[i] );
            push( stack, temp->right );
        }
  // If the next value is less than the stack's top value, make this value
  // as the left child of the stack's top node. Push the new node to stack
        else
        {
            peek( stack )->left = newNode( pre[i] );
            push( stack, peek( stack )->left );
        }
    }

    return root;
}
```

Time Complexity: O(n). The complexity looks more from first look. If we take a closer look, we can observe that every item is pushed and popped only once. So at most 2n push/pop operations are performed in the main loops of constructTree(). Therefore, time complexity is O(n).

131 Floor and Ceil from a BST

http://www.geeksforgeeks.org/floor-and-ceil-from-a-bst/

There are numerous applications we need to find floor (ceil) value of a key in a binary search tree or sorted array.

For example, consider designing memory management system in which free nodes are arranged in BST. Find best fit for the input request.

Ceil Value Node: Node with smallest data larger than or equal to key value.

Imagine we are moving down the tree, and assume we are root node. The comparison yields three possibilities,

A) Root data is equal to key. We are done, root data is ceil value.

B) Root data < key value, certainly the ceil value can't be in left subtree. Proceed to search on right subtree as reduced problem instance.

C) Root data > key value, the ceil value may be in left subtree. We may find a node with is larger data than key value in left subtree, if not the root itself will be ceil node.

Here is the code for ceil value.

```
// Function to find ceil of a given input in BST. If input is more
// than the max key in BST, return -1
int Ceil(node *root, int input)
{
    // Base case
    if( root == NULL )
        return -1;

    // We found equal key
    if( root->key == input )
        return root->key;

    // If root's key is smaller, ceil must be in right subtree
    if( root->key < input )
        return Ceil(root->right, input);

    // Else, either left subtree or root has the ceil value
    int ceil = Ceil(root->left, input);
    return (ceil >= input) ? ceil : root->key;
}
```

Exercise:

1. Modify above code to find floor value of input key in a binary search tree.
2. Write neat algorithm to find floor and ceil values in a sorted array. Ensure to handle all possible boundary conditions.

132 Convert a BST to a Binary Tree such that sum of all greater keys is added to every key

http://www.geeksforgeeks.org/convert-bst-to-a-binary-tree/

Given a Binary Search Tree (BST), convert it to a Binary Tree such that every key of the original BST is changed to key plus sum of all greater keys in BST.

Examples:

Input: Root of following BST

```
     5
    /  \
   2    13
```

Output: The given BST is converted to following Binary Tree

```
     18
    /  \
   20   13
```

Source: Convert a BST

Solution: Do reverse Inoorder traversal. Keep track of the sum of nodes visited so far. Let this sum be sum. For every node currently being visited, first add the key of this node to sum, i.e. sum = sum + node->key. Then change the key of current node to sum, i.e., node->key = sum.

When a BST is being traversed in reverse Inorder, for every key currently being visited, all keys that are already visited are all greater keys.

Time Complexity: O(n) where n is the number of nodes in given Binary Search Tree.

133 Sorted Linked List to Balanced BST

http://www.geeksforgeeks.org/sorted-linked-list-to-balanced-bst/

Given a Singly Linked List which has data members sorted in ascending order. Construct a Balanced Binary Search Tree which has same data members as the given Linked List.

Examples:

Input: Linked List 1->2->3

Output: A Balanced BST

```
   2
  / \
 1   3
```

Input: Linked List 1->2->3->4->5->6->7

Output: A Balanced BST

```
     4
    /  \
   2    6
  / \  / \
 1  3 4  7
```

Input: Linked List 1->2->3->4

Output: A Balanced BST

```
    3
   / \
  2   4
 /
1
```

Input: Linked List 1->2->3->4->5->6

Output: A Balanced BST

```
    4
   /  \
  2    6
 / \  /
1  3 5
```

Method 1 (Simple)

Following is a simple algorithm where we first find the middle node of list and make it root of the tree to be constructed.

1) Get the Middle of the linked list and make it root.
2) Recursively do same for left half and right half.
   a) Get the middle of left half and make it left child of the root created in step 1.
   b) Get the middle of right half and make it right child of the root created in step 1.

Time complexity: O(nLogn) where n is the number of nodes in Linked List.
See this forum thread for more details.

Method 2 (Tricky)

The method 1 constructs the tree from root to leaves. In this method, we construct from leaves to root. The idea is to insert nodes in BST in the same order as the appear in Linked List, so that the tree can be constructed in **O(n) time complexity**.

We **first count the number of nodes in the given Linked List**. Let the count be n. After counting nodes, **we take left n/2 nodes and recursively construct the left subtree**. After left subtree is constructed, we allocate memory for root and link the left subtree with root. Finally, we recursively construct the right subtree and link it with root.

While constructing the BST, we also keep moving the list head pointer to next so that we have the appropriate pointer in each recursive call.

```
/* This function counts the number of nodes in Linked List and then calls
   sortedListToBSTRecur() to construct BST */
struct TNode* sortedListToBST(struct LNode *head)
{
    /*Count the number of nodes in Linked List */
    int n = countLNodes(head);

    /* Construct BST */
    return sortedListToBSTRecur(&head, n);
}
```

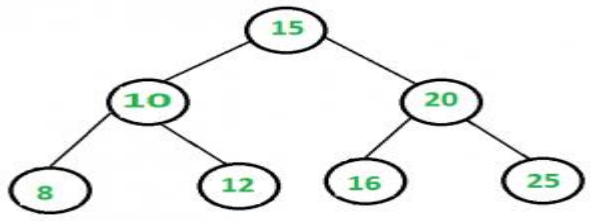
```
/* The main function that constructs balanced BST and returns root of it.
       head_ref -->  Pointer to pointer to head node of linked list
       n  --> No. of nodes in Linked List */
struct TNode* sortedListToBSTRecur(struct LNode **head_ref, int n)
{
    /* Base Case */
    if (n <= 0)
        return NULL;

    /* Recursively construct the left subtree */
    struct TNode *left = sortedListToBSTRecur(head_ref, n/2);

    /* Allocate memory for root, and link the above constructed left
       subtree with root */
    struct TNode *root = newNode((*head_ref)->data);
    root->left = left;

    /* Change head pointer of Linked List for parent recursive calls */
    *head_ref = (*head_ref)->next;

    /* Recursively construct the right subtree and link it with root
      The number of nodes in right subtree  is total nodes - nodes in
      left subtree - 1 (for root) which is n-n/2-1*/
    root->right = sortedListToBSTRecur(head_ref, n-n/2-1);

    return root;
}
```

Time Complexity: O(n)

134 Find a pair with given sum in a Balanced BST

http://www.geeksforgeeks.org/find-a-pair-with-given-sum-in-bst/

Given a Balanced Binary Search Tree and a target sum, write a function that returns true if there is a pair with sum equals to target sum, otherwise return false. Expected time complexity is O(n) and only O(Logn) extra space can be used. Any modification to Binary Search Tree is not allowed. Note that height of a Balanced BST is always O(Logn).



This problem is mainly extension of the previous post. Here we are not allowed to modify the BST.

The **Brute Force Solution** is to consider each pair in BST and check whether the sum equals to X.
The **time complexity** of this solution will be O(n^2).

A **Better Solution** is to create an auxiliary array and store Inorder traversal of BST in the array. The array will be sorted as Inorder traversal of BST always produces sorted data. Once we have the Inorder traversal, we can pair in O(n) time (See this for details). This solution works in **O(n) time**, but requires **O(n) auxiliary space**.

A **space optimized solution** is discussed in previous post. The idea was to first in-place convert BST to Doubly Linked List (DLL), then find pair in sorted DLL in O(n) time. This solution takes O(n) time and O(Logn) extra space, but it modifies the given BST.
The solution discussed below takes **O(n) time**, **O(Logn) space** and doesn't modify BST.

The **idea** is **same as finding the pair in sorted array** (See method 1 of this for details). We traverse BST in Normal Inorder and Reverse Inorder simultaneously. In reverse inorder, we start from the rightmost node which is the maximum value node.

In normal inorder, we start from the left most node which is minimum value node. We add sum of current nodes in both traversals and compare this sum with given target sum.
If the sum is same as target sum, we return true.

If the sum is more than target sum, we move to next node in reverse inorder traversal, otherwise we move to next node in normal inorder traversal. If any of the traversals is finished without finding a pair, we return false.

```
// Returns true if a pair with target sum exists in BST, otherwise false
bool isPairPresent(struct node *root, int target)
{
    // Create two stacks. s1 is used for normal inorder traversal
    // and s2 is used for reverse inorder traversal
    struct Stack* s1 = createStack(MAX_SIZE);
    struct Stack* s2 = createStack(MAX_SIZE);

    // Note the sizes of stacks is MAX_SIZE, we can find the tree size and
    // fix stack size as O(Logn) for balanced trees like AVL and Red Black
    // tree. We have used MAX_SIZE to keep the code simple

   // done1, val1 and curr1 are used for normal inorder traversal using s1
  // done2, val2 and curr2 are used for reverse inorder traversal using s2
    bool done1 = false, done2 = false;
    int val1 = 0, val2 = 0;
    struct node *curr1 = root, *curr2 = root;

    // The loop will break when we either find a pair or one of the two
    // traversals is complete
    while (1)
    {
        while (done1 == false)
        {
            if (curr1 != NULL)
            {
                push(s1, curr1);
                curr1 = curr1->left;
            }
            else
            {
                if (isEmpty(s1))
                    done1 = 1;
                else
                {
                    curr1 = pop(s1);
                    val1 = curr1->val;
                    curr1 = curr1->right;
                    done1 = 1;
                }
            }
        }

        // Find next node in REVERSE Inorder traversal. The only
        // difference between above and below loop is, in below loop
        // right subtree is traversed before left subtree
        while (done2 == false)
        {
            if (curr2 != NULL)
            {
                push(s2, curr2);
                curr2 = curr2->right;
            }
```

```
    else
                {
                    if (isEmpty(s2))
                        done2 = 1;
                    else
                    {
                        curr2 = pop(s2);
                        val2 = curr2->val;
                        curr2 = curr2->left;
                        done2 = 1;
                    }
                }
            }

            // If we find a pair, then print the pair and return. The first
            // condition makes sure that two same values are not added
            if ((val1 != val2) && (val1 + val2) == target)
            {
                printf("\n Pair Found: %d + %d = %d\n", val1, val2, target);
                return true;
            }

            // If sum of current values is smaller, then move to next node in
            // normal inorder traversal
            else if ((val1 + val2) < target)
                done1 = false;

            // If sum of current values is greater, then move to next node in
            // reverse inorder traversal
            else if ((val1 + val2) > target)
                done2 = false;

            // If any of the inorder traversals is over, then there is no pair
            // so return false
            if (val1 >= val2)
                return false;
        }

}
```

135 Total number of possible Binary Search Trees with n keys

http://www.geeksforgeeks.org/g-fact-18/

Total number of possible Binary Search Trees with n different keys = Catalan number Cn = (2n)!/(n+1)!*n!

For n = 0, 1, 2, 3, … values of Catalan numbers are 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, …. So are numbers of Binary Search Trees.

Here is a systematic way to enumerate these BSTs. Consider all possible binary search trees with each element at the root**. If there are n nodes, then for each choice of root node**, there are n – 1 non-root nodes and these non-root nodes must be partitioned into those that are less than a chosen root and those that are greater than the chosen root.

Let's say node i is chosen to be the root. Then there are i – 1 nodes smaller than i and n – i nodes bigger than i. For each of these two sets of nodes, there is a certain number of possible subtrees.

Let t(n) be the total number of BSTs with n nodes. The total number of BSTs with i at the root is t(i – 1) t(n – i). The two terms are multiplied together because the arrangements in the left and right subtrees are independent. That is, for each arrangement in the left tree and for each arrangement in the right tree, you get one BST with i at the root.

Summing over i gives the total number of binary search trees with n nodes.

$$t(n) = \sum_{i=1}^{n} t(i-1)\ t(n-i).$$

The base case is t(0) = 1 and t(1) = 1, i.e. there is one empty BST and there is one BST with one node.

136 Binary Tree to Binary Search Tree Conversion

http://www.geeksforgeeks.org/binary-tree-to-binary-search-tree-conversion/
Given a Binary Tree, convert it to a Binary Search Tree. The conversion must be done in such a way that keeps the original structure of Binary Tree.
Examples.
Example 1
Input:

```
      10
     /  \
    2    7
   / \
  8   4
```

Output:

```
      8
     / \
    4   10
   / \
  2   7
```

Solution

Following is a 3 step solution for converting Binary tree to Binary Search Tree.
1) Create a temp array arr[] that stores inorder traversal of the tree.
This step takes O(n) time.

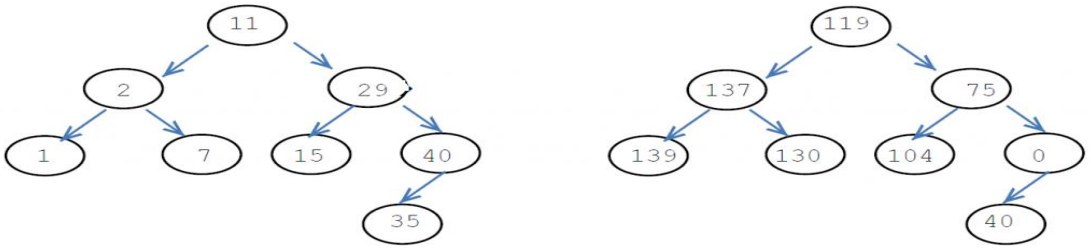
2) Sort the temp array arr[]. Time complexity of this step depends upon the sorting algorithm. In the following implementation, Quick Sort is used which takes (n^2) time. This can be done in O(nLogn) time using Heap Sort or Merge Sort.

3) Again do inorder traversal of tree and copy array elements to tree nodes one by one.
This step takes O(n) time.

137 Transform a BST to greater sum tree

http://www.geeksforgeeks.org/transform-bst-sum-tree/

Given a BST, transform it into greater sum tree where each node contains sum of all nodes greater than that node.



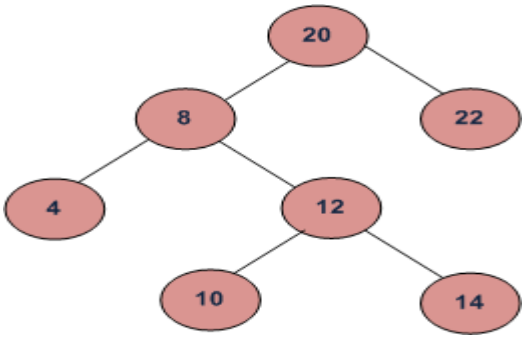
Method 1 (Naive):

This method doesn't require the tree to be a BST. Following are steps.
1. Traverse node by node(Inorder, preorder, etc.)
2. For each node find all the nodes greater than that of the current node, sum the values. Store all these sums.
3. Replace each node value with their corresponding sum by traversing in the same order as in Step 1.
This takes O(n^2) **Time Complexity**.

Method 2 (Using only one traversal)
By leveraging the fact that the tree is a BST, we can find an O(n) solution. The idea is to traverse BST in reverse inorder. Reverse inorder traversal of a BST gives us keys in decreasing order. Before visiting a node, we visit all greater nodes of that node. While traversing we keep track of sum of keys which is the sum of all the keys greater than the key of current node.
Time complexity of this method is O(n) as it does a simple traversal of tree.

138 K'th Largest Element in BST when modification to BST is not allowed

http://www.geeksforgeeks.org/kth-largest-element-in-bst-when-modification-to-bst-is-not-allowed/
Given a Binary Search Tree (BST) and a positive integer k, find the k'th largest element in the Binary Search Tree.
For example, in the following BST, if k = 3, then output should be 14, and if k = 5, then output should be 10.



We have discussed two methods in this post. The method 1 requires O(n) time. The method 2 takes O(h) time where h is height of BST, but requires augmenting the BST (storing count of nodes in left subtree with every node).

Can we find k'th largest element in better than O(n) time and no augmentation?
In this post, a method is discussed that takes O(h + k) time. This method doesn't require any change to BST.

The **idea** is to **do reverse inorder traversal of BST**. The reverse inorder traversal traverses all nodes in decreasing order. While doing the traversal, we keep track of count of nodes visited so far. When the count becomes equal to k, we stop the traversal and print the key.

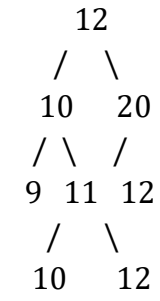
```
// A function to find k'th largest element in a given tree.
void kthLargestUtil(Node *root, int k, int &c)
{
    // Base cases, the second condition is important to
    // avoid unnecessary recursive calls
    if (root == NULL || c >= k)
        return;

    // Follow reverse inorder traversal so that the
    // largest element is visited first
    kthLargestUtil(root->right, k, c);

    // Increment count of visited nodes
    c++;

    // If c becomes k now, then this is the k'th largest
    if (c == k)
    {
        cout << "K'th largest element is "
             << root->key << endl;
        return;
    }

    // Recur for left subtree
    kthLargestUtil(root->left, k, c);
}

// Function to find k'th largest element
void kthLargest(Node *root, int k)
{
    // Initialize count of nodes visited as 0
    int c = 0;

    // Note that c is passed by reference
    kthLargestUtil(root, k, c);
}
```

Time complexity: The code first traverses down to the rightmost node which takes O(h) time, then traverses k elements in O(k) time.
Therefore overall time complexity is O(h + k).

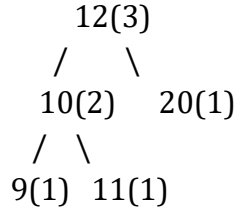
139 How to handle duplicates in Binary Search Tree?

http://www.geeksforgeeks.org/how-to-handle-duplicates-in-binary-search-tree/
In a Binary Search Tree (BST), all keys in left subtree of a key must be smaller and all keys in right subtree must be greater. So a Binary Search Tree by definition has distinct keys.
How to allow duplicates where every insertion inserts one more key with a value and every deletion deletes one occurrence?

A **Simple Solution** is to allow same keys on right side (we could also choose left side). For example consider insertion of keys 12, 10, 20, 9, 11, 10, 12, 12 in an empty Binary Search Tree



A **Better Solution** is to **augment every tree node** to store count together with regular fields like key, left and right pointers.
Insertion of keys 12, 10, 20, 9, 11, 10, 12, 12 in an empty Binary Search Tree would create following.



Count of a key is shown in bracket

This approach has following advantages over above simple approach.
1) Height of tree is small irrespective of number of duplicates. Note that most of the BST operations (search, insert and delete) have time complexity as O(h) where h is height of BST. So if we are able to keep the height small, we get advantage of less number of key comparisons.

2) Search, Insert and Delete become easier to do. We can use same insert, search and delete algorithms with small modifications (see below code).

3) This approach is suited for self-balancing BSTs (AVL Tree, Red-Black Tree, etc) also. These trees involve rotations, and a rotation may violate BST property of simple solution as a same key can be in either left side or right side after rotation.
Below is C implementation of normal Binary Search Tree with count with every key. This code basically is taken from code for insert and delete in BST. The changes made for handling duplicates are highlighted, rest of the code is same.
Insert

```
// If key already exists in BST, icnrement count and return
    if (key == node->key)
    {
       (node->count)++;
        return node;
    }
```

Delete

```
// If key is present more than once, simply decrement
        // count and return
        if (root->count > 1)
        {
           (root->count)--;
           return root;
        }
```

140 Print Common Nodes in Two Binary Search Trees

http://www.geeksforgeeks.org/print-common-nodes-in-two-binary-search-trees/

Given two Binary Search Trees, find common nodes in them. In other words, find intersection of two BSTs.

Example:

```
Input: root1  (Root of below tree)
           5
         /   \
        1     10
       / \    /
      0   4  7
              \
               9

      root2 (Root of below tree)
          10
        /   \
       7     20
      / \
     4   9

Output: 4 7 9 10
```

Method 1 (Simple Solution)

A simple way is to one by once search every node of first tree in second tree. **Time complexity** of this solution is O(m * h) where m is number of nodes in first tree and h is height of second tree.

Method 2 (Linear Time)

We can find common elements in O(n) time.
1) Do inorder traversal of first tree and store the traversal in an auxiliary array ar1[]. See sortedInorder() here.
2) Do inorder traversal of second tree and store the traversal in an auxiliary array ar2[]
3) Find intersection of ar1[] and ar2[]. See this for details.

Time complexity of this method is O(m+n) where m and n are number of nodes in first and second tree respectively. This solution requires O(m+n) extra space.

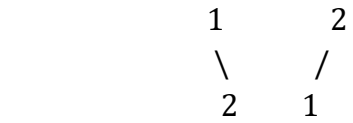
Method 3 (Linear Time and limited Extra Space)

We can find common elements in O(n) time and O(h1 + h2) extra space where h1 and h2 are heights of first and second BSTs respectively.
The **idea** is to **use iterative inorder traversal**. We use two auxiliary stacks for two BSTs. Since we need to find common elements, whenever we get same element, we print it.

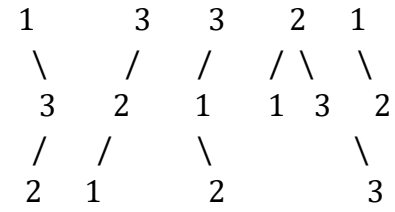
141 Construct all possible BSTs for keys 1 to N

http://www.geeksforgeeks.org/construct-all-possible-bsts-for-keys-1-to-n/
In this article, first count of possible BST (Binary Search Trees)s is discussed, then construction of all possible BSTs.

How many structurally unique BSTs for keys from 1..N?
For example, for N = 2, there are 2 unique BSTs



For N = 3, there are 5 possible BSTs



We know that all node in left subtree are smaller than root and in right subtree are larger than root so if we have ith number as root, all numbers from 1 to i-1 will be in left subtree and i+1 to N will be in right subtree. If 1 to i-1 can form x different trees

and i+1 to N can from y different trees then we will have x*y total trees when ith number is root and we also have N choices for root also so we can simply iterate from 1 to N for root and another loop for left and right subtree. If we take a closer look, we can notice that the count is basically n'th Catalan number. We have discussed different approaches to find n'th Catalan number here.

How to construct all BST for keys 1..N?

The idea is to maintain a list of roots of all BSTs. Recursively construct all possible left and right subtrees. Create a tree for every pair of left and right subtree and add the tree to list. Below is detailed algorithm.

1) Initialize list of BSTs as empty.
2) For every number i where i varies from 1 to N, do following
.....a) Create a new node with key as 'i', let this node be 'node'
.....b) Recursively construct list of all left subtrees.
.....c) Recursively construct list of all right subtrees.
3) Iterate for all left subtrees
  a) For current leftsubtree, iterate for all right subtrees
    Add current left and right subtrees to 'node' and add
    'node' to list.

```
//  function for constructing trees
vector<struct node *> constructTrees(int start, int end)
{
    vector<struct node *> list;

    /*  if start > end   then subtree will be empty so returning NULL
        in the list */
    if (start > end)
    {
        list.push_back(NULL);
        return list;
    }

    /*  iterating through all values from start to end  for constructing\
        left and right subtree recursively  */
    for (int i = start; i <= end; i++)
    {
        /*  constructing left subtree   */
        vector<struct node *> leftSubtree  = constructTrees(start, i - 1);

        /*  constructing right subtree  */
        vector<struct node *> rightSubtree = constructTrees(i + 1, end);
```

```
 /*  now looping through all left and right subtrees and connecting
            them to ith root  below  */
        for (int j = 0; j < leftSubtree.size(); j++)
        {
            struct node* left = leftSubtree[j];
            for (int k = 0; k < rightSubtree.size(); k++)
            {
                struct node * right = rightSubtree[k];
                struct node * node = newNode(i);// making value i as root
                node->left = left;              // connect left subtree
                node->right = right;            // connect right subtree
                list.push_back(node);           // add this tree to list
            }
        }
    }
    return list;
}
```

142 Count BST subtrees that lie in given range

http://www.geeksforgeeks.org/count-bst-subtrees-that-lie-in-given-range/

Given a Binary Search Tree (BST) of integer values and a range [low, high], return count of nodes where all the nodes under that node (or subtree rooted with that node) lie in the given range.

Examples:

Input:

```
     10
    /  \
   5    50
  /    / \
 1    40  100
```

Range: [5, 45]

Output: 1

There is only 1 node whose subtree is in the given range.

The node is 40

Input:

```
     10
    /  \
   5    50
  /    / \
 1    40  100
```

Range: [1, 45]

Output: 3
There are three nodes whose subtree is in the given range.
The nodes are 1, 5 and 40

The **idea** is to **traverse the given Binary Search Tree (BST) in bottom up manner**. For every node, recur for its subtrees, if subtrees are in range and the nodes is also in range, then increment count and return true (to tell the parent about its status). Count is passed as a pointer so that it can be incremented across all function calls.

```
// A recursive function to get count of nodes whose subtree
// is in range from low to hgih.  This function returns true
// if nodes in subtree rooted under 'root' are in range.
bool getCountUtil(node *root, int low, int high, int *count)
{
    // Base case
    if (root == NULL)
       return true;

    // Recur for left and right subtrees
    bool l = (root->left) ?  getCountUtil(root->left, low, high, count) :
true;
    bool r = (root->right) ? getCountUtil(root->right, low, high, count) :
true;


    // If both left and right subtrees are in range and current node
    // is also in range, then increment count and return true
    if (l && r && inRange(root, low, high))
    {
       ++*count;
       return true;
    }

    return false;
}
```

143 Count BST nodes that lie in a given range

http://www.geeksforgeeks.org/count-bst-nodes-that-are-in-a-given-range/
Given a Binary Search Tree (BST) and a range, count number of nodes that lie in the given range.
Examples:

Input:

```
     10
   /   \
  5     50
 /     /  \
1     40   100
```

Range: [5, 45]

Output: 3
There are three nodes in range, 5, 10 and 40

The **idea** is to **traverse the given binary search tree starting from root.** For every node being visited, check if this node lies in range, if yes, then add 1 to result and recur for both of its children. If current node is smaller than low value of range, then recur for right child, else recur for left child.

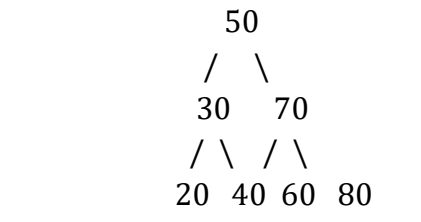
```
// Returns count of nodes in BST in range [low, high]
int getCount(node *root, int low, int high)
{
    // Base case
    if (!root) return 0;

    // Special Optional case for improving efficiency
    if (root->data == high && root->data == low)
        return 1;

    // If current node is in range, then include it in count and
    // recur for left and right children of it
    if (root->data <= high && root->data >= low)
         return 1 + getCount(root->left, low, high) +
                    getCount(root->right, low, high);

    // If current node is smaller than low, then recur for right
    // child
    else if (root->data < low)
         return getCount(root->right, low, high);

    // Else recur for left child
    else return getCount(root->left, low, high);
}
```

Time complexity of the above program is O(h + k) where h is height of BST and k is number of nodes in given range.

144 How to implement decrease key or change key in Binary Search Tree?

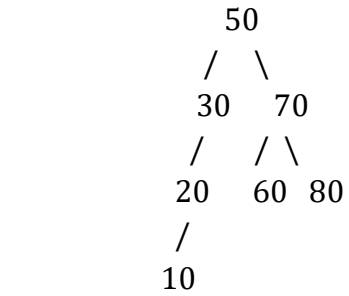
http://quiz.geeksforgeeks.org/how-to-implement-decrease-key-or-change-key-in-binary-search-tree/

Given a Binary Search Tree, write a function that takes following three as arguments:
1) Root of tree
2) Old key value
3) New Key Value
The function should change old key value to new key value. The function may assume that old key value always exists in Binary Search Tree.
Example:
Input: Root of below tree



Old key value: 40
New key value: 10

Output: BST should be modified to following



The **idea** is to **call delete for old key value**, then **call insert for new key value**.
Time complexity of above changeKey() is O(h) where h is height of BST.

145 Count inversions in an array | Set 2 (Using Self-Balancing BST)

http://www.geeksforgeeks.org/count-inversions-in-an-array-set-2-using-self-balancing-bst/

Inversion Count for an array indicates – how far (or close) the array is from being sorted. If array is already sorted then inversion count is 0. If array is sorted in reverse order that inversion count is the maximum.

Two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j. For simplicity, we may assume that all elements are unique.

Example:
Input: arr[] = {8, 4, 2, 1}
Output: 6
Given array has six inversions (8,4), (4,2), (8,2), (8,1), (4,1), (2,1).

Reference: GeeksForGeeks