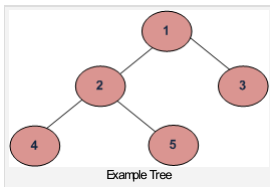


Tree

1. Tree Traversals

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



Depth First Traversals:

- (a) Inorder
- (b) Preorder
- (c) Postorder

Breadth First or Level Order Traversal

Please see [this](#) post for Breadth First Traversal.

Inorder Traversal:

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Uses of Inorder

In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal is reversed, can be used.

Example: Inorder traversal for the above given figure is 4 2 5 1 3.

Preorder Traversal:

Algorithm Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Uses of Preorder

Preorder traversal is used to create a copy of the tree. Preorder traversal is also used

to get prefix expression on of an expression tree. Please see http://en.wikipedia.org/wiki/Polish_notation to know why prefix expressions are useful. Example: Preorder traversal for the above given figure is 1 2 4 5 3.

Postorder Traversal:

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

Uses of Postorder

Postorder traversal is used to delete the tree. Please see [the question for deletion of tree](#) for details. Postorder traversal is also useful to get the postfix expression of an expression tree. Please see http://en.wikipedia.org/wiki/Reverse_Polish_notation to for the usage of postfix expression.

Example: Postorder traversal for the above given figure is 4 5 2 3 1.

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Given a binary tree, print its nodes according to the
   "bottom-up" postorder traversal. */
void printPostorder(struct node* node)
{
    if (node == NULL)
        return;

    // first recur on left subtree
    printPostorder(node->left);

    // then recur on right subtree
    printPostorder(node->right);
```

```

        // now deal with the node
        printf("%d ", node->data);
    }

    /* Given a binary tree, print its nodes in inorder*/
    void printInorder(struct node* node)
    {
        if (node == NULL)
            return;

        /* first recur on left child */
        printInorder(node->left);

        /* then print the data of node */
        printf("%d ", node->data);

        /* now recur on right child */
        printInorder(node->right);
    }

    /* Given a binary tree, print its nodes in preorder*/
    void printPreorder(struct node* node)
    {
        if (node == NULL)
            return;

        /* first print data of node */
        printf("%d ", node->data);

        /* then recur on left subtree */
        printPreorder(node->left);

        /* now recur on right subtree */
        printPreorder(node->right);
    }

    /* Driver program to test above functions*/
    int main()
    {
        struct node *root = newNode(1);
        root->left          = newNode(2);
        root->right          = newNode(3);
        root->left->left      = newNode(4);
        root->left->right     = newNode(5);

        printf("\n Preorder traversal of binary tree is \n");
        printPreorder(root);

        printf("\n Inorder traversal of binary tree is \n");
        printInorder(root);

        printf("\n Postorder traversal of binary tree is \n");
        printPostorder(root);

        getchar();
        return 0;
    }

```

Time Complexity: $O(n)$

Let us prove it:

Complexity function $T(n)$ — for all problem where tree traversal is involved — can be defined as:

$$T(n) = T(k) + T(n - k - 1) + c$$

Where k is the number of nodes on one side of root and $n-k-1$ on the other side.

Let's do analysis of boundary conditions

Case 1: Skewed tree (One of the subtrees is empty and other subtree is non-empty)

k is 0 in this case.

$$T(n) = T(0) + T(n-1) + c$$

$$T(n) = 2T(0) + T(n-2) + 2c$$

$$T(n) = 3T(0) + T(n-3) + 3c$$

$$T(n) = 4T(0) + T(n-4) + 4c$$

.....

$$T(n) = (n-1)T(0) + T(1) + (n-1)c$$

$$T(n) = nT(0) + (n)c$$

Value of $T(0)$ will be some constant say d . (traversing a empty tree will take some constants time)

$$T(n) = n(c+d)$$

$$T(n) = O(n) \text{ (Theta of } n)$$

Case 2: Both left and right subtrees have equal number of nodes.

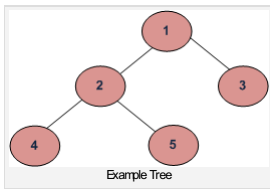
$$T(n) = 2T(\lfloor n/2 \rfloor) + c$$

This recursive function is in the standard form ($T(n) = aT(n/b) + O(n)$) for master method http://en.wikipedia.org/wiki/Master_theorem. If we solve it by master method we get $O(n \log n)$

Auxiliary Space : If we don't consider size of stack for function calls then $O(1)$ otherwise $O(n)$.

2. Write a C program to Calculate Size of a tree

Size of a tree is the number of elements present in the tree. Size of the below tree is 5.



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

```

#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Computes the number of nodes in a tree. */
int size(struct node* node)
{
    if (node==NULL)
        return 0;
    else
        return(size(node->left) + 1 + size(node->right));
}

/* Driver program to test size function*/
int main()
{
    struct node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Size of the tree is %d", size(root));
    getchar();
    return 0;
}

```

Time & Space Complexities: Since this program is similar to traversal of tree, time and space complexities will be same as Tree traversal (Please see our [Tree Traversal](#) post for details)

3. 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.

Algorithm:

```
sameTree(tree1, tree2)
```

1. If both trees are empty then return 1.
2. Else If both trees are non -empty
 - (a) Check data of the root nodes (tree1->data == tree2->data)
 - (b) Check left subtrees recursively i.e., call sameTree(tree1->left_subtree, tree2->left_subtree)
 - (c) Check right subtrees recursively i.e., call sameTree(tree1->right_subtree, tree2->right_subtree)
 - (d) If a,b and c are true then return 1.
- 3 Else return 0 (one is empty and other is not)

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* 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;
}

/* Driver program to test identicalTrees function*/
int main()
{
    struct node *root1 = newNode(1);
    struct node *root2 = newNode(1);
    root1->left = newNode(2);
    root1->right = newNode(3);
    root1->left->left = newNode(4);
    root1->left->right = newNode(5);

    root2->left = newNode(2);
    root2->right = newNode(3);
    root2->left->left = newNode(4);
    root2->left->right = newNode(5);

    if(identicalTrees(root1, root2))
        printf("Both tree are identical.");
    else
        printf("Trees are not identical.");

    getchar();
    return 0;
}

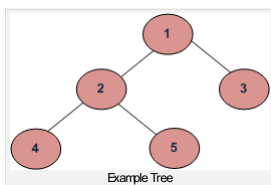
```

Time Complexity:

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 $m < n$.

4. Write a C Program to Find the Maximum Depth or Height of a Tree

Maximum depth or height of the below tree is 3.



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.

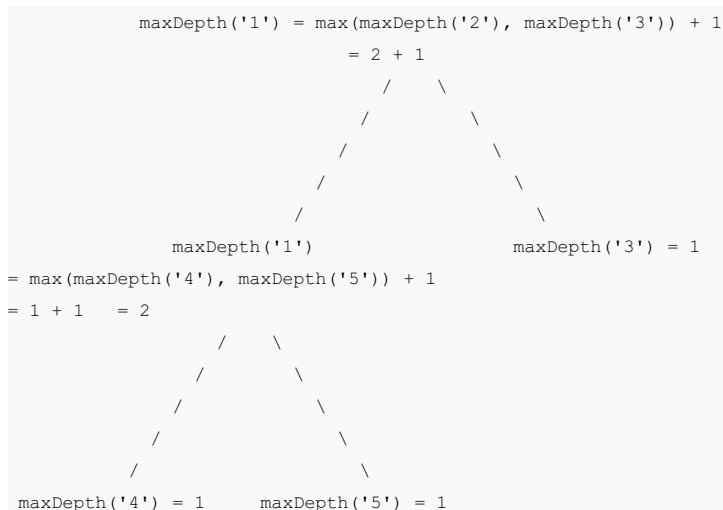
Algorithm:

```

maxDepth()
1. If tree is empty then return 0
2. Else
    (a) Get the max depth of left subtree recursively i.e.,
        call maxDepth( tree->left-subtree)
    (a) Get the max depth of right subtree recursively i.e.,
        call maxDepth( tree->right-subtree)
    (c) Get the max of max depths of left and right
        subtrees and add 1 to it for the current node.
        max_depth = max(max dept of left subtree,
                        max depth of right subtree)
                    + 1
    (d) Return max_depth

```

See the below diagram for more clarity about execution of the recursive function **maxDepth()** for above example tree.



Implementation:

```

#include<stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Compute the "maxDepth" of a tree -- the number of
   nodes along the longest path from the root node
   down to the farthest leaf node.*/
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);
    }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

int main()
{
    struct node *root = newNode(1);

    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Hight of tree is %d", maxDepth(root));

    getchar();
    return 0;
}

```

Time Complexity: $O(n)$ (Please see our post [Tree Traversal](#) for details)

References:

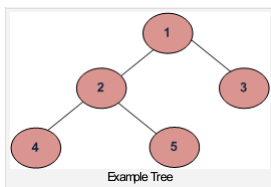
<http://cslibrary.stanford.edu/110/BinaryTrees.html>

5. 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.

For the following tree nodes are deleted in order – 4, 5, 2, 3, 1



Program

```

#include<stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));

    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

/* 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);
}

/* Driver program to test deleteTree function*/
int main()
{
    struct node *root = newNode(1);
    root->left          = newNode(2);
    root->right          = newNode(3);
    root->left->left      = newNode(4);
    root->left->right     = newNode(5);

    deleteTree(root);
    root = NULL;

    printf("\n Tree deleted ");

    getchar();
    return 0;
}

```

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.

```

#include<stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));

    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

/* 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;
}

/* Driver program to test deleteTree function*/
int main()
{
    struct node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    // Note that we pass the address of root here
    deleteTree(&root);
    printf("\n Tree deleted ");

    getchar();
    return 0;
}

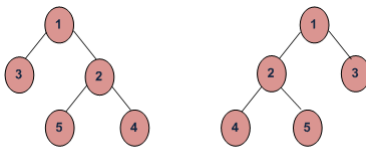
```

Time Complexity: $O(n)$

Space Complexity: If we don't consider size of stack for function calls then $O(1)$ otherwise $O(n)$

6. Write an Efficient C Function to Convert a Binary 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

Trees in the below figure are mirror of each other

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
```

Program:

```
#include<stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
```

```

        malloc(sizeof(struct node)),
        node->data = data;
        node->left = NULL;
        node->right = NULL;

    return(node);
}

```

/* Change a tree so that the roles of the left and right pointers are swapped at every node.

So the tree...



is changed to...



```

*/
void mirror(struct node* node)
{
    if (node==NULL)
        return;
    else
    {
        struct node* temp;

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

        /* swap the pointers in this node */
        temp      = node->left;
        node->left = node->right;
        node->right = temp;
    }
}

/* Helper function to test mirror(). Given a binary
   search tree, print out its data elements in
   increasing sorted order.*/
void inOrder(struct node* node)
{
    if (node == NULL)
        return;

    inOrder(node->left);
    printf("%d ", node->data);

    inOrder(node->right);
}

/* Driver program to test mirror() */
int main()

```



```

{
    struct node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    /* Print inorder traversal of the input tree */
    printf("\n Inorder traversal of the constructed tree is \n");
    inOrder(root);

    /* Convert tree to its mirror */
    mirror(root);

    /* Print inorder traversal of the mirror tree */
    printf("\n Inorder traversal of the mirror tree is \n");
    inOrder(root);

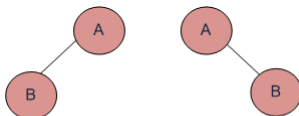
    getch();
    return 0;
}

```

Time & Space Complexities: This program is similar to traversal of tree space and time complexities will be same as Tree traversal (Please see our [Tree Traversal](#) post for details)

7. 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.

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

Asked by Varun Bhatia

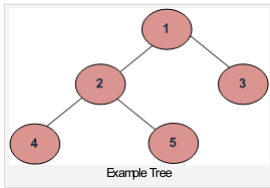
Here is the solution.

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)
```

Example:



Output for the above example will be

```

1 2 4
1 2 5
1 3

```

Implementation:

```

/*program to print all of its root-to-leaf paths for a tree*/
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

void printArray(int [], int);
void printPathsRecur(struct node*, int [], int);
struct node* newNode(int );
void printPaths(struct node*);

/* Given a binary tree, print out all of its root-to-leaf
   paths, one per line. Uses a recursive helper to do the work.*/
void printPaths(struct node* node)
{
    int path[1000];
    printPathsRecur(node, path, 0);
}

/* Recursive helper function -- given a node, and an array containing
   the path from the root node up to but not including this node,
   print out all the root-leaf paths. */
void printPathsRecur(struct node* node, int path[], int pathLen)
{
    if (node==NULL) return;

    /* append this node to the path array */
    path[pathLen] = node->data;
    pathLen++;

    /* it's a leaf, so print the path that led to here */
    if (node->left==NULL && node->right==NULL)
    {
        printArray(path, pathLen);
    }
    else
    {

```

```

    /* otherwise try both subtrees */
    printPathsRecur(node->left, path, pathLen);
    printPathsRecur(node->right, path, pathLen);
}
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Utility that prints out an array on a line */
void printArray(int ints[], int len)
{
    int i;
    for (i=0; i<len; i++) {
        printf("%d ", ints[i]);
    }
    printf("\n");
}

/* Driver program to test mirror() */
int main()
{
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    /* Print all root-to-leaf paths of the input tree */
    printPaths(root);

    getchar();
    return 0;
}

```

References:

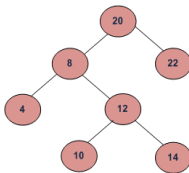
<http://cslibrary.stanford.edu/110/BinaryTrees.html>

9. 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., $n_1 < n < n_2$ or same as one of the n1 or n2, is LCA of n1 and n2 (assuming that $n_1 < n_2$). 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 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)

// A recursive C program to find LCA of two nodes n1 and n2.

```
#include <stdio.h>
#include <stdlib.h>
```

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

```
/* 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;
}

/* Helper function that allocates a new node with the given data.*/
struct node* newNode(int data)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = node->right = NULL;
    return(node);
}

/* Driver program to test mirror() */
int main()
{
    // Let us construct the BST shown in the above figure
    struct node *root = newNode(20);
    root->left = newNode(8);
    root->right = newNode(22);
    root->left->left = newNode(4);
    root->left->right = newNode(12);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(14);

    int n1 = 10, n2 = 14;
    struct node *t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d \n", n1, n2, t->data);

    n1 = 14, n2 = 8;
    t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d \n", n1, n2, t->data);

    n1 = 10, n2 = 22;
    t = lca(root, n1, n2);
    printf("LCA of %d and %d is %d \n", n1, n2, t->data);

    getchar();
    return 0;
}

```

Output:

```

LCA of 10 and 14 is 12
LCA of 14 and 8 is 8
LCA of 10 and 22 is 20

```

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**.

```

/* 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)
{
    while (root != NULL)
    {
        // If both n1 and n2 are smaller than root, then LCA lies in left
        if (root->data > n1 && root->data > n2)
            root = root->left;

        // If both n1 and n2 are greater than root, then LCA lies in right
        else if (root->data < n1 && root->data < n2)
            root = root->right;

        else break;
    }
    return root;
}

```

See [this](#) for complete program.

You may like to see [Lowest Common Ancestor in a Binary Tree](#) also.

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.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

10. The Great Tree-List Recursion Problem.

Asked by Varun Bhatia.

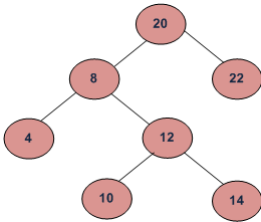
Question:

Write a recursive function `treeToList(Node root)` that takes an ordered binary tree and rearranges the internal pointers to make a circular doubly linked list out of the tree nodes. The "previous" pointers should be stored in the "small" field and the "next" pointers should be stored in the "large" field. The list should be arranged so that the nodes are in increasing order. Return the head pointer to the new list.

This is very well explained and implemented at <http://cslibrary.stanford.edu/109/TreeListRecursion.html>

11. Find the node with minimum value 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.



For the above tree, we start with 20, then we move left 8, we keep on moving to left until we see NULL. Since left of 4 is NULL, 4 is the node with minimum value.

```
#include <stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node
with the given data and NULL left and right
pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Give a binary search tree and a number,
inserts a new node with the given number in
the correct place in the tree. Returns the new
root pointer which the caller should then use
(the standard trick to avoid using reference
parameters). */
struct node* insert(struct node* node, int data)
{
    /* 1. If the tree is empty, return a new,
       single node */
    if (node == NULL)
        return(newNode(data));
    else
    {
        /* 2. Otherwise, recur down the tree */
```



```

    if (data <= node->data)
        node->left = insert(node->left, data);
    else
        node->right = insert(node->right, data);

    /* return the (unchanged) node pointer */
    return node;
}

/* Given a non-empty binary search tree,
return the minimum data value found in that
tree. Note that the entire tree does not need
to be searched. */
int minValue(struct node* node) {
    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL) {
        current = current->left;
    }
    return(current->data);
}

/* Driver program to test sameTree function*/
int main()
{
    struct node* root = NULL;
    root = insert(root, 4);
    insert(root, 2);
    insert(root, 1);
    insert(root, 3);
    insert(root, 6);
    insert(root, 5);

    printf("\n Minimum value in BST is %d", minValue(root));
    getchar();
    return 0;
}

```

Time Complexity: $O(n)$ Worst case happens for left skewed trees.

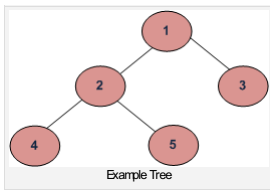
Similarly we can get the maximum value by recursively traversing the right node of a binary search tree.

References:

<http://cslibrary.stanford.edu/110/BinaryTrees.html>

12. Level Order Tree Traversal

Level order traversal of a tree is **breadth first traversal** for the 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.

```

/*Function to print level order traversal of tree*/
printLevelorder(tree)
for d = 1 to height(tree)
    printGivenLevel(tree, d);

/*Function to print all nodes at a given level*/
printGivenLevel(tree, level)
if tree is NULL then return;
if level is 1, then
    print(tree->data);
else if level greater than 1, then
    printGivenLevel(tree->left, level-1);
    printGivenLevel(tree->right, level-1);
  
```

Implementation:

```

#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/*Function prototypes*/
void printGivenLevel(struct node* root, int level);
int height(struct node* node);
struct node* newNode(int data);

/* Function to print level order traversal a tree*/
void printLevelOrder(struct node* root)
{
  
```

```

    int h = height(root);
    int i;
    for(i=1; i<=h; i++)
        printGivenLevel(root, i);
}

/* Print nodes at a given level */
void printGivenLevel(struct node* root, int level)
{
    if(root == NULL)
        return;
    if(level == 1)
        printf("%d ", root->data);
    else if (level > 1)
    {
        printGivenLevel(root->left, level-1);
        printGivenLevel(root->right, level-1);
    }
}

/* Compute the "height" of a tree -- the number of
   nodes along the longest path from the root node
   down to the farthest leaf node.*/
int height(struct node* node)
{
    if (node==NULL)
        return 0;
    else
    {
        /* compute the height of each subtree */
        int lheight = height(node->left);
        int rheight = height(node->right);

        /* use the larger one */
        if (lheight > rheight)
            return(lheight+1);
        else return(rheight+1);
    }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
}

```

```

printf("Level Order traversal of binary tree is \n");
printLevelOrder(root);

getchar();
return 0;
}

```

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 $O(n) + O(n-1) + O(n-2) + \dots + O(1)$ which is $O(n^2)$.

METHOD 2 (Use Queue)

Algorithm:

For each node, first the node is visited and then its 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 its value to temp_node

```

Implementation:

Here is a simple implementation of the above algorithm. Queue is implemented using an array with maximum size of 500. We can implement queue as linked list also.

```

#include <stdio.h>
#include <stdlib.h>
#define MAX_Q_SIZE 500

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* function prototypes */
struct node** createQueue(int *, int *);
void enqueue(struct node **, int *, struct node *);
struct node *deQueue(struct node **, int *);

/* Given a binary tree, print its nodes in level order
   using array for implementing queue */
void printLevelOrder(struct node* root)
{
    int rear, front;
    struct node **queue = createQueue(&front, &rear);
    struct node *temp_node = root;
    if (temp_node != NULL)
        enqueue(queue, rear, temp_node);
    while (temp_node != NULL)
    {
        temp_node = deQueue(queue, front);
        printf("%d\t", temp_node->data);
        if (temp_node->left != NULL)
            enqueue(queue, rear, temp_node->left);
        if (temp_node->right != NULL)
            enqueue(queue, rear, temp_node->right);
    }
}

```

```

    struct node *temp_node = root;

    while(temp_node)
    {
        printf("%d ", temp_node->data);

        /*Enqueue left child */
        if(temp_node->left)
            enqueue(queue, &rear, temp_node->left);

        /*Enqueue right child */
        if(temp_node->right)
            enqueue(queue, &rear, temp_node->right);

        /*Dequeue node and make it temp_node*/
        temp_node = dequeue(queue, &front);
    }
}

/*UTILITY FUNCTIONS*/
struct node** createQueue(int *front, int *rear)
{
    struct node **queue =
        (struct node **)malloc(sizeof(struct node)*MAX_Q_SIZE);

    *front = *rear = 0;
    return queue;
}

void enqueue(struct node **queue, int *rear, struct node *new_node)
{
    queue[*rear] = new_node;
    (*rear)++;
}

struct node *dequeue(struct node **queue, int *front)
{
    (*front)++;
    return queue[*front - 1];
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
}

```

```

printf("Level Order traversal of binary tree is \n");
printLevelOrder(root);

getchar();
return 0;
}

```

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

References:

http://en.wikipedia.org/wiki/Breadth-first_traversal

Please write comments if you find any bug in the above programs/algorithms or other ways to solve the same problem.

13. Program to count leaf nodes in a binary tree

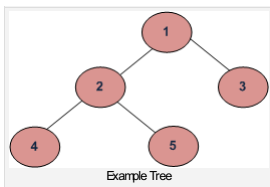
A node is a leaf node if both left and right child nodes of it are NULL.

Here is an algorithm to get the leaf node count.

```

getLeafCount(node)
1) If node is NULL then return 0.
2) Else If left and right child nodes are NULL return 1.
3) Else recursively calculate leaf count of the tree using below formula.
    Leaf count of a tree = Leaf count of left subtree +
                          Leaf count of right subtree

```



Leaf count for the above tree is 3.

Implementation:

```

#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Function to get the count of leaf nodes in a binary tree*/
unsigned int getLeafCount(struct node* node)
{
    if(node == NULL)
        return 0;
    if(node->left == NULL && node->right==NULL)
        return 1;
    else
        return getLeafCount(node->left)+
               getLeafCount(node->right);
}

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/*Driver program to test above functions*/
int main()
{
    /*create a tree*/
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    /*get leaf count of the above created tree*/
    printf("Leaf count of the tree is %d", getLeafCount(root));

    getchar();
    return 0;
}

```

Time & Space Complexities: Since this program is similar to traversal of tree, time and space complexities will be same as Tree traversal (Please see our [Tree Traversal](#) post for details)

Please write comments if you find any bug in the above programs/algorithms or other ways to solve the same problem.

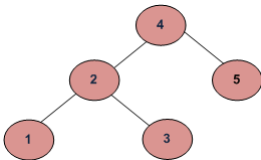
14. A program to check if a binary tree is BST or not

A binary search tree (BST) 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.
- Both the left and right subtrees must also be binary search trees.

From the above properties it naturally follows that:

- Each node (item in the tree) has a distinct key.



METHOD 1 (Simple but Wrong)

Following is a simple program. For each node, check if left node of it is smaller than the node and right node of it is greater than the node.

```
int isBST(struct node* node)
{
    if (node == NULL)
        return 1;

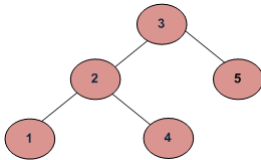
    /* false if left is > than node */
    if (node->left != NULL && node->left->data > node->data)
        return 0;

    /* false if right is < than node */
    if (node->right != NULL && node->right->data < node->data)
        return 0;

    /* false if, recursively, the left or right is not a BST */
    if (!isBST(node->left) || !isBST(node->right))
        return 0;

    /* passing all that, it's a BST */
    return 1;
}
```

This approach is wrong as this will return true for below binary tree (and below tree is not a BST because 4 is in left subtree of 3)



METHOD 2 (Correct but not efficient)

For each node, check if max value in left subtree is smaller than the node and min value in right subtree greater than the node.

```

/* Returns true if a binary tree is a binary search tree */
int isBST(struct node* node)
{
    if (node == NULL)
        return(true);

    /* false if the max of the left is > than us */
    if (node->left!=NULL && maxValue(node->left) > node->data)
        return(false);

    /* false if the min of the right is <= than us */
    if (node->right!=NULL && minValue(node->right) < node->data)
        return(false);

    /* false if, recursively, the left or right is not a BST */
    if (!isBST(node->left) || !isBST(node->right))
        return(false);

    /* passing all that, it's a BST */
    return(true);
}

```

It is assumed that you have helper functions `minValue()` and `maxValue()` that return the min or max int value from a non-empty tree

METHOD 3 (Correct and Efficient)

Method 2 above runs slowly since it traverses over some parts of the tree many times. A better solution looks at each node only once. The trick is to write a utility helper function `isBSTUtil(struct node* node, int min, int max)` that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once. The initial values for min and max should be `INT_MIN` and `INT_MAX` — they narrow from there.

```

/* Returns true if the given tree is a binary search tree
(efficient version). */
int isBST(struct node* node)
{
    return(isBSTUtil(node, INT_MIN, INT_MAX));
}

```

```

}

/* Returns true if the given tree is a BST and its
   values are >= min and <= max. */
int isBSTUtil(struct node* node, int min, int max)

```

Implementation:

```

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

int isBSTUtil(struct node* node, int min, int max);

/* Returns true if the given tree is a binary search tree
   (efficient version). */
int isBST(struct node* node)
{
    return(isBSTUtil(node, INT_MIN, INT_MAX));
}

/* Returns true if the given tree is a BST and its
   values are >= min and <= max. */
int isBSTUtil(struct node* node, int min, int max)
{
    /* an empty tree is BST */
    if (node==NULL)
        return 1;

    /* false if this node violates the min/max constraint */
    if (node->data < min || node->data > max)
        return 0;

    /* otherwise check the subtrees recursively,
       tightening the min or max constraint */
    return
        isBSTUtil(node->left, min, node->data-1) && // Allow only distinct
        isBSTUtil(node->right, node->data+1, max); // Allow only distinct
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return node;
}

```

```

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(4);
    root->left = newNode(2);
    root->right = newNode(5);
    root->left->left = newNode(1);
    root->left->right = newNode(3);

    if(isBST(root))
        printf("Is BST");
    else
        printf("Not a BST");

    getchar();
    return 0;
}

```

Time Complexity: $O(n)$

Auxiliary Space : $O(1)$ if Function Call Stack size is not considered, otherwise $O(n)$

METHOD 4(Using In-Order Traversal)

Thanks to [LJV489](#) for suggesting this method.

- 1) Do In-Order Traversal of the given tree and store the result in a temp array.
- 3) Check if the temp array is sorted in ascending order, if it is, then the tree is BST.

Time Complexity: $O(n)$

We can avoid the use of Auxiliary Array. While doing In-Order traversal, we can keep track of previously visited node. If the value of the currently visited node is less than the previous value, then tree is not BST. Thanks to [ygos](#) for this space optimization.

```

bool isBST(struct node* root)
{
    static struct node *prev = NULL;

    // traverse the tree in inorder fashion and keep track of prev node
    if (root)
    {
        if (!isBST(root->left))
            return false;

        // Allows only distinct valued nodes
        if (prev != NULL && root->data <= prev->data)
            return false;

        prev = root;

        return isBST(root->right);
    }

    return true;
}

```

The use of static variable can also be avoided by using reference to prev node as a parameter (Similar to [this](#) post).

Sources:

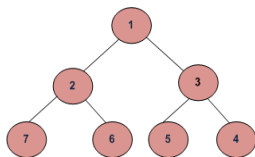
http://en.wikipedia.org/wiki/Binary_search_tree

<http://cslibrary.stanford.edu/110/BinaryTrees.html>

Please write comments if you find any bug in the above programs/algorithms or other ways to solve the same problem.

15. 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 be 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);

```

Following is C implementation of above algorithm.

```

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Function prototypes */
void printGivenLevel(struct node* root, int level, int ltr);
int height(struct node* node);
struct node* newNode(int data);

/* Function to print spiral traversal of a tree*/
void printSpiral(struct node* root)
{
    int h = height(root);
    int i;

    /*ltr -> Left to Right. If this variable is set,
       then the given level is traverseed from left to right. */
    bool ltr = false;
    for(i=1; i<=h; i++)
    {
        printGivenLevel(root, i, ltr);

        /*Revert ltr to traverse next level in opposite order*/
        ltr = !ltr;
    }
}

/* Print nodes at a given level */
void printGivenLevel(struct node* root, int level, int ltr)
{
    if(root == NULL)
        return;
    if(level == 1)
        printf("%d ", root->data);
    else if (level > 1)
    {
        if(ltr)
        {
            printGivenLevel(root->left, level-1, ltr);
            printGivenLevel(root->right, level-1, ltr);
        }
        else

```

```

        {
            printGivenLevel(root->right, level-1, ltr);
            printGivenLevel(root->left, level-1, ltr);
        }
    }
}

/* Compute the "height" of a tree -- the number of
   nodes along the longest path from the root node
   down to the farthest leaf node.*/
int height(struct node* node)
{
    if (node==NULL)
        return 0;
    else
    {
        /* compute the height of each subtree */
        int lheight = height(node->left);
        int rheight = height(node->right);

        /* use the larger one */
        if (lheight > rheight)
            return(lheight+1);
        else return(rheight+1);
    }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left = newNode(7);
    root->left->right = newNode(6);
    root->right->left = newNode(5);
    root->right->right = newNode(4);
    printf("Spiral Order traversal of binary tree is \n");
    printSpiral(root);

    return 0;
}

```

Output:

```
Spiral Order traversal of binary tree is
```

```
1 2 3 4 5 6 7
```

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.

```
// C++ implementation of a O(n) time method for spiral order traversal
#include <iostream>
#include <stack>
using namespace std;

// Binary Tree node
struct node
{
    int data;
    struct node *left, *right;
};

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
    stack<struct node*> s2; // For levels to be printed from left to

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

    // Keep printing 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);
    }
}

```

// A utility function to create a new node

```

struct node* newNode(int data)
{
    struct node* node = new struct node;
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

int main()
{
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(7);
    root->left->right = newNode(6);
    root->right->left = newNode(5);
    root->right->right = newNode(4);
    cout << "Spiral Order traversal of binary tree is \n";
    printSpiral(root);

    return 0;
}

```

Output:

```

Spiral Order traversal of binary tree is
1 2 3 4 5 6 7

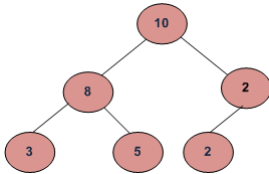
```

Please write comments if you find any bug in the above program/algorithm; or if you want to share more information about spiral traversal.

16. 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.

Implementation:

```

/* Program to check children sum property */
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, left child and right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* returns 1 if children sum property holds for the given
   node and both of its children*/
int isSumProperty(struct node* node)
{
    /* left_data is left child data and right_data is for right child da
    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;
    }
}
  
```

```

/*
Helper function that allocates a new node
with the given data and NULL left and right
pointers.
*/
struct node* newNode(int data)
{
    struct node* node =
        (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

/* Driver program to test above function */
int main()
{
    struct node *root = newNode(10);
    root->left = newNode(8);
    root->right = newNode(2);
    root->left->left = newNode(3);
    root->left->right = newNode(5);
    root->right->right = newNode(2);
    if(isSumProperty(root))
        printf("The given tree satisfies the children sum property ");
    else
        printf("The given tree does not satisfy the children sum property

    getchar();
    return 0;
}

```

Time Complexity: $O(n)$, we are doing a complete traversal of the tree.

As an exercise, extend the above question for an n-ary tree.

This question was asked by Shekhar.

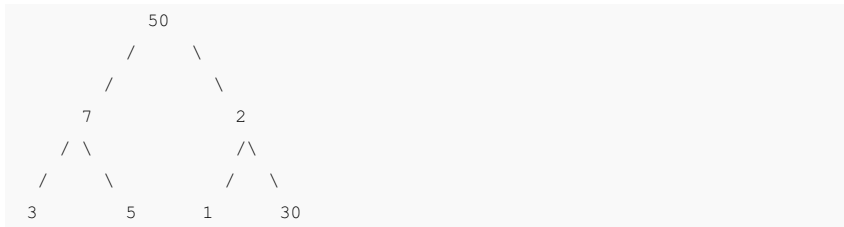
Please write comments if you find any bug in the above algorithm or a better way to solve the same problem.

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

Question: 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
```

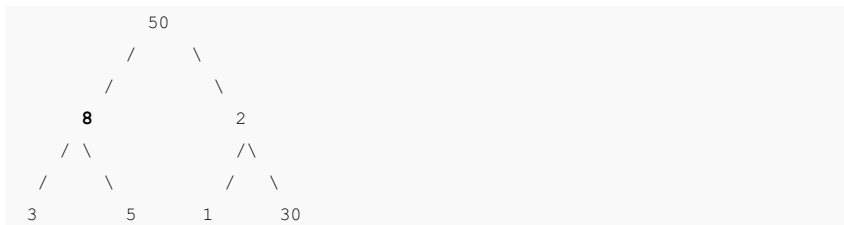
If diff is 0 then nothing needs to be done.

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

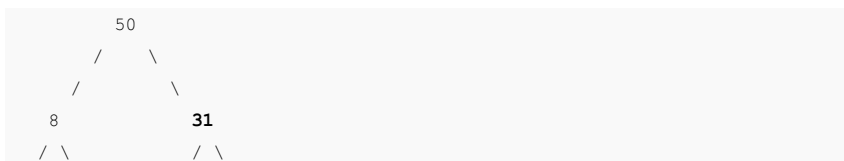
If diff < 0 (node's data is greater than the node's children sum) then increment one child's data. We can choose to increment either left or right child if they both are not NULL. Let us always first increment the left child. Incrementing a child changes the subtree's children sum property so we need 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)

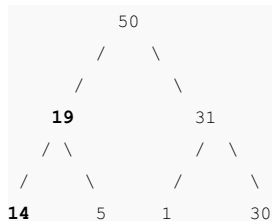


```

/      \      /      \
3        5    1       30

```

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.

Implementation:

```

/* Program to convert an arbitrary binary tree to
a tree that holds children sum property */

#include <stdio.h>
#include <stdlib.h>

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

/* This function is used to increment left subtree */
void increment(struct node* node, int diff);

/* Helper function that allocates a new node
with the given data and NULL left and right
pointers. */
struct node* newNode(int data);

/* This function changes a tree to hold children sum
property */
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 */

```

```

    /* 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);
    }
}
}

```

```

/* Given a binary tree, printInorder() prints out its
inorder traversal*/

```

```

void printInorder(struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    printf("%d ", node->data);

    /* now recur on right child */
    printInorder(node->right);
}

```

```

/* Helper function that allocates a new node
with the given data and NULL left and right
pointers. */

```

```

struct node* newNode(int data)
{
    struct node* node =
        (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

/* Driver program to test above functions */
int main()
{
    struct node *root = newNode(50);
    root->left      = newNode(7);
    root->right     = newNode(2);
    root->left->left = newNode(3);
    root->left->right = newNode(5);
    root->right->left = newNode(1);
    root->right->right = newNode(30);

    printf("\n Inorder traversal before conversion ");
    printInorder(root);

    convertTree(root);

    printf("\n Inorder traversal after conversion ");
    printInorder(root);

    getchar();
    return 0;
}

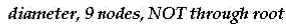
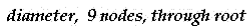
```

Time Complexity: $O(n^2)$, Worst case complexity is for a skewed tree such that nodes are in decreasing order from root to leaf.

Please write comments if you find any bug in the above algorithm or a better way to solve the same problem.

18. 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).



- * 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)

```
#include <stdio.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* function to create a new node of tree and returns pointer */
struct node* newNode(int data);

/* returns max of two integers */
int max(int a, int b);

/* function to Compute height of a tree. */
int height(struct node* node);

/* Function to get diameter of a binary tree */
int diameter(struct node * tree)
{
    /* base case where tree is empty */
    if (tree == 0)
        return 0;

    /* get the height of left and right sub-trees */
    int lheight = height(tree->left);
    int rheight = height(tree->right);

    /* get the diameter of left and right sub-trees */
    int ldiameter = diameter(tree->left);
    int rdiameter = diameter(tree->right);

    /* Return max of following three
    1) Diameter of left subtree
    2) Diameter of right subtree
    3) Height of left subtree + height of right subtree + 1 */
```

```

    return max(lheight + rheight + 1, max(ldiameter, rdiameter));
}

/* UTILITY FUNCTIONS TO TEST diameter() FUNCTION */

/* The function Compute the "height" of a tree. Height is the
   number of nodes along the longest path from the root node
   down to the farthest leaf node.*/
int height(struct node* node)
{
    /* base case tree is empty */
    if(node == NULL)
        return 0;

    /* If tree is not empty then height = 1 + max of left
       height and right heights */
    return 1 + max(height(node->left), height(node->right));
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* returns maximum of two integers */
int max(int a, int b)
{
    return (a >= b)? a: b;
}

/* Driver program to test above functions*/
int main()
{
    /* Constructed binary tree is
      1
     / \
    2   3
   / \
  4   5
  */
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Diameter of the given binary tree is %d\n", diameter(root));

    getchar();
    return 0;
}

```


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 rdiameter */
    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: $O(n)$

References:

<http://www.cs.duke.edu/courses/spring00/cps100/assign/trees/diameter.html>

Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

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

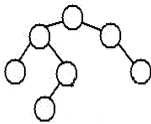
A tree where no leaf is much farther away from the root than any other leaf. Different balancing schemes allow different definitions of “much farther” and different amounts of work to keep them balanced.

Consider a height-balancing scheme where following conditions should be checked to determine if a binary tree is balanced.

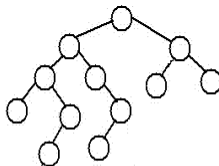
An empty tree is height-balanced. A non-empty binary tree T is balanced if:

- 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.

```
/* program to check if a tree is height-balanced or not */
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Returns the height of a binary tree */
int height(struct node* node);

/* Returns true if binary tree with root as root is height-balanced */
bool isBalanced(struct node *root)
{
    int lh; /* for height of left subtree */
    int rh; /* for height of right subtree */

    /* If tree is empty then return true */
    if(root == NULL)
        return 1;
```

```

    return 1;

    /* Get the height of left and right sub trees */
    lh = height(root->left);
    rh = height(root->right);

    if( abs(lh-rh) <= 1 &&
        isBalanced(root->left) &&
        isBalanced(root->right))
        return 1;

    /* If we reach here then tree is not height-balanced */
    return 0;
}

/* UTILITY FUNCTIONS TO TEST isBalanced() FUNCTION */

/* returns maximum of two integers */
int max(int a, int b)
{
    return (a >= b)? a: b;
}

/* The function Compute the "height" of a tree. Height is the
   number of nodes along the longest path from the root node
   down to the farthest leaf node.*/
int height(struct node* node)
{
    /* base case tree is empty */
    if(node == NULL)
        return 0;

    /* If tree is not empty then height = 1 + max of left
       height and right heights */
    return 1 + max(height(node->left), height(node->right));
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

int main()
{
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->left->left->left = newNode(8);

    if(isBalanced(root))
        printf("Tree is balanced");
    else
        printf("Tree is not balanced");
}

```

```

    getchar();
    return 0;
}

```

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 $O(n)$.

```

/* program to check if a tree is height-balanced or not */
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* 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))

```

```

    if((l1 - r1 >= 2) || (r1 - l1 >= 2))
        return 0;

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

/* UTILITY FUNCTIONS TO TEST isBalanced() FUNCTION */

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));

    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

int main()
{
    int height = 0;

    /* Constructed binary tree is
        1
       / \
      2   3
     / \ / \
    4  5 6  7
   */
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->left->left->left = newNode(7);

    if(isBalanced(root, &height))
        printf("Tree is balanced");
    else
        printf("Tree is not balanced");

    getchar();
    return 0;
}

```

Time Complexity: $O(n)$

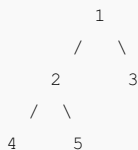
Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

20. 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.

Let us consider the below tree for example



Step 1 Creates an empty stack: S = NULL

Step 2 sets current as address of root: current -> 1

Step 3 Pushes the current node and set current = current->left until current is NULL

```
current -> 1
push 1: Stack S -> 1
current -> 2
push 2: Stack S -> 2, 1
current -> 4
push 4: Stack S -> 4, 2, 1
current = NULL
```

Step 4 pops from S

- a) Pop 4: Stack S -> 2, 1
- b) print "4"
- c) current = NULL /*right of 4 */ and go to step 3

Since current is NULL step 3 doesn't do anything.

Step 4 pops again.

```
a) Pop 2: Stack S -> 1
b) print "2"
c) current -> 5/*right of 2 */ and go to step 3
```

Step 3 pushes 5 to stack and makes current NULL

```
Stack S -> 5, 1
current = NULL
```

Step 4 pops from S

```
a) Pop 5: Stack S -> 1
b) print "5"
c) current = NULL /*right of 5 */ and go to step 3
```

Since current is NULL step 3 doesn't do anything

Step 4 pops again.

```
a) Pop 1: Stack S -> NULL
b) print "1"
c) current -> 3 /*right of 5 */
```

Step 3 pushes 3 to stack and makes current NULL

```
Stack S -> 3
current = NULL
```

Step 4 pops from S

```
a) Pop 3: Stack S -> NULL
b) print "3"
c) current = NULL /*right of 3 */
```

Traversal is done now as stack S is empty and current is NULL.

Implementation:

```
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* A binary tree tNode has data, pointer to left child
   and a pointer to right child */
struct tNode
{
    int data;
    struct tNode* left;
    struct tNode* right;
};

/* Structure of a stack node. Linked List implementation is used for
   stack. A stack node contains a pointer to tree node and a pointer to
   next stack node */
struct sNode
{
    struct tNode *t;
```

```

    struct sNode *next;
};

/* Stack related functions */
void push(struct sNode** top_ref, struct tNode *t);
struct tNode *pop(struct sNode** top_ref);
bool isEmpty(struct sNode *top);

/* Iterative function for inorder tree traversal */
void inOrder(struct tNode *root)
{
    /* set current to root of binary tree */
    struct tNode *current = root;
    struct sNode *s = NULL; /* Initialize stack s */
    bool done = 0;

    while (!done)
    {
        /* Reach the left most tNode of the current tNode */
        if(current != NULL)
        {
            /* place pointer to a tree node on the stack before traversing
               the node's left subtree */
            push(&s, current);
            current = current->left;
        }

        /* backtrack from the empty subtree and visit the tNode
           at the top of the stack; however, if the stack is empty,
           you are done */
        else
        {
            if (!isEmpty(s))
            {
                current = pop(&s);
                printf("%d ", current->data);

                /* we have visited the node and its left subtree.
                   Now, it's right subtree's turn */
                current = current->right;
            }
            else
            {
                done = 1;
            }
        } /* end of while */
    }

    /* UTILITY FUNCTIONS */
    /* Function to push an item to sNode*/
    void push(struct sNode** top_ref, struct tNode *t)
    {
        /* allocate tNode */
        struct sNode* new_tNode =
            (struct sNode*) malloc(sizeof(struct sNode));

        if(new_tNode == NULL)
        {
            printf("Stack Overflow \n");
            getchar();
            exit(0);
        }
    }
}

```



```

    /* put in the data */
    new_tNode->t = t;

    /* link the old list off the new tNode */
    new_tNode->next = (*top_ref);

    /* move the head to point to the new tNode */
    (*top_ref) = new_tNode;
}

/* The function returns true if stack is empty, otherwise false */
bool isEmpty(struct sNode *top)
{
    return (top == NULL)? 1 : 0;
}

/* Function to pop an item from stack*/
struct tNode *pop(struct sNode** top_ref)
{
    struct tNode *res;
    struct sNode *top;

    /*If sNode is empty then error */
    if(isEmpty(*top_ref))
    {
        printf("Stack Underflow \n");
        getchar();
        exit(0);
    }
    else
    {
        top = *top_ref;
        res = top->t;
        *top_ref = top->next;
        free(top);
        return res;
    }
}

/* Helper function that allocates a new tNode with the
   given data and NULL left and right pointers. */
struct tNode* newtNode(int data)
{
    struct tNode* tNode = (struct tNode*)
        malloc(sizeof(struct tNode));

    tNode->data = data;
    tNode->left = NULL;
    tNode->right = NULL;

    return(tNode);
}

/* Driver program to test above functions*/
int main()
{

```

```

    /* Constructed binary tree is

```

```

        1
       / \
      2   3
     / \
    4   5
   */

```

```

/
struct tNode *root = newtNode(1);
root->left      = newtNode(2);
root->right     = newtNode(3);
root->left->left = newtNode(4);
root->left->right = newtNode(5);

inOrder(root);

getchar();
return 0;
}

```

Time Complexity: $O(n)$

References:

<http://web.cs.wpi.edu/~cs2005/common/iterative.inorder>

<http://neural.cs.nthu.edu.tw/jang/courses/cs2351/slide/animation/Iterative%20Inorder%20Traver>

See [this post](#) for another approach of Inorder Tree Traversal without recursion and without stack!

Please write comments if you find any bug in above code/algorithm, or want to share more information about stack based Inorder Tree Traversal.

21. 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.

```

#include<stdio.h>
#include<stdlib.h>

```

```

/* A binary tree tNode has data, pointer to left child
   and a pointer to right child */
struct tNode
{
    int data;
    struct tNode* left;
    struct tNode* right;
};

/* 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 predecessor */
            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 */
}

/* UTILITY FUNCTIONS */
/* Helper function that allocates a new tNode with the
   given data and NULL left and right pointers. */
struct tNode* newtNode(int data)
{
    struct tNode* tNode = (struct tNode*)
                           malloc(sizeof(struct tNode));
    tNode->data = data;
    tNode->left = NULL;
    tNode->right = NULL;
}

```

```

        tNode->right = NULL;
    }
    return(tNode);
}

/* Driver program to test above functions*/
int main()
{
    /* Constructed binary tree is
        1
       / \
      2   3
     / \
    4   5
    */
    struct tNode *root = newtNode(1);
    root->left      = newtNode(2);
    root->right     = newtNode(3);
    root->left->left = newtNode(4);
    root->left->right = newtNode(5);

    MorrisTraversal(root);

    getch();
    return 0;
}

```

References:

www.liacs.nl/~deutz/DS/september28.pdf

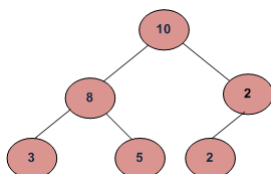
<http://comsci.liu.edu/~murali/algo/Morris.htm>

www.scss.tcd.ie/disciplines/software_systems/.../HughGibbonsSlides.pdf

Please write comments if you find any bug in above code/algorithm, or want to share more information about stack Morris Inorder Tree Traversal.

22. 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)

Implementation:

```
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/*
Given a tree and a sum, return true if there is a path from the root
down to a leaf, such that adding up all the values along the path
equals the given sum.

Strategy: subtract the node value from the sum when recurring down,
and check to see if the sum is 0 when you run out of tree.
*/
bool hasPathSum(struct node* node, int sum)
{
    /* return true if we run out of tree and sum==0 */
    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;
    }
}
```

```

    }
}

/* UTILITY FUNCTIONS */
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    int sum = 21;

    /* Constructed binary tree is
        10
       / \
      8   2
     / \ / \
    3  5 2
   */
    struct node *root = newNode(10);
    root->left = newNode(8);
    root->right = newNode(2);
    root->left->left = newNode(3);
    root->left->right = newNode(5);
    root->right->left = newNode(2);

    if(hasPathSum(root, sum))
        printf("There is a root-to-leaf path with sum %d", sum);
    else
        printf("There is no root-to-leaf path with sum %d", sum);

    getchar();
    return 0;
}

```

Time Complexity: O(n)

References:

<http://cslibrary.stanford.edu/110/BinaryTrees.html>

Author: Tushar Roy

Please write comments if you find any bug in above code/algorithm, or find other ways to solve the same problem

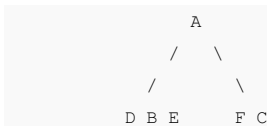
23. Construct Tree from given Inorder and Preorder traversals

Let us consider the below traversals:

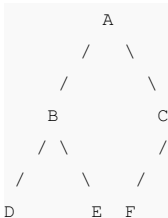
Inorder sequence: D B E A F C

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.

Thanks to Rohini and Tushar for suggesting the code.

```
/* program to construct tree using inorder and preorder traversals */
#include<stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
```

```

struct node
{
    char data;
    struct node* left;
    struct node* right;
};

/* Prototypes for utility functions */
int search(char arr[], int strt, int end, char value);
struct node* newNode(char data);

/* Recursive function to construct binary of size len from
Inorder traversal in[] and Preorder traversal pre[]. Initial value
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 subtreess */
    tNode->left = buildTree(in, pre, inStrt, inIndex-1);
    tNode->right = buildTree(in, pre, inIndex+1, inEnd);

    return tNode;
}

/* UTILITY FUNCTIONS */
/* Function to find index of value in arr[start...end]
The function assumes that value is present in in[] */
int search(char arr[], int strt, int end, char value)
{
    int i;
    for(i = strt; i <= end; i++)
    {
        if(arr[i] == value)
            return i;
    }
}

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node* newNode(char data)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
}

```



```

    return(node);
}

/* This funcion is here just to test buildTree() */
void printInorder(struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    /* then print the data of node */
    printf("%c ", node->data);

    /* now recur on right child */
    printInorder(node->right);
}

/* Driver program to test above functions */
int main()
{
    char in[] = {'D', 'B', 'E', 'A', 'F', 'C'};
    char pre[] = {'A', 'B', 'D', 'E', 'C', 'F'};
    int len = sizeof(in)/sizeof(in[0]);
    struct node *root = buildTree(in, pre, 0, len - 1);

    /* Let us test the built tree by printing Insorder traversal */
    printf("\n Inorder traversal of the constructed tree is \n");
    printInorder(root);
    getchar();
}

```

Time Complexity: $O(n^2)$. Worst case occurs when tree is left skewed. Example
 Preorder and Inorder traversals for worst case are {A, B, C, D} and {D, C, B, A}.

Please write comments if you find any bug in above codes/algorithms, or find other
 ways to solve the same problem.

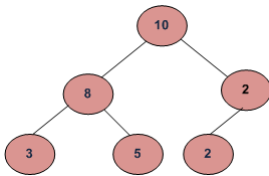
24. Given a binary tree, print all root-to-leaf paths

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

```

10 → 8 → 3
10 → 8 → 5
10 → 2 → 2

```



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.

```

#include<stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Prototypes for funtions needed in printPaths() */
void printPathsRecur(struct node* node, int path[], int pathLen);
void printArray(int ints[], int len);

/*Given a binary tree, print out all of its root-to-leaf
paths, one per line. Uses a recursive helper to do the work.*/
void printPaths(struct node* node)
{
    int path[1000];
    printPathsRecur(node, path, 0);
}

/* Recursive helper function -- given a node, and an array containing
the path from the root node up to but not including this node,
print out all the root-leaf paths.*/
void printPathsRecur(struct node* node, int path[], int pathLen)
{
    if (node==NULL)
        return;

    /* append this node to the path array */
    path[pathLen] = node->data;
    pathLen++;

    /* it's a leaf, so print the path that led to here */
    if (node->left==NULL && node->right==NULL)
    {
        printArray(path, pathLen);
    }
    else
    {
        /* otherwise try both subtrees */
        printPathsRecur(node->left, path, pathLen);
        printPathsRecur(node->right, path, pathLen);
    }
}
  
```

J

```
/* UTILITY FUNCTIONS */
/* Utility that prints out an array on a line. */
void printArray(int ints[], int len)
{
    int i;
    for (i=0; i<len; i++)
    {
        printf("%d ", ints[i]);
    }
    printf("\n");
}

/* utility that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    /* Constructed binary tree is
        10
       /  \
      8    2
     /  \  /
    3    5 2
    */
    struct node *root = newNode(10);
    root->left = newNode(8);
    root->right = newNode(2);
    root->left->left = newNode(3);
    root->left->right = newNode(5);
    root->right->left = newNode(2);

    printPaths(root);

    getchar();
    return 0;
}
```

Time Complexity: O(n)

References:

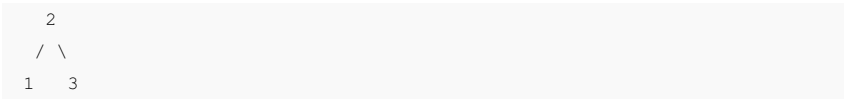
<http://cslibrary.stanford.edu/110/BinaryTrees.html>

Please write comments if you find any bug in above codes/algorithms, or find other ways to solve the same problem.

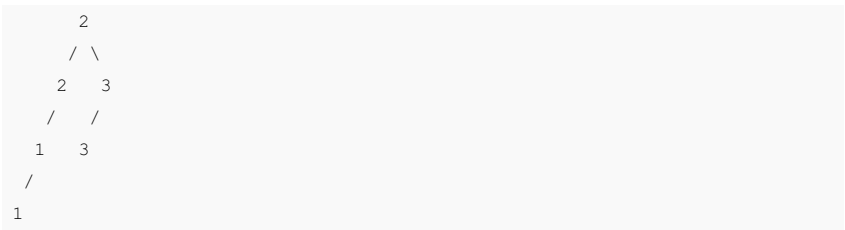
25. Double Tree

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.

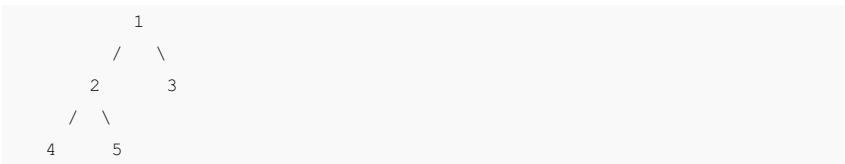
So the tree...



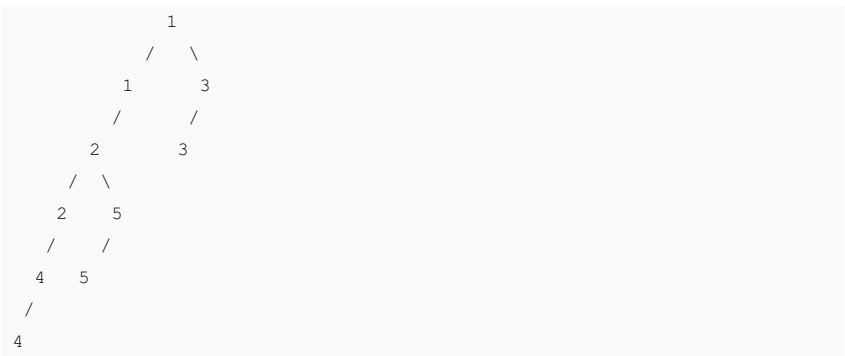
is changed to...



And 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.

Implementation:

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* function to create a new node of tree and returns pointer */
struct node* newNode(int data);

/* 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;
}

/* UTILITY FUNCTIONS TO TEST doubleTree() FUNCTION */
/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
```

```

    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

/* Driver program to test above functions*/
int main()
{
    /* Constructed binary tree is
        1
       / \
      2   3
     / \
    4   5
    */
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Inorder traversal of the original tree is \n");
    printInorder(root);

    doubleTree(root);

    printf("\n Inorder traversal of the double tree is \n");
    printInorder(root);

    getchar();
    return 0;
}

```

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

References:

<http://cslibrary.stanford.edu/110/BinaryTrees.html>

Please write comments if you find any bug in above code/algorithm, or find other ways to solve the same problem.

26. 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
   / \

```



For the above tree,
width of level 1 is 1,
width of level 2 is 2,
width of level 3 is 3
width of level 4 is 2.

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.

```
/*Function to print level order traversal of tree*/
```

```
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);
```

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
/* A binary tree node has data, pointer to left child  
and a pointer to right child */
```

```
struct node
```

```
{
```

```
    int data;
```

```
    struct node* left;
```

```
    struct node* right;
```

```
};
```

```
/*Function prototypes*/
```

```
int getWidth(struct node* root, int level);
```

```

int height(struct node* node);
struct node* newNode(int data);

/* Function to get the maximum width of a binary tree*/
int getMaxWidth(struct node* root)
{
    int maxWidth = 0;
    int width;
    int h = height(root);
    int i;

    /* Get width of each level and compare
    the width with maximum width so far */
    for(i=1; i<=h; i++)
    {
        width = getWidth(root, i);
        if(width > maxWidth)
            maxWidth = width;
    }

    return maxWidth;
}

/* Get width of a given level */
int getWidth(struct node* root, int level)
{
    if(root == NULL)
        return 0;

    if(level == 1)
        return 1;

    else if (level > 1)
        return getWidth(root->left, level-1) +
            getWidth(root->right, level-1);
}

/* UTILITY FUNCTIONS */
/* Compute the "height" of a tree -- the number of
nodes along the longest path from the root node
down to the farthest leaf node.*/
int height(struct node* node)
{
    if (node==NULL)
        return 0;
    else
    {
        /* compute the height of each subtree */
        int lHeight = height(node->left);
        int rHeight = height(node->right);
        /* use the larger one */

        return (lHeight > rHeight)? (lHeight+1): (rHeight+1);
    }
}

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));

```



```

        malloc(sizeof(struct node));
node->data = data;
node->left = NULL;
node->right = NULL;
return(node);
}
/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(8);
    root->right->right->left = newNode(6);
    root->right->right->right = newNode(7);

    /*
    Constructed binary tree is:
        1
       / \
      2   3
     / \   \
    4  5   8
         / \
        6  7
    */
    printf("Maximum width is %d \n", getMaxWidth(root));
    getch();
    return 0;
}

```

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. Thanks to [Nitish](#), [DivyaC](#) and [tech.login.id2](#) for suggesting this optimization. See their comments for implementation using queue based traversal.

Method 2 (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.

```

#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

```

```

// A utility function to get height of a binary tree
int height(struct node* node);

// A utility function to allocate a new node with given data
struct node* newNode(int data);

// A utility function that returns maximum value in arr[] of size n
int getMax(int arr[], int n);

// 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);
    }
}

/* UTILITY FUNCTIONS */
/* Compute the "height" of a tree -- the number of
   nodes along the longest path from the root node
   down to the farthest leaf node.*/
int height(struct node* node)
{
    if (node==NULL)
        return 0;
    else
    {
        /* compute the height of each subtree */
        int lHeight = height(node->left);
        int rHeight = height(node->right);
        /* use the larger one */
        return (lHeight > rHeight)? (lHeight+1): (rHeight+1);
    }
}

```

```

}
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

// Return the maximum value from count array
int getMax(int arr[], int n)
{
    int max = arr[0];
    int i;
    for (i = 0; i < n; i++)
    {
        if (arr[i] > max)
            max = arr[i];
    }
    return max;
}

/* Driver program to test above functions*/
int main()
{
    struct node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(8);
    root->right->right->left = newNode(6);
    root->right->right->right = newNode(7);

    /*
       Constructed bunary tree is:
           1
          / \
         2  3
        / \  \
       4  5  8
            / \
           6  7
    */
    printf("Maximum width is %d \n", getMaxWidth(root));
    getchar();
    return 0;
}

```

Thanks to [Raja](#) and [jagdish](#) for suggesting this method.

Time Complexity: $O(n)$

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem.

27. Total number of possible Binary Search Trees with n keys

Total number of possible Binary Search Trees with n different keys = Catalan number
 $C_n = \frac{(2n)!}{(n+1)!n!}$

See references for proof and examples.

References:

http://en.wikipedia.org/wiki/Catalan_number

28. Foldable Binary Trees

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

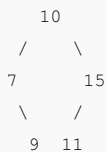
A tree can be folded if left and right subtrees of the tree are structure wise mirror image of each other. An empty tree is considered as foldable.

Consider the below trees:

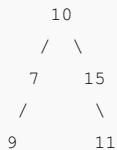
(a) and (b) can be folded.

(c) and (d) cannot be folded.

(a)

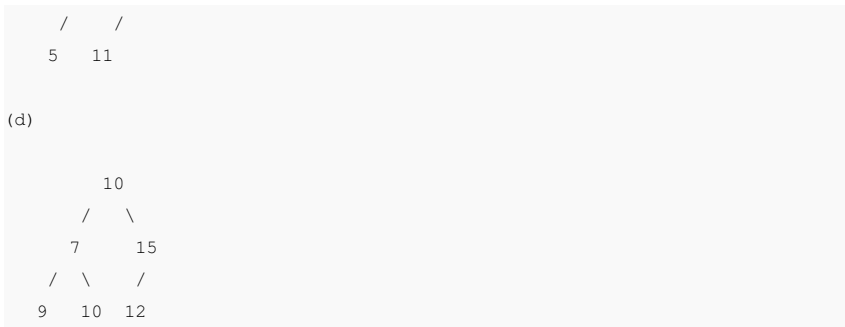


(b)



(c)





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.

Thanks to [ajaym](#) for suggesting this approach.

```

#include<stdio.h>
#include<stdlib.h>

/* You would want to remove below 3 lines if your compiler
supports bool, true and false */
#define bool int
#define true 1
#define false 0

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* converts a tree to its mirror image */
void mirror(struct node* node);

/* returns true if structure of two trees a and b is same
Only structure is considered for comparison, not data! */
bool isStructSame(struct node *a, struct node *b);

```

```

/* Returns true if the given tree is foldable */
bool isFoldable(struct node *root)
{
    bool res;

    /* base case */
    if(root == NULL)
        return true;

    /* convert left subtree to its mirror */
    mirror(root->left);

    /* Compare the structures of the right subtree and mirrored
    left subtree */
    res = isStructSame(root->left, root->right);

    /* Get the original tree back */
    mirror(root->left);

    return res;
}

```

```

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;
}

```

```

/* UTILITY FUNCTIONS */
/* Change a tree so that the roles of the left and
right pointers are swapped at every node.
See http://geeksforgeeks.org/?p=662 for details */
void mirror(struct node* node)

```

```

{
    if (node==NULL)
        return;
    else
    {
        struct node* temp;

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

        /* swap the pointers in this node */
        temp      = node->left;
        node->left = node->right;
        node->right = temp;
    }
}

```

```

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node* newNode(int data)

```

```

struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test mirror() */
int main(void)
{
    /* The constructed binary tree is
        1
       / \
      2   3
     / \ / \
    4  5
    */
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->right->left = newNode(4);
    root->left->right = newNode(5);

    if(isFoldable(root) == 1)
    { printf("\n tree is foldable"); }
    else
    { printf("\n tree is not foldable"); }

    getchar();
    return 0;
}

```

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

```

#include<stdio.h>
#include<stdlib.h>

/* You would want to remove below 3 lines if your compiler
   supports bool, true and false */
#define bool int
#define true 1
#define false 0

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* 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);

/* Returns true if the given tree can be folded */
bool IsFoldable(struct node *root)
{
    if (root == NULL)
    { return true; }

    return IsFoldableUtil(root->left, root->right);
}

/* 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);
}

/*UTILITY FUNCTIONS */
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
}

```



```

    return(node);
}

/* Driver program to test mirror() */
int main(void)
{
    /* The constructed binary tree is
        1
       / \
      2   3
     \   /
    4   5
    */
    struct node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->right = newNode(4);
    root->right->left = newNode(5);

    if(IsFoldable(root) == true)
    { printf("\n tree is foldable"); }
    else
    { printf("\n tree is not foldable"); }

    getchar();
    return 0;
}

```

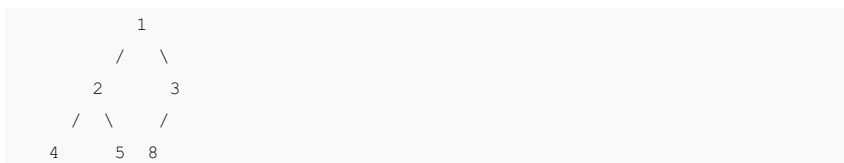
Thanks to [Dzmitry Huba](#) for suggesting this approach.

Please write comments if you find the above code/algorithm incorrect, or find other ways to solve the same problem.

29. 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.



The problem can be solved using recursion. Thanks to [eldho](#) for suggesting the solution.

```

#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child

```

```

    and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

void printKDistant(node *root , int k)
{
    if(root == NULL)
        return;
    if( k == 0 )
    {
        printf( "%d ", root->data );
        return ;
    }
    else
    {
        printKDistant( root->left, k-1 );
        printKDistant( root->right, k-1 );
    }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    /* Constructed binary tree is
      1
     / \
    2   3
   / \ / \
  4  5 8
    */
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(8);

    printKDistant(root, 2);

    getchar();
    return 0;
}

```

The above program prints 4, 5 and 8.

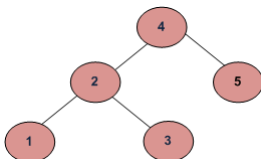
Time Complexity: $O(n)$ where n is number of nodes in the given binary tree.

Please write comments if you find the above code/algorithm incorrect, or find better ways to solve the same problem.

30. Sorted order printing of a given array that represents a BST

Given an array that stores a complete Binary Search Tree, write a function that efficiently prints the given array in ascending order.

For example, given an array [4, 2, 5, 1, 3], the function should print 1, 2, 3, 4, 5



Solution:

Inorder traversal of BST prints it in ascending order. The only trick is to modify recursion termination condition in [standard Inorder Tree Traversal](#).

Implementation:

```
#include<stdio.h>

void printSorted(int arr[], int start, int end)
{
    if(start > end)
        return;

    // print left subtree
    printSorted(arr, start*2 + 1, end);

    // print root
    printf("%d ", arr[start]);

    // print right subtree
    printSorted(arr, start*2 + 2, end);
}

int main()
{
    int arr[] = {4, 2, 5, 1, 3};
    int arr_size = sizeof(arr)/sizeof(int);
    printSorted(arr, 0, arr_size-1);
    getchar();
    return 0;
}
```

Time Complexity: $O(n)$

Please write comments if you find the above solution incorrect, or find better ways to solve the same problem.

31. Applications of tree data structure

Difficulty Level: Rookie

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  
>
```

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 $O(\log n)$ 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 $O(\log n)$ 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

References:

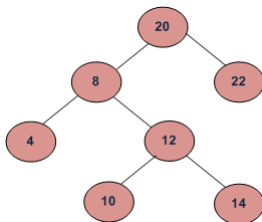
<http://www.cs.bu.edu/teaching/c/tree/binary/>

http://en.wikipedia.org/wiki/Tree_%28data_structure%29#Common_uses

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

32. 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 Inorder 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 subtree 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 its 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.

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
```

```

/* Given a non-empty binary search tree, return the minimum data
   value found in that tree. Note that the entire tree does not need
   to be searched. */
struct node * minValue(struct node* node) {
    struct node * current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL) {
        current = current->left;
    }
    return current;
}

/* Helper function that allocates a new node with the given data and
   NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    node->parent = NULL;

    return(node);
}

/* Give a binary search tree and a number, inserts a new node with
   the given number in the correct place in the tree. Returns the new
   root pointer which the caller should then use (the standard trick to
   avoid using reference parameters). */
struct node* insert(struct node* node, int data)
{
    /* 1. If the tree is empty, return a new,
       single node */

```

```

if (node == NULL)
    return(newNode(data));
else
{
    struct node *temp;

    /* 2. Otherwise, recur down the tree */
    if (data <= node->data)
    {
        temp = insert(node->left, data);
        node->left = temp;
        temp->parent= node;
    }
    else
    {
        temp = insert(node->right, data);
        node->right = temp;
        temp->parent = node;
    }

    /* return the (unchanged) node pointer */
    return node;
}
}

/* Driver program to test above functions*/
int main()
{
    struct node* root = NULL, *temp, *succ, *min;

    //creating the tree given in the above diagram
    root = insert(root, 20);
    root = insert(root, 8);
    root = insert(root, 22);
    root = insert(root, 4);
    root = insert(root, 12);
    root = insert(root, 10);
    root = insert(root, 14);
    temp = root->left->right->right;

    succ = inOrderSuccessor(root, temp);
    if(succ != NULL)
        printf("\n Inorder Successor of %d is %d ", temp->data, succ->data)
    else
        printf("\n Inorder Successor doesn't exist");

    getchar();
    return 0;
}

```

Output of the above program:

Inorder Successor of 14 is 20

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 subtree of *node* is *NULL*, then start from *root* and use 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.

```
struct node * inOrderSuccessor(struct node *root, struct node *n)
{
    // step 1 of the above algorithm
    if( n->right != NULL )
        return minValue(n->right);

    struct node *succ = NULL;

    // Start from root and search for successor down the tree
    while (root != NULL)
    {
        if (n->data < root->data)
        {
            succ = root;
            root = root->left;
        }
        else if (n->data > root->data)
            root = root->right;
        else
            break;
    }

    return succ;
}
```

Thanks to [R.Srinivasan](#) for suggesting this method.

Time Complexity: $O(h)$ where h is height of tree.

References:

<http://net.pku.edu.cn/~course/cs101/2007/resource/Intro2Algorithm/book6/chap13.htm>

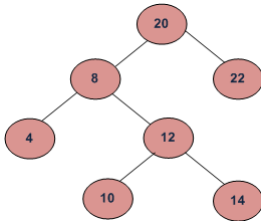
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

33. 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
```

Implementation:

```
#include <stdio.h>
#include <stdlib.h>

#define ARRAY_SIZE(arr) sizeof(arr)/sizeof(arr[0])

/* just add elements to test */
/* NOTE: A sorted array results in skewed tree */
int ele[] = { 20, 8, 22, 4, 12, 10, 14 };

/* ----- */
```

```

/* same alias */
typedef struct node_t node_t;

/* Binary tree node */
struct node_t
{
    int data;

    node_t* left;
    node_t* right;
};

/* simple stack that stores node addresses */
typedef struct stack_t stack_t;

/* initial element always NULL, uses as sentinel */
struct stack_t
{
    node_t*   base[ARRAY_SIZE(ele) + 1];
    int       stackIndex;
};

/* pop operation of stack */
node_t *pop(stack_t *st)
{
    node_t *ret = NULL;

    if( st && st->stackIndex > 0 )
    {
        ret = st->base[st->stackIndex];
        st->stackIndex--;
    }

    return ret;
}

/* push operation of stack */
void push(stack_t *st, node_t *node)
{
    if( st )
    {
        st->stackIndex++;
        st->base[st->stackIndex] = node;
    }
}

/* Iterative insertion
   Recursion is least preferred unless we gain something
*/
node_t *insert_node(node_t *root, node_t* node)
{
    /* A crawling pointer */
    node_t *pTraverse = root;
    node_t *currentParent = root;

    // Traverse till appropriate node
    while(pTraverse)
    {
        currentParent = pTraverse;

        if( node->data < pTraverse->data )
        {
            /* left subtree */

```

```

        , // left subtree ,
        pTraverse = pTraverse->left;
    }
    else
    {
        /* right subtree */
        pTraverse = pTraverse->right;
    }
}

/* If the tree is empty, make it as root node */
if( !root )
{
    root = node;
}
else if( node->data < currentParent->data )
{
    /* Insert on left side */
    currentParent->left = node;
}
else
{
    /* Insert on right side */
    currentParent->right = node;
}

return root;
}

```

```

/* Elements are in an array. The function builds binary tree */
node_t* binary_search_tree(node_t *root, int keys[], int const size)
{
    int iterator;
    node_t *new_node = NULL;

    for(iterator = 0; iterator < size; iterator++)
    {
        new_node = (node_t *)malloc( sizeof(node_t) );

        /* initialize */
        new_node->data = keys[iterator];
        new_node->left = NULL;
        new_node->right = NULL;

        /* insert into BST */
        root = insert_node(root, new_node);
    }

    return root;
}

```

```

node_t *k_smallest_element_inorder(stack_t *stack, node_t *root, int k)
{
    stack_t *st = stack;
    node_t *pCrawl = root;

    /* move to left extremen (minimum) */
    while( pCrawl )
    {
        push(st, pCrawl);
        pCrawl = pCrawl->left;
    }
}

```

```

/* pop off stack and process each node */
while( pCrawl = pop(st) )
{
    /* each pop operation emits one element
    in the order
    */
    if( !--k )
    {
        /* loop testing */
        st->stackIndex = 0;
        break;
    }

    /* there is right subtree */
    if( pCrawl->right )
    {
        /* push the left subtree of right subtree */
        pCrawl = pCrawl->right;
        while( pCrawl )
        {
            push(st, pCrawl);
            pCrawl = pCrawl->left;
        }

        /* pop off stack and repeat */
    }
}

/* node having k-th element or NULL node */
return pCrawl;
}

/* Driver program to test above functions */
int main(void)
{
    node_t* root = NULL;
    stack_t stack = { {0}, 0 };
    node_t *kNode = NULL;

    int k = 5;

    /* Creating the tree given in the above diagram */
    root = binary_search_tree(root, ele, ARRAY_SIZE(ele));

    kNode = k_smallest_element_inorder(&stack, root, k);

    if( kNode )
    {
        printf("kth smallest element for k = %d is %d", k, kNode->data)
    }
    else
    {
        printf("There is no such element");
    }

    getchar();
    return 0;
}

```

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 start
stop:
```

Implementation:

```
#include <stdio.h>
#include <stdlib.h>

#define ARRAY_SIZE(arr) sizeof(arr)/sizeof(arr[0])

typedef struct node_t node_t;

/* Binary tree node */
struct node_t
{
    int data;
    int lCount;

    node_t* left;
    node_t* right;
};

/* Iterative insertion
Recursion is least preferred unless we gain something
*/
node_t *insert_node(node_t *root, node_t* node)
{
    /* A crawling pointer */
    node_t *pTraverse = root;
    node_t *currentParent = root;
```

```

// Traverse till appropriate node
while(pTraverse)
{
    currentParent = pTraverse;

    if( node->data < pTraverse->data )
    {
        /* We are branching to left subtree
           increment node count */
        pTraverse->lCount++;
        /* left subtree */
        pTraverse = pTraverse->left;
    }
    else
    {
        /* right subtree */
        pTraverse = pTraverse->right;
    }
}

/* If the tree is empty, make it as root node */
if( !root )
{
    root = node;
}
else if( node->data < currentParent->data )
{
    /* Insert on left side */
    currentParent->left = node;
}
else
{
    /* Insert on right side */
    currentParent->right = node;
}

return root;
}

/* Elements are in an array. The function builds binary tree */
node_t* binary_search_tree(node_t *root, int keys[], int const size)
{
    int iterator;
    node_t *new_node = NULL;

    for(iterator = 0; iterator < size; iterator++)
    {
        new_node = (node_t *)malloc( sizeof(node_t) );

        /* initialize */
        new_node->data = keys[iterator];
        new_node->lCount = 0;
        new_node->left = NULL;
        new_node->right = NULL;

        /* insert into BST */
        root = insert_node(root, new_node);
    }

    return root;
}

```

```

int k_smallest_element(node_t *root, int k)
{
    int ret = -1;

    if( root )
    {
        /* A crawling pointer */
        node_t *pTraverse = root;

        /* Go to k-th smallest */
        while(pTraverse)
        {
            if( (pTraverse->lCount + 1) == k )
            {
                ret = pTraverse->data;
                break;
            }
            else if( k > pTraverse->lCount )
            {
                /* There are less nodes on left subtree
                   Go to right subtree */
                k = k - (pTraverse->lCount + 1);
                pTraverse = pTraverse->right;
            }
            else
            {
                /* The node is on left subtree */
                pTraverse = pTraverse->left;
            }
        }
    }

    return ret;
}

/* Driver program to test above functions */
int main(void)
{
    /* just add elements to test */
    /* NOTE: A sorted array results in skewed tree */
    int ele[] = { 20, 8, 22, 4, 12, 10, 14 };
    int i;
    node_t* root = NULL;

    /* Creating the tree given in the above diagram */
    root = binary_search_tree(root, ele, ARRAY_SIZE(ele));

    /* It should print the sorted array */
    for(i = 1; i <= ARRAY_SIZE(ele); i++)
    {
        printf("\n kth smallest element for k = %d is %d",
            i, k_smallest_element(root, i));
    }

    getchar();
    return 0;
}

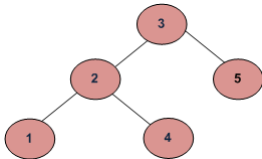
```

Thanks to **Venki** for providing post. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

34. 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.



Thanks to [prandeey](#) for suggesting the following solution.

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.

```
#include<stdio.h>

/* A tree node structure */
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

/* Helper function for getLevel(). It returns level of the data if data
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);
}
```



```

/* Utility function to create a new Binary Tree node */
struct node* newNode(int data)
{
    struct node *temp = new struct node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;

    return temp;
}

/* Driver function to test above functions */
int main()
{
    struct node *root = new struct node;
    int x;

    /* Constructing tree given in the above figure */
    root = newNode(3);
    root->left = newNode(2);
    root->right = newNode(5);
    root->left->left = newNode(1);
    root->left->right = newNode(4);

    for (x = 1; x <=5; x++)
    {
        int level = getLevel(root, x);
        if (level)
            printf(" Level of %d is %d\n", x, getLevel(root, x));
        else
            printf(" %d is not present in tree \n", x);
    }

    getchar();
    return 0;
}

```

Output:

```

Level of 1 is 3
Level of 2 is 2
Level of 3 is 1
Level of 4 is 3
Level of 5 is 2

```

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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

35. 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.



Thanks to [Mike](#) , [Sambasiva](#) and [wgpshashank](#) for their contribution.

```
#include<iostream>
#include<stdio.h>
#include<stdlib.h>
```

```
using namespace std;
```

```
/* A binary tree node has data, pointer to left child
   and a pointer to right child */
```

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

```
/* 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;
}
```

```

}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    /* Construct the following binary tree
        1
       / \
      2   3
     / \
    4   5
   /
  7
    */
    struct node *root = newNode(1);
    root->left      = newNode(2);
    root->right     = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->left->left->left = newNode(7);

    printAncestors(root, 7);

    getch();
    return 0;
}

```

Output:

4 2 1

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

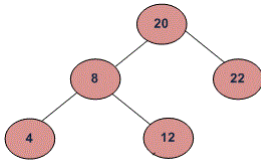
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

36. Print BST keys in the given range

Given two values k_1 and k_2 (where $k_1 < k_2$) 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 $k1 \leq x \leq k2$ 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.



Thanks to [bhasker](#) for suggesting the following solution.

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.

Implementation:

```
#include<stdio.h>

/* A tree node structure */
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

/* The functions prints all the keys which in the given range [k1..k2]
   The function assumes than k1 < k2 */
void Print(struct node *root, int k1, int k2)
{
    /* base case */
    if ( NULL == root )
        return;

    /* Since the desired o/p is sorted, recurse for left subtree first
       If root->data is greater than k1, then only we can get o/p keys
       in left subtree */
    if ( k1 < root->data )
        Print(root->left, k1, k2);

    /* if root's data lies in range, then prints root's data */
    if ( k1 <= root->data && k2 >= root->data )
        printf("%d ", root->data );

    /* If root->data is smaller than k2, then only we can get o/p keys
       in right subtree */
    if ( k2 > root->data )
        Print(root->right, k1, k2);
}

/* Utility function to create a new Binary Tree node */
struct node* newNode(int data)
{

```

```

    struct node *temp = new struct node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;

    return temp;
}

/* Driver function to test above functions */
int main()
{
    struct node *root = new struct node;
    int k1 = 10, k2 = 25;

    /* Constructing tree given in the above figure */
    root = newNode(20);
    root->left = newNode(8);
    root->right = newNode(22);
    root->left->left = newNode(4);
    root->left->right = newNode(12);

    Print(root, k1, k2);

    getchar();
    return 0;
}

```

Output:

12 20 22

Time Complexity: $O(n)$ where n is the total number of keys in tree.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

37. Tournament Tree (Winner Tree) and Binary Heap

Given a team of N players. How many minimum games are required to find second best player?

We can use adversary arguments based on tournament tree (Binary Heap).

Tournament tree is a form of min (max) heap which is a complete binary tree. Every external node represents a player and internal node represents winner. In a tournament tree every internal node contains winner and every leaf node contains one player.

There will be $N - 1$ internal nodes in a binary tree with N leaf (external) nodes. For details see [this post](#) (put $n = 2$ in equation given in the post).

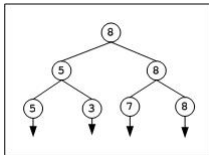
It is obvious that to select the best player among N players, $(N - 1)$ players to

be eliminated, i.e. we need minimum of $(N - 1)$ games (comparisons). Mathematically we can prove it. In a binary tree $I = E - 1$, where I is number of internal nodes and E is number of external nodes. It means to find maximum or minimum element of an array, we need $N - 1$ (internal nodes) comparisons.

Second Best Player

The information explored during best player selection can be used to minimize the number of comparisons in tracing the next best players. For example, we can pick second best player in $(N + \log_2 N - 2)$ comparisons. For details read [this comment](#).

The following diagram displays a tournament tree (*winner tree*) as a max heap. Note that the concept of *loser tree* is different.



The above tree contains 4 leaf nodes that represent players and have 3 levels 0, 1 and 2. Initially 2 games are conducted at level 2, one between 5 and 3 and another one between 7 and 8. In the next move, one more game is conducted between 5 and 8 to conclude the final winner. Overall we need 3 comparisons. For second best player we need to trace the candidates participated with final winner, that leads to 7 as second best.

Median of Sorted Arrays

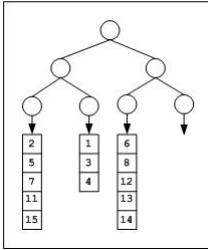
Tournament tree can effectively be used to find median of sorted arrays. Assume, given M sorted arrays of equal size L (for simplicity). We can attach all these sorted arrays to the tournament tree, one array per leaf. We need a tree of height **CEIL** ($\log_2 M$) to have atleast M external nodes.

Consider an example. Given 3 ($M = 3$) sorted integer arrays of maximum size 5 elements.

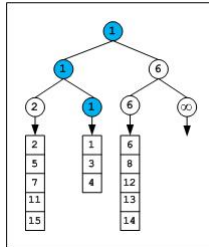
```

{ 2, 5, 7, 11, 15 } ---- Array1
{1, 3, 4} ---- Array2
{6, 8, 12, 13, 14} ---- Array3
  
```

What should be the height of tournament tree? We need to construct a tournament tree of height $\log_2 3 \approx 1.585 = 2$ rounded to next integer. A binary tree of height 2 will have 4 leaves to which we can attach the arrays as shown in the below figure.



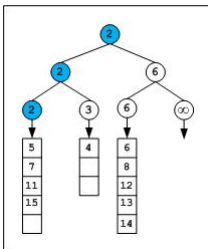
After the first tournament, the tree appears as below,



We can observe that the winner is from Array2. Hence the next element from Array2 will dive-in and games will be played along the winner path of previous tournament.

Note that infinity is used as sentinel element. Based on data being hold in nodes we can select the sentinel character. For example we usually store the pointers in nodes rather than keys, so NULL can serve as sentinel. If any of the array exhausts we will fill the corresponding leaf and upcoming internal nodes with sentinel.

After the second tournament, the tree appears as below,



The next winner is from Array1, so next element of Array1 array which is 5 will dive-in to the next round, and next tournament played along the path of 2.

The tournaments can be continued till we get median element which is $(5+3+5)/2 = 7$ th element. Note that there are even better algorithms for finding median of union of sorted arrays, for details see the related links given below.

In general with M sorted lists of size $L_1, L_2 \dots L_m$ requires time complexity of $O((L_1 + L_2 + \dots + L_m) * \log M)$ to merge all the arrays, and $O(m * \log M)$ time to find median, where

m is median position.

Select smallest one million elements from one billion unsorted elements:

As a simple solution, we can sort the billion numbers and select first one million.

On a limited memory system sorting billion elements and picking the first one million seems to be impractical. We can use tournament tree approach. At any time only elements of tree to be in memory.

Split the large array (perhaps stored on disk) into smaller size arrays of size one million each (or even smaller that can be sorted by the machine). Sort these 1000 small size arrays and store them on disk as individual files. Construct a tournament tree which can have atleast 1000 leaf nodes (tree to be of height 10 since $2^9 < 1000 < 2^{10}$, if the individual file size is even smaller we will need more leaf nodes). Every leaf node will have an engine that picks next element from the sorted file stored on disk. We can play the tournament tree game to extract first one million elements.

Total cost = sorting 1000 lists of one million each + tree construction + tournaments

Implementation

We need to build the tree (heap) in bottom-up manner. All the leaf nodes filled first. Start at the left extreme of tree and fill along the breadth (i.e. from 2^{k-1} to $2^k - 1$ where k is depth of tree) and play the game. After practicing with few examples it will be easy to write code. We will have code in an upcoming article.

Related Posts

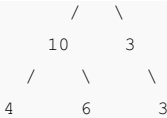
[Link 1](#), [Link 2](#), [Link 3](#), [Link 4](#), [Link 5](#), [Link 6](#), [Link 7](#).

— by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

38. 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.

```

#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, left child and right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* A utility function to get the sum of values in tree with root
as root */
int sum(struct node *root)
{
    if(root == NULL)
        return 0;
    return sum(root->left) + root->data + sum(root->right);
}

/* returns 1 if sum property holds for the given
node and both of its children */
int isSumTree(struct node* node)
{
    int ls, rs;

    /* If node is NULL or it's a leaf node then
return true */
    if(node == NULL ||
        (node->left == NULL && node->right == NULL))
        return 1;

    /* Get sum of nodes in left and right subtrees */
    ls = sum(node->left);
    rs = sum(node->right);

    /* if the node and both of its children satisfy the
property return 1 else 0*/
    if((node->data == ls + rs)&&
        isSumTree(node->left) &&
        isSumTree(node->right))
        return 1;

    return 0;
}

/*
Helper function that allocates a new node
with the given data and NULL left and right
pointers.
*/

```

```

struct node* newNode(int data)
{
    struct node* node =
        (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

/* Driver program to test above function */
int main()
{
    struct node *root = newNode(26);
    root->left = newNode(10);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(6);
    root->right->right = newNode(3);
    if(isSumTree(root))
        printf("The given tree is a SumTree ");
    else
        printf("The given tree is not a SumTree ");

    getchar();
    return 0;
}

```

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).

```

#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, left child and right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Utility function to check if the given node is leaf or not */
int isLeaf(struct node *node)
{
    if(node == NULL)
        return 0;
    if(node->left == NULL && node->right == NULL)
        return 1;
    return 0;
}

/* returns 1 if SumTree property holds for the given

```

```

/* 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;
}

/* Helper function that allocates a new node
   with the given data and NULL left and right
   pointers.
   */
struct node* newNode(int data)
{
    struct node* node =
        (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

/* Driver program to test above function */
int main()
{
    struct node *root = newNode(26);
    root->left = newNode(10);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(6);
    root->right->right = newNode(3);
    if(isSumTree(root))
        printf("The given tree is a SumTree ");
}

```

```

    else
        printf("The given tree is not a SumTree ");

    getchar();
    return 0;
}

```

Time Complexity: $O(n)$

Please write comments if you find the above codes/algorithms incorrect, or find other ways to solve the same problem.

39. Decision Trees – Fake (Counterfeit) Coin Puzzle (12 Coin Puzzle)

Let us solve the classic “fake coin” puzzle using decision trees. There are the two different variants of the puzzle given below. I am providing description of both the puzzles below, try to solve on your own, assume $N = 8$.

Easy: Given a two pan fair balance and N identically looking coins, out of which only one coin is **lighter (or heavier)**. To figure out the odd coin, how many minimum number of weighing are required in the worst case?

Difficult: Given a two pan fair balance and N identically looking coins out of which only one coin **may be** defective. How can we trace which coin, if any, is odd one and also determine whether it is lighter or heavier in minimum number of trials in the worst case?

Let us start with relatively simple examples. After reading every problem try to solve on your own.

Problem 1: (Easy)

Given 5 coins out of which one coin is **lighter**. In the worst case, how many minimum number of weighing are required to figure out the odd coin?

Name the coins as 1, 2, 3, 4 and 5. We know that one coin is lighter. Considering best out come of balance, we can group the coins in two different ways, [(1, 2), (3, 4) and (5)], or [(12), (34) and (5)]. We can easily rule out groups like [(123) and (45)], as we will get obvious answer. Any other combination will fall into one of these two groups, like [(2) (45) and (13)], etc.

Consider the first group, pairs (1, 2) and (3, 4). We can check (1, 2), if they are equal we go ahead with (3, 4). We need two weighing in worst case. The same analogy can be applied when the coin is heavier.

With the second group, weigh (12) and (34). If they balance (5) is defective one,

otherwise pick the lighter pair, and we need one more weighing to find odd one.

Both the combinations need two weighing in case of 5 coins with prior information of one coin is lighter.

Analysis: In general, if we know that the coin is heavy or light, we can trace the coin in $\log_3(N)$ trials (rounded to next integer). If we represent the outcome of balance as ternary tree, every leaf represent an outcome. Since any coin among N coins can be defective, we need to get a 3-ary tree having minimum of N leaves. A 3-ary tree at k -th level will have 3^k leaves and hence we need $3^k \geq N$.

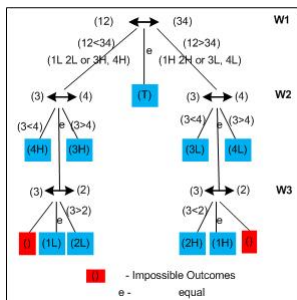
In other-words, in k trials we can examine upto 3^k coins, if we know whether the defective coin is heavier or lighter. Given that a coin is heavier, verify that 3 trials are sufficient to find the odd coin among 12 coins, because $3^2 < 12 < 3^3$.

Problem 2: (Difficult)

*We are given 4 coins, out of which only one coin **may be** defective. We don't know, whether all coins are genuine or any defective one is present. How many number of weighing are required in worst case to figure out the odd coin, if present? We also need to tell whether it is heavier or lighter.*

From the above analysis we may think that $k = 2$ trials are sufficient, since a two level 3-ary tree yields 9 leaves which is greater than $N = 4$ (read the problem once again). Note that it is impossible to solve above 4 coins problem in two weighing. The decision tree confirms the fact (try to draw).

We can group the coins in two different ways, $[(12, 34)]$ or $[(1, 2) \text{ and } (3, 4)]$. Let us consider the combination $(12, 34)$, the corresponding decision tree is given below. Blue leaves are valid outcomes, and red leaves are impossible cases. We arrived at impossible cases due to the assumptions made earlier on the path.



The outcome can be $(12) < (34)$ i.e. we go on to left subtree or $(12) > (34)$ i.e. we go on to right subtree.

The left subtree is possible in two ways,

- A) Either 1 or 2 can be lighter OR
- B) Either 3 or 4 can be heavier.

Further on the left subtree, as second trial, we weigh (1, 2) or (3, 4). Let us consider (3, 4) as the analogy for (1, 2) is similar. The outcome of second trail can be three ways

- A) $(3) < (4)$ yielding 4 as defective heavier coin, OR
- B) $(3) > (4)$ yielding 3 as defective heavier coin OR
- C) $(3) = (4)$, yielding ambiguity. Here we need one more weighing to check a genuine coin against 1 or 2. In the figure I took (3, 2) where 3 is confirmed as genuine. We can get $(3) > (2)$ in which 2 is lighter, or $(3) = (2)$ in which 1 is lighter. Note that it impossible to get $(3) < (2)$, it contradicts our assumption leaned to left side.

Similarly we can analyze the right subtree. We need two more weighings on right subtree as well.

Overall we need 3 weighings to trace the odd coin. Note that we are unable to utilize two outcomes of 3-ary trees. Also, the tree is not full tree, middle branch terminated after first weighing. Infact, we can get 27 leaves of 3 level full 3-ary tree, but only we got 11 leaves including impossible cases.

Analysis: Given N coins, all may be genuine or only one coin is defective. We need a decision tree with atleast $(2N + 1)$ leaves correspond to the outputs. Because there can be N leaves to be lighter, or N leaves to be heavier or one genuine case, on total $(2N + 1)$ leaves.

As explained earlier ternary tree at level k , can have utmost 3^k leaves and we need a tree with leaves of $3^k > (2N + 1)$.

In other words, we need atleast $k > \log_3(2N + 1)$ weighing to find the defective one.

Observe the above figure that not all the branches are generating leaves, i.e. we are missing valid outputs under some branches that leading to more number of trials. When possible, we should group the coins in such a way that every branch is going to yield valid output (in simple terms generate full 3-ary tree). Problem 4 describes this approach of 12 coins.

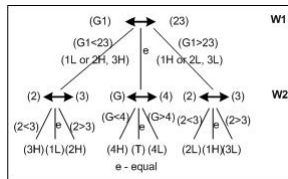
Problem 3: (Special case of two pan balance)

*We are given 5 coins, a group of 4 coins out of which one coin is defective (we **don't know** whether it is heavier or lighter), and one coin is genuine. How many weighing are required in worst case to figure out the odd coin whether it is heavier or lighter?*

Label the coins as 1, 2, 3, 4 and G (genuine). We now have some information on coin purity. We need to make use that in the groupings.

We can best group them as $[(G1, 23) \text{ and } (4)]$. Any other group can't generate full 3-ary

tree, try yourself. The following diagram explains the procedure.



The middle case $(G1) = (23)$ is self explanatory, i.e. 1, 2, 3 are genuine and 4th coin can be figured out lighter or heavier in one more trial.

The left side of tree corresponds to the case $(G1) < (23)$. This is possible in two ways, either 1 should be lighter or either of (2, 3) should be heavier. The former instance is obvious when next weighing (2, 3) is balanced, yielding 1 as lighter. The later instance could be $(2) < (3)$ yielding 3 as heavier or $(2) > (3)$ yielding 2 as heavier. The leaf nodes on left branch are named to reflect these outcomes.

The right side of tree corresponds to the case $(G1) > (23)$. This is possible in two ways, either 1 is heavier or either of (2, 3) should be lighter. The former instance is obvious when the next weighing (2, 3) is balanced, yielding 1 as heavier. The later case could be $(2) < (3)$ yielding 2 as lighter coin, or $(2) > (3)$ yielding 3 as lighter.

In the above problem, under any possibility we need only two weighing. We are able to use all outcomes of two level full 3-ary tree. We started with $(N + 1) = 5$ coins where $N = 4$, we end up with $(2N + 1) = 9$ leaves. ***Infact we should have 11 outcomes since we started with 5 coins, where are other 2 outcomes? These two outcomes can be declared at the root of tree itself (prior to first weighing), can you figure these two out comes?***

If we observe the figure, after the first weighing the problem reduced to “we know three coins, either one can be lighter (heavier) or one among other two can be heavier (lighter)”. This can be solved in one weighing (read Problem 1).

Analysis: Given $(N + 1)$ coins, one is genuine and the rest N can be genuine or only one coin is defective. The required decision tree should result in minimum of $(2N + 1)$ leaves. Since the total possible outcomes are $(2(N + 1) + 1)$, number of weighing (trials) are given by the height of ternary tree, $k \geq \log_3[2(N + 1) + 1]$. *Note the equality sign.*

Rearranging k and N , we can weigh maximum of $N \leq (3^k - 3)/2$ coins in k trials.

Problem 4: (The classic 12 coin puzzle)

You are given two pan fair balance. You have 12 identically looking coins out of which one coin may be lighter or heavier. How can you find odd coin, if any, in minimum trials, also determine whether defective coin is lighter or heavier, in the worst case?

How do you want to group them? Bi-set or tri-set? Clearly we can discard the option of

dividing into two equal groups. It can't lead to best tree. *From the above two examples, we can ensure that the decision tree can be used in optimal way if we can reveal atleast one genuine coin.* Remember to group coins such that the first weighing reveals atleast one genuine coin.

Let us name the coins as 1, 2, ... 8, A, B, C and D. We can combine the coins into 3 groups, namely (1234), (5678) and (ABCD). Weigh (1234) and (5678). You are encouraged to draw decision tree while reading the procedure. The outcome can be three ways,

1. (1234) = (5678), both groups are equal. Defective coin may be in (ABCD) group.
2. (1234) < (5678), i.e. first group is less in weight than second group.
3. (1234) > (5678), i.e. first group is more in weight than second group.

The output (1) can be solved in two more weighing as special case of two pan balance given in Problem 3. We know that groups (1234) and (5678) are genuine and defective coin may be in (ABCD). Pick one genuine coin from any of weighed groups, and proceed with (ABCD) as explained in Problem 3.

Outcomes (2) and (3) are special. In both the cases, we know that (ABCD) is genuine. And also, we know a set of coins being lighter and a set of coins being heavier. We need to shuffle the weighed two groups in such a way that we end up with smaller height decision tree.

Consider the second outcome where (1234) < (5678). It is possible when any coin among (1, 2, 3, 4) is lighter or any coin among (5, 6, 7, 8) is heavier. We revealed lighter or heavier possibility after first weighing. If we proceed as in Problem 1, we will not generate best decision tree. Let us shuffle coins as (1235) and (4BCD) as new groups (there are different shuffles possible, they also lead to minimum weighing, can you try?). If we weigh these two groups again the outcome can be three ways, i) (1235) < (4BCD) yielding one among 1, 2, 3 is lighter which is similar to Problem 1 explained above, we need one more weighing, ii) (1235) = (4BCD) yielding one among 6, 7, 8 is heavier which is similar to Problem 1 explained above, we need one more weighing iii) (1235) > (4BCD) yielding either 5 as heavier coin or 4 as lighter coin, at the expense of one more weighing.

Similar way we can also solve the right subtree (third outcome where (1234) > (5678)) in two more weighing.

We are able to solve the 12 coin puzzle in 3 weighing in the worst case.

Few Interesting Puzzles:

1. Solve Problem 4 with $N = 8$ and $N = 13$, How many minimum trials are required in each case?
2. Given a function `int weigh(A[], B[])` where A and B are arrays (need not be equal size). The function returns -1, 0 or 1. It returns 0 if sum of all elements in A and B are equal, -1 if $A < B$ and 1 if $A > B$. Given an array of 12 elements, all elements are

equal except one. The odd element can be as that of others, smaller or greater than others. Write a program to find the odd element (if any) using *weigh()* minimum number of times.

3. You might have seen 3-pan balance in science labs during school days. Given a 3-pan balance (4 outcomes) and N coins, how many minimum trials are needed to figure out odd coin?

References:

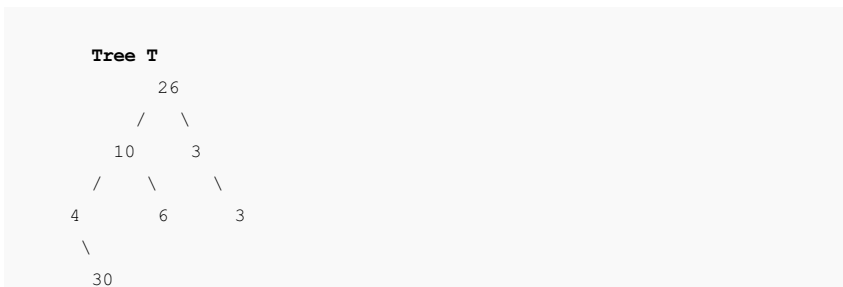
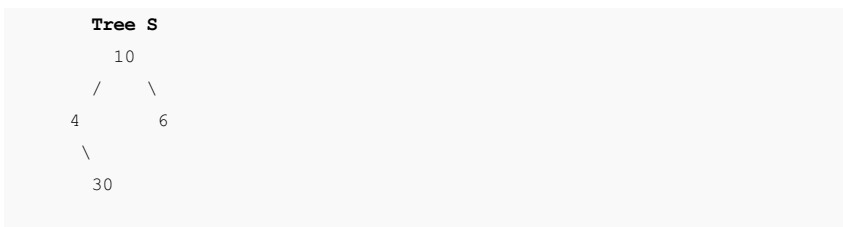
Similar problem was provided in one of the exercises of the book "Introduction to Algorithms by Levitin". Specifically read section 5.5 and section 11.2 including exercises.

— — — by **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

40. Check if a binary tree is subtree of another binary tree | Set 1

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.



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.

Following is C implementation for this.

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, left child and right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* A utility function to check whether trees with roots as root1 and
root2 are identical or not */
bool areIdentical(struct node * root1, struct node *root2)
{
    /* base cases */
    if (root1 == NULL && root2 == NULL)
        return true;

    if (root1 == NULL || root2 == NULL)
        return false;

    /* Check if the data of both roots is same and data of left and right
subtrees are also same */
    return (root1->data == root2->data &&
        areIdentical(root1->left, root2->left) &&
        areIdentical(root1->right, root2->right) );
}

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

    if (T == NULL)
        return false;

    /* Check the tree with root as current node */
    if (areIdentical(T, S))
        return true;

    /* If the tree with root as current node doesn't match then
try left and right subtrees one by one */
    return isSubtree(T->left, S) ||
        isSubtree(T->right, S);
}

/* Helper function that allocates a new node with the given data
and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node =
        (struct node*)malloc(sizeof(struct node));
    node->data = data;
}
```

```

    node->left = NULL;
    node->right = NULL;
    return(node);
}

/* Driver program to test above function */
int main()
{
    /* Construct the following tree
        26
       / \
      10  3
     / \  \
    4   6  3
   / \
  30
    */
    struct node *T = newNode(26);
    T->right = newNode(3);
    T->right->right = newNode(3);
    T->left = newNode(10);
    T->left->left = newNode(4);
    T->left->left->right = newNode(30);
    T->left->right = newNode(6);

    /* Construct the following tree
        10
       / \
      4   6
     / \
    30
    */
    struct node *S = newNode(10);
    S->right = newNode(6);
    S->left = newNode(4);
    S->left->right = newNode(30);

    if (isSubtree(T, S))
        printf("Tree S is subtree of tree T");
    else
        printf("Tree S is not a subtree of tree T");

    getchar();
    return 0;
}

```

Output:

```
Tree S is subtree of tree T
```

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. Please refer [Check if a binary tree is subtree of another binary tree | Set 2](#) for $O(n)$ solution.

Please write comments if you find anything incorrect, or you want to share more

information about the topic discussed above.

41. Trie | (Insert and Search)

Trie is an efficient information **retrieval** data structure. Using trie, search complexities can be brought to optimal limit (key length). If we store keys in binary search tree, a well balanced BST will need time proportional to $M * \log N$, where M is maximum string length and N is number of keys in tree. Using trie, we can search the key in $O(M)$ time. However the penalty is on trie storage requirements.

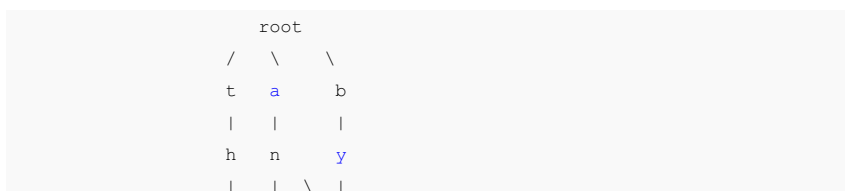
Every node of trie consists of multiple branches. Each branch represents a possible character of keys. We need to mark the last node of every key as leaf node. A trie node field *value* will be used to distinguish the node as leaf node (there are other uses of the *value* field). A simple structure to represent nodes of English alphabet can be as following,

```
struct trie_node
{
    int value; /* Used to mark leaf nodes */
    trie_node_t *children[ALPHABET_SIZE];
};
```

Inserting a key into trie is simple approach. Every character of input key is inserted as an individual trie node. Note that the *children* is an array of pointers to next level trie nodes. The key character acts as an index into the array *children*. If the input key is new or an extension of existing key, we need to construct non-existing nodes of the key, and mark leaf node. If the input key is prefix of existing key in trie, we simply mark the last node of key as leaf. The key length determines trie depth.

Searching for a key is similar to insert operation, however we only compare the characters and move down. The search can terminate due to end of string or lack of key in trie. In the former case, if the *value* field of last node is non-zero then the key exists in trie. In the second case, the search terminates without examining all the characters of key, since the key is not present in trie.

The following picture explains construction of trie using keys given in the example below,



```

      e   s   y   e
    /   |   |
   i   r   w
   |   |   |
   r   e   e
       |
       r

```

In the picture, every character is of type *trie_node_t*. For example, the *root* is of type *trie_node_t*, and it's children *a*, *b* and *t* are filled, all other nodes of root will be NULL. Similarly, “a” at the next level is having only one child (“n”), all other children are NULL. The leaf nodes are in [blue](#).

Insert and search costs **$O(\text{key_length})$** , however the memory requirements of trie is **$O(\text{ALPHABET_SIZE} * \text{key_length} * N)$** where N is number of keys in trie. There are efficient representation of trie nodes (e.g. compressed trie, [ternary search tree](#), etc.) to minimize memory requirements of trie.

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define ARRAY_SIZE(a) sizeof(a)/sizeof(a[0])

// Alphabet size (# of symbols)
#define ALPHABET_SIZE (26)
// Converts key current character into index
// use only 'a' through 'z' and lower case
#define CHAR_TO_INDEX(c) ((int)c - (int)'a')

// trie node
typedef struct trie_node trie_node_t;
struct trie_node
{
    int value;
    trie_node_t *children[ALPHABET_SIZE];
};

// trie ADT
typedef struct trie trie_t;
struct trie
{
    trie_node_t *root;
    int count;
};

// Returns new trie node (initialized to NULLs)
trie_node_t *getNode(void)
{
    trie_node_t *pNode = NULL;

    pNode = (trie_node_t *)malloc(sizeof(trie_node_t));

    if( pNode )
    {
        int i;

```

```

        pNode->value = 0;

        for(i = 0; i < ALPHABET_SIZE; i++)
        {
            pNode->children[i] = NULL;
        }
    }

    return pNode;
}

// Initializes trie (root is dummy node)
void initialize(trie_t *pTrie)
{
    pTrie->root = getNode();
    pTrie->count = 0;
}

// If not present, inserts key into trie
// If the key is prefix of trie node, just marks leaf node
void insert(trie_t *pTrie, char key[])
{
    int level;
    int length = strlen(key);
    int index;
    trie_node_t *pCrawl;

    pTrie->count++;
    pCrawl = pTrie->root;

    for( level = 0; level < length; level++ )
    {
        index = CHAR_TO_INDEX(key[level]);
        if( !pCrawl->children[index] )
        {
            pCrawl->children[index] = getNode();
        }

        pCrawl = pCrawl->children[index];
    }

    // mark last node as leaf
    pCrawl->value = pTrie->count;
}

// Returns non zero, if key presents in trie
int search(trie_t *pTrie, char key[])
{
    int level;
    int length = strlen(key);
    int index;
    trie_node_t *pCrawl;

    pCrawl = pTrie->root;

    for( level = 0; level < length; level++ )
    {
        index = CHAR_TO_INDEX(key[level]);

        if( !pCrawl->children[index] )
        {
            return 0;
        }
    }
}

```

```

    }

    pCrawl = pCrawl->children[index];
}

return (0 != pCrawl && pCrawl->value);
}

// Driver
int main()
{
    // Input keys (use only 'a' through 'z' and lower case)
    char keys[][8] = {"the", "a", "there", "answer", "any", "by", "bye"};
    trie_t trie;

    char output[][32] = {"Not present in trie", "Present in trie"};

    initialize(&trie);

    // Construct trie
    for(int i = 0; i < ARRAY_SIZE(keys); i++)
    {
        insert(&trie, keys[i]);
    }

    // Search for different keys
    printf("%s --- %s\n", "the", output[search(&trie, "the")] );
    printf("%s --- %s\n", "these", output[search(&trie, "these")] );
    printf("%s --- %s\n", "their", output[search(&trie, "their")] );
    printf("%s --- %s\n", "thaw", output[search(&trie, "thaw")] );

    return 0;
}

```

— **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

42. Trie | (Delete)

In the [previous post](#) on [trie](#) we have described how to insert and search a node in trie. Here is an algorithm how to delete a node from trie.

During delete operation we delete the key in bottom up manner using recursion. The following are possible conditions when deleting key from trie,

1. Key may not be there in trie. Delete operation should not modify trie.
2. Key present as unique key (no part of key contains another key (prefix), nor the key itself is prefix of another key in trie). Delete all the nodes.
3. Key is prefix key of another long key in trie. Unmark the leaf node.
4. Key present in trie, having atleast one other key as prefix key. Delete nodes from

end of key until first leaf node of longest prefix key.

The highlighted code presents algorithm to implement above conditions. (One may be in dilemma how a pointer passed to delete helper is reflecting changes from deleteHelper to deleteKey. Note that we are holding trie as an ADT in trie_t node, which is passed by reference or pointer).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define ARRAY_SIZE(a) sizeof(a)/sizeof(a[0])

// Alphabet size (# of symbols)

#define ALPHABET_SIZE (26)
#define INDEX(c) ((int)c - (int)'a')

#define FREE(p) \
    free(p); \
    p = NULL;

// forward declration
typedef struct trie_node trie_node_t;

// trie node
struct trie_node
{
    int value; // non zero if leaf
    trie_node_t *children[ALPHABET_SIZE];
};

// trie ADT
typedef struct trie trie_t;

struct trie
{
    trie_node_t *root;
    int count;
};

trie_node_t *getNode(void)
{
    trie_node_t *pNode = NULL;

    pNode = (trie_node_t *)malloc(sizeof(trie_node_t));

    if( pNode )
    {
        int i;

        pNode->value = 0;

        for(i = 0; i < ALPHABET_SIZE; i++)
        {
            pNode->children[i] = NULL;
        }
    }

    return pNode;
}
```



```

,

void initialize(trie_t *pTrie)
{
    pTrie->root = getNode();
    pTrie->count = 0;
}

void insert(trie_t *pTrie, char key[])
{
    int level;
    int length = strlen(key);
    int index;
    trie_node_t *pCrawl;

    pTrie->count++;
    pCrawl = pTrie->root;

    for( level = 0; level < length; level++ )
    {
        index = INDEX(key[level]);

        if( pCrawl->children[index] )
        {
            // Skip current node
            pCrawl = pCrawl->children[index];
        }
        else
        {
            // Add new node
            pCrawl->children[index] = getNode();
            pCrawl = pCrawl->children[index];
        }
    }

    // mark last node as leaf (non zero)
    pCrawl->value = pTrie->count;
}

int search(trie_t *pTrie, char key[])
{
    int level;
    int length = strlen(key);
    int index;
    trie_node_t *pCrawl;

    pCrawl = pTrie->root;

    for( level = 0; level < length; level++ )
    {
        index = INDEX(key[level]);

        if( !pCrawl->children[index] )
        {
            return 0;
        }

        pCrawl = pCrawl->children[index];
    }

    return (0 != pCrawl && pCrawl->value);
}

```

```

int leafNode(trie_node_t *pNode)
{
    return (pNode->value != 0);
}

int isItFreeNode(trie_node_t *pNode)
{
    int i;
    for(i = 0; i < ALPHABET_SIZE; i++)
    {
        if( pNode->children[i] )
            return 0;
    }

    return 1;
}

```

```

bool deleteHelper(trie_node_t *pNode, char key[], int level, int len)
{
    if( pNode )
    {
        // Base case
        if( level == len )
        {
            if( pNode->value )
            {
                // Unmark leaf node
                pNode->value = 0;

                // If empty, node to be deleted
                if( isItFreeNode(pNode) )
                {
                    return true;
                }

                return false;
            }
        }
        else // Recursive case
        {
            int index = INDEX(key[level]);

            if( deleteHelper(pNode->children[index], key, level+1, len) )
            {
                // last node marked, delete it
                FREE(pNode->children[index]);

                // recursively climb up, and delete eligible nodes
                return ( !leafNode(pNode) && isItFreeNode(pNode) );
            }
        }
    }

    return false;
}

```

```

void deleteKey(trie_t *pTrie, char key[])
{
    int len = strlen(key);

    if( len > 0 )
    {
        deleteHelper(pTrie->root, key, 0, len);
    }
}

```

```

        deleteHelper(ptrie->root, key, 0, len);
    }
}

int main()
{
    char keys[][8] = {"she", "sells", "sea", "shore", "the", "by", "sh"};
    trie_t trie;

    initialize(&trie);

    for(int i = 0; i < ARRAY_SIZE(keys); i++)
    {
        insert(&trie, keys[i]);
    }

    deleteKey(&trie, keys[0]);

    printf("%s %s\n", "she", search(&trie, "she") ? "Present in trie"
    : "Not Present in trie");

    return 0;
}

```

— **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

43. 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.

```

#include <stdio.h>
#include <stdlib.h>

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

void connectRecur(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.

```

```

Assumption: p is a complete 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);
}

```

```

/* UTILITY FUNCTIONS */
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    node->nextRight = NULL;

    return(node);
}

```

```

/* Driver program to test above functions*/
int main()
{

```

```

    /* Constructed binary tree is

```

```

        10
       /  \
      8    2
     /
    3
    */

```

```

    struct node *root = newNode(10);
    root->left = newNode(8);
    root->right = newNode(2);
    root->left->left = newNode(3);

```

```

    // Populates nextRight pointer in all nodes
    connect(root);

```

```

    // Let us check the values of nextRight pointers
    printf("Following are populated nextRight pointers in the tree "
           "(-1 is printed if there is no nextRight) \n");
    printf("nextRight of %d is %d \n", root->data,
           root->nextRight? root->nextRight->data: -1);
    printf("nextRight of %d is %d \n", root->left->data,
           root->left->nextRight? root->left->nextRight->data: -1);
    printf("nextRight of %d is %d \n", root->right->data,
           root->right->nextRight? root->right->nextRight->data: -1);

```

```

print( nextRight of %d is %d \n", root->right->data,
      root->right->nextRight? root->right->nextRight->data: -1);
printf("nextRight of %d is %d \n", root->left->left->data,
      root->left->left->nextRight? root->left->left->nextRight->data

getchar();
return 0;
}

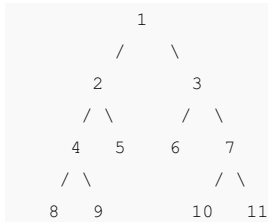
```

Thanks to Dhanya for suggesting this approach.

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.



See [next post](#) for more solutions.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

44. Connect nodes at same level using constant extra space

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

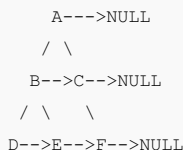
pointers to point next right for each node. You can use only constant extra space.

Example

Input Tree



Output Tree

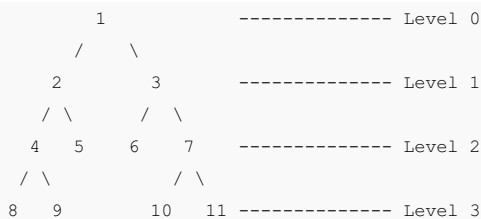


We discussed two different approaches to do it in the [previous post](#). The auxiliary space required in both of those approaches is not constant. Also, the method 2 discussed there only works for complete Binary Tree.

In this post, we will first modify the method 2 to make it work for all kind of trees. After that, we will remove recursion from this method so that the extra space becomes constant.

A Recursive Solution

In the method 2 of previous post, 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 *n)
```

```

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 the
nextRight of nodes at 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
This function is used to get the next 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.

```

#include <stdio.h>
#include <stdlib.h>

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

/* This function returns the leftmost child of nodes at the same level
   This function is used to getNext right of p's right child
   If right child of 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;
}

/* 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 children nodes of p and children nodes of all o
       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
            // 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);
}
}

```

```

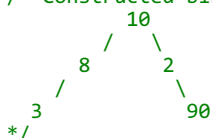
/* UTILITY FUNCTIONS */
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    node->nextRight = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{

```

/* Constructed binary tree is



```

struct node *root = newnode(10);
root->left      = newnode(8);
root->right     = newnode(2);
root->left->left = newnode(3);
root->right->right = newnode(90);

// Populates nextRight pointer in all nodes
connect(root);

// Let us check the values of nextRight pointers
printf("Following are populated nextRight pointers in the tree "
      "(-1 is printed if there is no nextRight) \n");
printf("nextRight of %d is %d \n", root->data,
      root->nextRight? root->nextRight->data: -1);
printf("nextRight of %d is %d \n", root->left->data,
      root->left->nextRight? root->left->nextRight->data: -1);
printf("nextRight of %d is %d \n", root->right->data,
      root->right->nextRight? root->right->nextRight->data: -1);
printf("nextRight of %d is %d \n", root->left->left->data,
      root->left->left->nextRight? root->left->left->nextRight->data: -1);
printf("nextRight of %d is %d \n", root->right->right->data,
      root->right->right->nextRight? root->right->right->nextRight->data: -1);

getchar();
return 0;
}

```

Output:

```

Following are populated nextRight pointers in the tree (-1 is printed if
there is no nextRight)
nextRight of 10 is -1
nextRight of 8 is 2
nextRight of 2 is -1
nextRight of 3 is 90
nextRight of 90 is -1

```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

45. 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

In the [previous post](#), we discussed construction of BST from sorted Linked List. 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.

Following is C implementation of the above algorithm. The main code which creates Balanced BST is highlighted.

```

#include<stdio.h>
#include<stdlib.h>

/* A Binary Tree node */
struct TNode
{
    int data;
    struct TNode* left;
    struct TNode* right;
};

struct TNode* newNode(int data);

/* A function that constructs Balanced Binary Search Tree from a sorted array */
struct TNode* sortedArrayToBST(int arr[], int start, int end)
{
    /* Base Case */
    if (start > end)
        return NULL;

    /* Get the middle element and make it root */
    int mid = (start + end)/2;
    struct TNode *root = newNode(arr[mid]);

    /* Recursively construct the left subtree and make it
       left child of root */

```

```

    left child of root */
    root->left = sortedArrayToBST(arr, start, mid-1);

    /* Recursively construct the right subtree and make it
       right child of root */
    root->right = sortedArrayToBST(arr, mid+1, end);

    return root;
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct TNode* newNode(int data)
{
    struct TNode* node = (struct TNode*)
        malloc(sizeof(struct TNode));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return node;
}

/* A utility function to print preorder traversal of BST */
void preOrder(struct TNode* node)
{
    if (node == NULL)
        return;
    printf("%d ", node->data);
    preOrder(node->left);
    preOrder(node->right);
}

/* Driver program to test above functions */
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    /* Convert List to BST */
    struct TNode *root = sortedArrayToBST(arr, 0, n-1);
    printf("\n PreOrder Traversal of constructed BST ");
    preOrder(root);

    return 0;
}

```

Time Complexity: $O(n)$

Following is the recurrence 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)

The above recurrence can be solved using **Master Theorem** as it falls in case 2.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

46. 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.

```
#include <stdio.h>
#include <stdlib.h>
```

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

```
/* Set next of p and all descendents of p by traversing them in reverse
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);
    }
}
```

```

}
}

/* UTILITY FUNCTIONS */
/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    node->next = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    /* Constructed binary tree is
        10
       / \
      8   12
     /
    3
    */
    struct node *root = newNode(10);
    root->left = newNode(8);
    root->right = newNode(12);
    root->left->left = newNode(3);

    // Populates nextRight pointer in all nodes
    populateNext(root);

    // Let us see the populated values
    struct node *ptr = root->left->left;
    while(ptr)
    {
        // -1 is printed if there is no successor
        printf("Next of %d is %d \n", ptr->data, ptr->next? ptr->next->data : -1);
        ptr = ptr->next;
    }

    return 0;
}

```

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

```

// An implementation that doesn't use static variable
// A wrapper over populateNextRecur
void populateNext(struct node *root)
{
    // The first visited node will be the rightmost node
    // next of the rightmost node will be NULL
    struct node *next = NULL;

    populateNextRecur(root, &next);
}

/* Set next of all descendents of p by traversing them in reverse Inorder
void populateNextRecur(struct node* p, struct node **next_ref)
{
    if (p)
    {
        // First set the next pointer in right subtree
        populateNextRecur(p->right, next_ref);

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

        // Change the prev for subsequent node
        *next_ref = p;

        // Finally, set the next pointer in right subtree
        populateNextRecur(p->left, next_ref);
    }
}

```

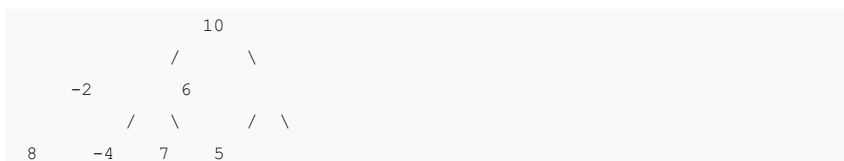
Time Complexity: $O(n)$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

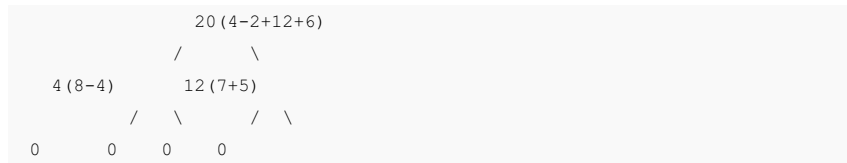
47. Convert a given tree to its Sum Tree

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



Solution:

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).

```
#include<stdio.h>

/* A tree node structure */
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

// Convert a given tree to a tree where every node contains sum of value of
// nodes in left and right subtrees in the original tree
int toSumTree(struct node *node)
{
    // Base case
    if(node == NULL)
        return 0;

    // Store the old value
    int old_val = node->data;

    // Recursively call for left and right subtrees and store the sum of
    // new value of this node
    node->data = toSumTree(node->left) + toSumTree(node->right);

    // Return the sum of values of nodes in left and right subtrees and
    // old_value of this node
    return node->data + old_val;
}

// A utility function to print inorder traversal of a Binary Tree
void printInorder(struct node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

/* Utility function to create a new Binary Tree node */
struct node* newNode(int data)
{
    struct node *temp = new struct node;
```

```

temp->data = data;
temp->left = NULL;
temp->right = NULL;

return temp;
}

/* Driver function to test above functions */
int main()
{
    struct node *root = NULL;
    int x;

    /* Constructing tree given in the above figure */
    root = newNode(10);
    root->left = newNode(-2);
    root->right = newNode(6);
    root->left->left = newNode(8);
    root->left->right = newNode(-4);
    root->right->left = newNode(7);
    root->right->right = newNode(5);

    toSumTree(root);

    // Print inorder traversal of the converted tree to test result of to
    printf("Inorder Traversal of the resultant tree is: \n");
    printInorder(root);

    getch();
    return 0;
}

```

Output:

```

Inorder Traversal of the resultant tree is:
0 4 0 20 0 12 0

```

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.

See [this](#) forum thread for the original question. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

48. Find the largest BST subtree in a given Binary Tree

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:

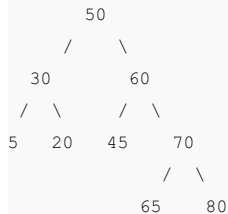


Output: 3

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.

```

/*
See http://www.geeksforgeeks.org/archives/632 for implementation of
See Method 3 of http://www.geeksforgeeks.org/archives/3042 for
implementation of isBST()

max() returns maximum of two integers
*/
int largestBST(struct node *root)
{
    if (isBST(root))
        return size(root);
    else
        return max(largestBST(root->left), largestBST(root->right));
}

```

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.

```

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */

```

```

struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

int largestBSTUtil(struct node* node, int *min_ref, int *max_ref,
                  int *max_size_ref, bool *is_bst_ref);

```

```

/* 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
   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_
       *max_ref = INT_MIN;
       ls = largestBSTUtil(node->left, min_ref, max_ref, max_size_ref, is_b
       if (*is_bst_ref == 1 && node->data < *max_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
   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_
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;
}
}
}

```

```

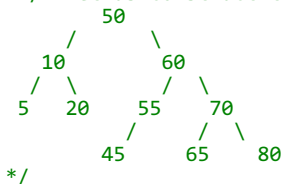
/* Driver program to test above functions*/
int main()
{

```

```

    /* Let us construct the following Tree

```



```

struct node *root = newNode(50);
root->left      = newNode(10);
root->right     = newNode(60);
root->left->left = newNode(5);
root->left->right = newNode(20);
root->right->left = newNode(55);
root->right->left->left = newNode(45);
root->right->right = newNode(70);
root->right->right->left = newNode(65);
root->right->right->right = newNode(80);

```

```

/* The complete tree is not BST as 45 is in right subtree of 50.
   The following subtree is the largest BST
           60
          /  \
         55   70
        /  \  /  \
       45  65 65  80
      */
printf(" Size of the largest BST is %d", largestBST(root));

getchar();
return 0;
}

```

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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

49. AVL Tree | Set 1 (Insertion)

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

Why AVL Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take $O(h)$ time where h is the height of the BST. The cost of these operations may become $O(n)$ for a skewed Binary tree. If we make sure that height of the tree remains $O(\log n)$ after every insertion and deletion, then we can guarantee an upper bound of $O(\log n)$ for all these operations. The height of an AVL tree is always $O(\log n)$ where n is the number of nodes in the tree (See [this](#) video lecture for proof).

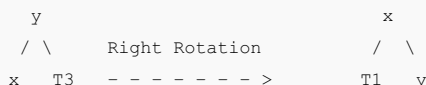
Insertion

To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property ($\text{keys}(\text{left}) < \text{key}(\text{root}) < \text{keys}(\text{right})$).

1) Left Rotation

2) Right Rotation

T1, T2 and T3 are subtrees of the tree rooted with y (on left side)
or x (on right side)



```

      / \      < - - - - -      / \
     T1  T2      Left Rotation      T2  T3

```

Keys in both of the above trees follow the following order

$$\text{keys}(T1) < \text{key}(x) < \text{keys}(T2) < \text{key}(y) < \text{keys}(T3)$$

So BST property is not violated anywhere.

Steps to follow for insertion

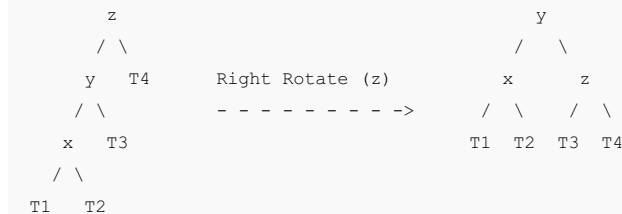
Let the newly inserted node be w

- 1) Perform standard BST insert for w.
- 2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the child of z that comes on the path from w to z and x be the grandchild of z that comes on the path from w to z.
- 3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:
 - a) y is left child of z and x is left child of y (Left Left Case)
 - b) y is left child of z and x is right child of y (Left Right Case)
 - c) y is right child of z and x is right child of y (Right Right Case)
 - d) y is right child of z and x is left child of y (Right Left Case)

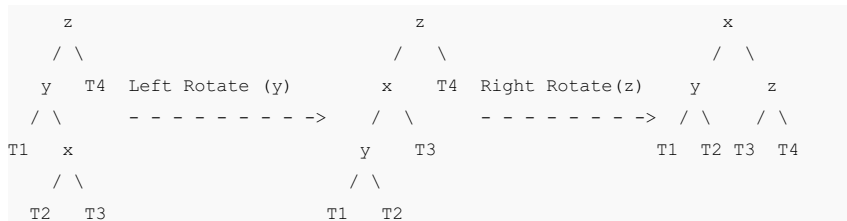
Following are the operations to be performed in above mentioned 4 cases. In all of the cases, we only need to re-balance the subtree rooted with z and the complete tree becomes balanced as the height of subtree (After appropriate rotations) rooted with z becomes same as it was before insertion. (See [this](#) video lecture for proof)

a) Left Left Case

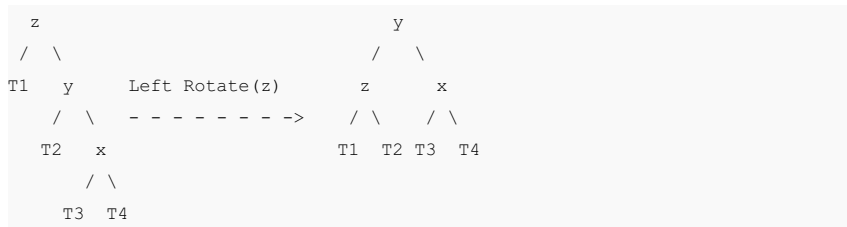
T1, T2, T3 and T4 are subtrees.



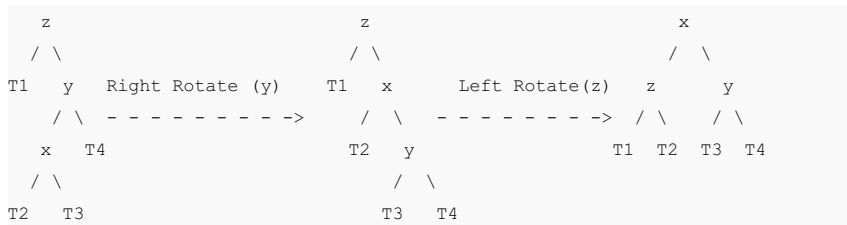
b) Left Right Case



c) Right Right Case



d) Right Left Case



C implementation

Following is the C implementation for AVL Tree Insertion. The following C implementation uses the recursive BST insert to insert a new node. In the recursive BST insert, after insertion, we get pointers to all ancestors one by one in bottom up manner. So we don't need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.

- 1) Perform the normal BST insertion.
- 2) The current node must be one of the ancestors of the newly inserted node. Update the height of the current node.
- 3) Get the balance factor (left subtree height – right subtree height) of the current node.
- 4) If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or left Right case. To check whether it is left left case or not, compare the newly inserted key with the key in left subtree root.
- 5) If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or not, compare the newly inserted key with the key in right subtree root.

```
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct node
{
    int key;
    struct node *left;
    struct node *right;
    int height;
};

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get height of the tree
```

```

// A utility function to get height of the tree
int height(struct node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Helper function that allocates a new node with the given key and
   NULL left and right pointers. */
struct node* newNode(int key)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->key    = key;
    node->left   = NULL;
    node->right  = NULL;
    node->height = 1; // new node is initially added at leaf
    return(node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct node *rightRotate(struct node *y)
{
    struct node *x = y->left;
    struct node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right))+1;
    x->height = max(height(x->left), height(x->right))+1;

    // Return new root
    return x;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    struct node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;

    // Return new root
    return y;
}

```

```

}

// Get Balance factor of node N
int getBalance(struct node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

struct node* insert(struct node* node, int key)
{
    /* 1. Perform the normal BST rotation */
    if (node == NULL)
        return (newNode(key));

    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    /* 2. Update height of this ancestor node */
    node->height = max(height(node->left), height(node->right)) + 1;

    /* 3. Get the balance factor of this ancestor node to check whether
    this node became unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}

```

```

// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(struct node *root)
{
    if (root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

```

```

        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Drier program to test above function*/
int main()
{
    struct node *root = NULL;

    /* Constructing tree given in the above figure */
    root = insert(root, 10);
    root = insert(root, 20);
    root = insert(root, 30);
    root = insert(root, 40);
    root = insert(root, 50);
    root = insert(root, 25);

    /* The constructed AVL Tree would be
        30
       /  \
      20   40
     /  \   \
    10  25  50
    */

    printf("Pre order traversal of the constructed AVL tree is \n");
    preOrder(root);

    return 0;
}

```

Output:

```

Pre order traversal of the constructed AVL tree is
30 20 10 25 40 50

```

Time Complexity: The rotation operations (left and right rotate) take constant time as only few pointers are being changed there. Updating the height and getting the balance factor also take constant time. So the time complexity of AVL insert remains same as BST insert which is $O(h)$ where h is height of the tree. Since AVL tree is balanced, the height is $O(\log n)$. So time complexity of AVL insert is $O(\log n)$.

The AVL tree and other self balancing search trees like Red Black are useful to get all basic operations done in $O(\log n)$ time. The AVL trees are more balanced compared to Red Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is more frequent operation, then AVL tree should be preferred over Red Black Tree.

Following are some previous posts that have used self-balancing search trees.

[Median in a stream of integers \(running integers\)](#)

[Maximum of all subarrays of size k](#)

Count smaller elements on right side

References:

[IITD Video Lecture on AVL Tree Introduction](#)

[IITD Video Lecture on AVL Tree Insertion and Deletion](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

50. 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. Thanks to [Nages](#) for suggesting this method.

```
import java.util.HashMap;

// Class for a tree node
class TreeNode {

    // data members
    private int key;
    private TreeNode left;
    private TreeNode right;

    // Accessor methods
    public int key() { return key; }
    public TreeNode left() { return left; }
    public TreeNode right() { return right; }

    // Constructor
    public TreeNode(int key) { this.key = key; left = null; right = null; }

    // Methods to set left and right subtrees
    public void setLeft(TreeNode left) { this.left = left; }
    public void setRight(TreeNode right) { this.right = right; }
}

// Class for a Binary Tree
class Tree {

    private TreeNode root;

    // Constructors
    public Tree() { root = null; }
    public Tree(TreeNode n) { root = n; }

    // Method to be called by the consumer classes like Main class
    public void VerticalSumMain() { VerticalSum(root); }

    // 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);
}

// Driver class to test the verticalSum methods
public class Main {

    public static void main(String[] args) {
        /* Create following Binary Tree
            1
           / \
          2   3
         / \ / \
        4  5 6  7
        */
        TreeNode root = new TreeNode(1);
        root.setLeft(new TreeNode(2));
    }
}

```

```

        root.setRight(new TreeNode(3));
        root.left().setLeft(new TreeNode(4));
        root.left().setRight(new TreeNode(5));
        root.right().setLeft(new TreeNode(6));
        root.right().setRight(new TreeNode(7));
        Tree t = new Tree(root);

        System.out.println("Following are the values of vertical sums with "
            + "the positions of the columns with respect to root ");
        t.VerticalSumMain();
    }
}

```

See [this](#) for a sample run.

Time Complexity: $O(n)$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

51. AVL Tree | Set 2 (Deletion)

We have discussed AVL insertion in the [previous post](#). In this post, we will follow a similar approach for deletion.

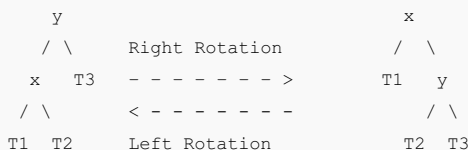
Steps to follow for deletion.

To make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property ($\text{keys}(\text{left}) < \text{key}(\text{root}) < \text{keys}(\text{right})$).

1) Left Rotation

2) Right Rotation

T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side)



Keys in both of the above trees follow the following order

$\text{keys}(T1) < \text{key}(x) < \text{keys}(T2) < \text{key}(y) < \text{keys}(T3)$

So BST property is not violated anywhere.

Let w be the node to be deleted

1) Perform standard BST delete for w.

2) Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from [insertion](#) here.

3) Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements:

a) y is left child of z and x is left child of y (Left Left Case)

b) y is left child of z and x is right child of y (Left Right Case)

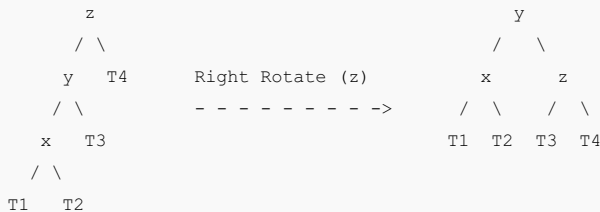
c) y is right child of z and x is right child of y (Right Right Case)

d) y is right child of z and x is left child of y (Right Left Case)

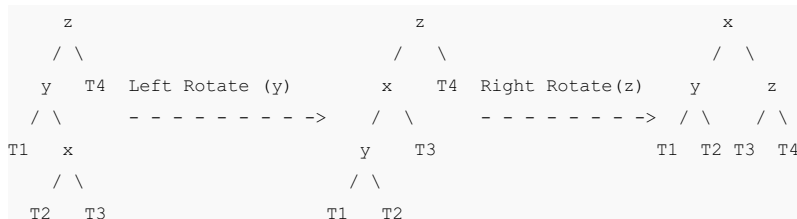
Like insertion, following are the operations to be performed in above mentioned 4 cases. Note that, unlike insertion, fixing the node z won't fix the complete AVL tree. After fixing z, we may have to fix ancestors of z as well (See [this video lecture](#) for proof)

a) Left Left Case

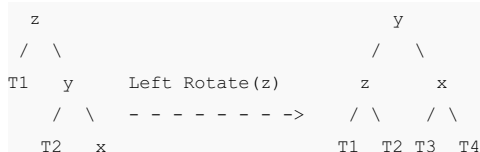
T1, T2, T3 and T4 are subtrees.



b) Left Right Case



c) Right Right Case

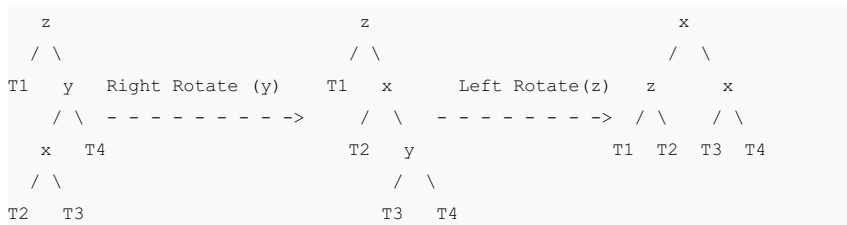


```

    / \
   T3  T4

```

d) Right Left Case



Unlike insertion, in deletion, after we perform a rotation at z , we may have to perform a rotation at ancestors of z . Thus, we must continue to trace the path until we reach the root.

C implementation

Following is the C implementation for AVL Tree Deletion. The following C implementation uses the recursive BST delete as basis. In the recursive BST delete, after deletion, we get pointers to all ancestors one by one in bottom up manner. So we don't need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the deleted node.

- 1) Perform the normal BST deletion.
- 2) The current node must be one of the ancestors of the deleted node. Update the height of the current node.
- 3) Get the balance factor (left subtree height – right subtree height) of the current node.
- 4) If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or Left Right case. To check whether it is Left Left case or Left Right case, get the balance factor of left subtree. If balance factor of the left subtree is greater than or equal to 0, then it is Left Left case, else Left Right case.
- 5) If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right Left case. To check whether it is Right Right case or Right Left case, get the balance factor of right subtree. If the balance factor of the right subtree is smaller than or equal to 0, then it is Right Right case, else Right Left case.

```

#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct node
{
    int key;
    struct node *left;
    struct node *right;
    int height;
};

// A utility function to get maximum of two integers
int max(int a, int b);

```

```

// A utility function to get height of the tree
int height(struct node *N)
{
    if (N == NULL)
        return 0;
    return N->height;
}

// A utility function to get maximum of two integers
int max(int a, int b)
{
    return (a > b)? a : b;
}

/* Helper function that allocates a new node with the given key and
   NULL left and right pointers. */
struct node* newNode(int key)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // new node is initially added at leaf
    return(node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct node *rightRotate(struct node *y)
{
    struct node *x = y->left;
    struct node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left), height(y->right))+1;
    x->height = max(height(x->left), height(x->right))+1;

    // Return new root
    return x;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    struct node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left), height(x->right))+1;
    y->height = max(height(y->left), height(y->right))+1;

    // Return new root

```

```

    return y;
}

// Get Balance factor of node N
int getBalance(struct node *N)
{
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

struct node* insert(struct node* node, int key)
{
    /* 1. Perform the normal BST rotation */
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    /* 2. Update height of this ancestor node */
    node->height = max(height(node->left), height(node->right)) + 1;

    /* 3. Get the balance factor of this ancestor node to check whether
    this node became unbalanced */
    int balance = getBalance(node);

    // If this node becomes unbalanced, then there are 4 cases

    // Left Left Case
    if (balance > 1 && key < node->left->key)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && key > node->right->key)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && key > node->left->key)
    {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && key < node->right->key)
    {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    /* return the (unchanged) node pointer */
    return node;
}

/* Given a non-empty binary search tree, return the node with minimum
key value found in that tree. Note that the entire tree does not
need to be searched. */
struct node * minValueNode(struct node* node)
{
    struct node* current = node;
    while (current->left != NULL)
        current = current->left;
    return current;
}

```

```

    struct node* current = node;

    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
        current = current->left;

    return current;
}

struct node* deleteNode(struct node* root, int key)
{
    // STEP 1: PERFORM STANDARD BST DELETE

    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) || (root->right == NULL) )
        {
            struct node *temp = root->left ? root->left : root->right;

            // No child case
            if(temp == NULL)
            {
                temp = root;
                root = NULL;
            }
            else // One child case
                *root = *temp; // Copy the contents of the non-empty child

            free(temp);
        }
        else
        {
            // node with two children: Get the inorder successor (smallest
            // in the right subtree)
            struct node* temp = minValueNode(root->right);

            // Copy the inorder successor's data to this node
            root->key = temp->key;

            // Delete the inorder successor
            root->right = deleteNode(root->right, temp->key);
        }
    }

    // If the tree had only one node then return
    if (root == NULL)
        return root;
}

```

```

return root;

// STEP 2: UPDATE HEIGHT OF THE CURRENT NODE
root->height = max(height(root->left), height(root->right)) + 1;

// STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether
// this node became unbalanced)
int balance = getBalance(root);

// If this node becomes unbalanced, then there are 4 cases

// Left Left Case
if (balance > 1 && getBalance(root->left) >= 0)
    return rightRotate(root);

// Left Right Case
if (balance > 1 && getBalance(root->left) < 0)
{
    root->left = leftRotate(root->left);
    return rightRotate(root);
}

// Right Right Case
if (balance < -1 && getBalance(root->right) <= 0)
    return leftRotate(root);

// Right Left Case
if (balance < -1 && getBalance(root->right) > 0)
{
    root->right = rightRotate(root->right);
    return leftRotate(root);
}

return root;
}

```

// A utility function to print preorder traversal of the tree.

// The function also prints height of every node

```

void preOrder(struct node *root)
{
    if(root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

```

/* Driver program to test above function*/

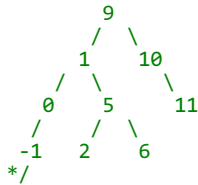
```

int main()
{
    struct node *root = NULL;

    /* Constructing tree given in the above figure */
    root = insert(root, 9);
    root = insert(root, 5);
    root = insert(root, 10);
    root = insert(root, 0);
    root = insert(root, 6);
    root = insert(root, 11);
    root = insert(root, -1);
    root = insert(root, 1);
    root = insert(root, 2);
}

```

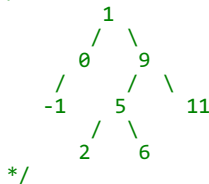
```
/* The constructed AVL Tree would be
```



```
printf("Pre order traversal of the constructed AVL tree is \n");
preOrder(root);
```

```
root = deleteNode(root, 10);
```

```
/* The AVL Tree after deletion of 10
```



```
printf("\nPre order traversal after deletion of 10 \n");
preOrder(root);
```

```
return 0;
```

```
}
```

Output:

```
Pre order traversal of the constructed AVL tree is
```

```
9 1 0 -1 5 2 6 10 11
```

```
Pre order traversal after deletion of 10
```

```
1 0 -1 9 5 2 6 11
```

Time Complexity: The rotation operations (left and right rotate) take constant time as only few pointers are being changed there. Updating the height and getting the balance factor also take constant time. So the time complexity of AVL delete remains same as BST delete which is $O(h)$ where h is height of the tree. Since AVL tree is balanced, the height is $O(\log n)$. So time complexity of AVL delete is $O(\log n)$.

References:

[IITD Video Lecture on AVL Tree Insertion and Deletion](#)

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

52. 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.

In the following solutions, it is assumed that sizes of trees are also given as input. If the size is not given, then we can get the size by traversing the tree (See [this](#)).

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 $\text{Log}(n) + \text{Log}(n+1) \dots \text{Log}(m+n-1)$. The value of this expression will be between $m\text{Log}n$ and $m\text{Log}(m+n-1)$. 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 using the technique discussed in [this](#) post. 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.

Following is C++ implementation of this method.

```
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

// A utility function to merge two sorted arrays into one
int *merge(int arr1[], int arr2[], int m, int n);

// A helper function that stores inorder traversal of a tree in inorder
void storeInorder(struct node* node, int inorder[], int *index_ptr);

/* A function that constructs Balanced Binary Search Tree from a sorted
   See http://www.geeksforgeeks.org/archives/17138 */
struct node* sortedArrToBST(int arr[], int start, int end);
```



```

// ... here ... return root1, root2, arr1, arr2, arr3;

/* This function merges two balanced BSTs with roots as root1 and root2.
m and n are the sizes of the trees respectively */
struct node* mergeTrees(struct node *root1, struct node *root2, int m,
{
    // Store inorder traversal of first tree in an array arr1[]
    int *arr1 = new int[m];
    int i = 0;
    storeInorder(root1, arr1, &i);

    // Store inorder traversal of second tree in another array arr2[]
    int *arr2 = new int[n];
    int j = 0;
    storeInorder(root2, arr2, &j);

    // Merge the two sorted array into one
    int *mergedArr = merge(arr1, arr2, m, n);

    // Construct a tree from the merged array and return root of the tree
    return sortedArrayToBST (mergedArr, 0, m+n-1);
}

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

// A utility function to print inorder traversal of a given binary tree
void printInorder(struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder(node->left);

    printf("%d ", node->data);

    /* now recur on right child */
    printInorder(node->right);
}

// A utility function to merge two sorted arrays into one
int *merge(int arr1[], int arr2[], int m, int n)
{
    // mergedArr[] is going to contain result
    int *mergedArr = new int[m + n];
    int i = 0, j = 0, k = 0;

    // Traverse through both arrays
    while (i < m && j < n)
    {
        // Pick the smaller element and put it in mergedArr
        if (arr1[i] < arr2[j])

```

```

        {
            mergedArr[k] = arr1[i];
            i++;
        }
        else
        {
            mergedArr[k] = arr2[j];
            j++;
        }
        k++;
    }

    // If there are more elements in first array
    while (i < m)
    {
        mergedArr[k] = arr1[i];
        i++; k++;
    }

    // If there are more elements in second array
    while (j < n)
    {
        mergedArr[k] = arr2[j];
        j++; k++;
    }

    return mergedArr;
}

// A helper function that stores inorder traversal of a tree rooted with node
void storeInorder(struct node* node, int inorder[], int *index_ptr)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    storeInorder(node->left, inorder, index_ptr);

    inorder[*index_ptr] = node->data;
    (*index_ptr)++; // increase index for next entry

    /* now recur on right child */
    storeInorder(node->right, inorder, index_ptr);
}

// A function that constructs Balanced Binary Search Tree from a sorted array
// See http://www.geeksforgeeks.org/archives/17138 */
struct node* sortedArrayToBST(int arr[], int start, int end)
{
    /* Base Case */
    if (start > end)
        return NULL;

    /* Get the middle element and make it root */
    int mid = (start + end)/2;
    struct node *root = newNode(arr[mid]);

    /* Recursively construct the left subtree and make it
       left child of root */
    root->left = sortedArrayToBST(arr, start, mid-1);

    /* Recursively construct the right subtree and make it
       right child of root */

```

```

        right child of root */
        root->right = sortedArrayToBST(arr, mid+1, end);

    return root;
}

/* Driver program to test above functions*/
int main()
{
    /* Create following tree as first balanced BST
        100
       /  \
      50   300
     /  \
    20   70
    */
    struct node *root1 = newNode(100);
    root1->left = newNode(50);
    root1->right = newNode(300);
    root1->left->left = newNode(20);
    root1->left->right = newNode(70);

    /* Create following tree as second balanced BST
        80
       /  \
      40   120
    */
    struct node *root2 = newNode(80);
    root2->left = newNode(40);
    root2->right = newNode(120);

    struct node *mergedTree = mergeTrees(root1, root2, 5, 3);

    printf ("Following is Inorder traversal of the merged tree \n");
    printInorder(mergedTree);

    getchar();
    return 0;
}

```

Output:

```

Following is Inorder traversal of the merged tree
20 40 50 70 80 100 120 300

```

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 (Refer [this post](#) for this step).
- 2) Merge the two sorted Linked Lists (Refer [this post](#) for this step).
- 3) Build a Balanced Binary Search Tree from the merged list created in step 2. (Refer [this post](#) for this step)

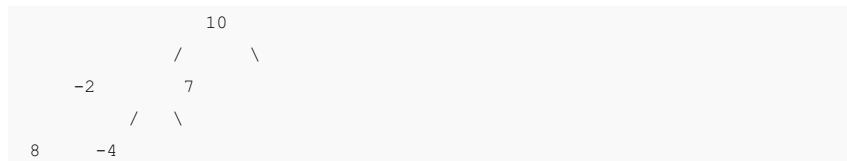
Time complexity of this method is also $O(m+n)$ and this method does conversion in place.

Thanks to [Dheeraj](#) and [Ronzi](#) for suggesting this method.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

53. Find the maximum sum leaf to root 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.

```
#include<stdio.h>
#include<limits.h>

/* A tree node structure */
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

// 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))
        return true;
}
```

```

        printf("%d ", target_leaf->data);
    }

    printf("%d ", root->data);
    return true;
}

return false;
}

// This function Sets the target_leaf_ref to refer the leaf node of the
// 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 the
    curr_sum = curr_sum + node->data;

    // If this is a leaf node and path to this node has maximum sum so
    // 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
    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
}

/* Utility function to create a new Binary Tree node */
struct node* newNode (int data)
{
    struct node *temp = new struct node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

```

```

}

/* Driver function to test above functions */
int main()
{
    struct node *root = NULL;

    /* Constructing tree given in the above figure */
    root = newNode(10);
    root->left = newNode(-2);
    root->right = newNode(7);
    root->left->left = newNode(8);
    root->left->right = newNode(-4);

    printf ("Following are the nodes on the maximum sum path \n");
    int sum = maxSumPath(root);
    printf ("\nSum of the nodes is %d ", sum);

    getchar();
    return 0;
}

```

Output:

```

Following are the nodes on the maximum sum path
7 10
Sum of the nodes is 17

```

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

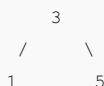
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

54. 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 $O(\text{height of the first tree} + \text{height of the second tree})$.

Examples:

First BST



Second BST

```

      4
     / \
    2   6
Output: 1 2 3 4 5 6

```

First BST

```

      8
     / \
    2   10
   /
  1

```

Second BST

```

      5
     /
    3
   /
  0

```

Output: 0 1 2 3 5 8 10

Source: [Google interview question](#)

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.

Thanks to [Kumar](#) for suggesting the following solution.

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.

```

#include<stdio.h>
#include<stdlib.h>

// Structure of a BST Node
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

//..... START OF STACK RELATED STUFF.....
// A stack node
struct snode
{

```

```

`
    struct node *t;
    struct snode *next;
};

// Function to add an elemnt k to stack
void push(struct snode **s, struct node *k)
{
    struct snode *tmp = (struct snode *) malloc(sizeof(struct snode));

    //perform memory check here
    tmp->t = k;
    tmp->next = *s;
    (*s) = tmp;
}

// Function to pop an element t from stack
struct node *pop(struct snode **s)
{
    struct node *t;
    struct snode *st;
    st=*s;
    (*s) = (*s)->next;
    t = st->t;
    free(st);
    return t;
}

// Fucntion to check whether the stack is empty or not
int isEmpty(struct snode *s)
{
    if (s == NULL )
        return 1;

    return 0;
}

//..... END OF STACK RELATED STUFF.....

/* Utility function to create a new Binary Tree node */
struct node* newNode (int data)
{
    struct node *temp = new struct node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

/* A utility function to print Inoder traversal of a Binary Tree */
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

// 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
        // 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, print the other tree
        if (isEmpty(s1))
        {
            while (!isEmpty(s2))
            {
                current2 = pop (&s2);
                current2->left = NULL;
                inorder(current2);
            }
            return ;
        }
        else
        {
            while (!isEmpty(s1))
            {
                current1 = pop (&s1);
                current1->right = NULL;
                inorder(current1);
            }
            return ;
        }
    }
}

```



```

merge(root1, root2);

return 0;
}

```

Time Complexity: $O(m+n)$

Auxiliary Space: $O(\text{height of the first tree} + \text{height of the second tree})$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

55. 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

```

Example 2

Input:

```

      10
     /  \
    30   15
   /     \
  20      5

```

Output:

```

      15
     /  \

```



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 $O(n\log n)$ 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.

Following is C implementation of the above approach. The main function to convert is highlighted in the following code.

```

/* A program to convert Binary Tree to Binary Search Tree */
#include<stdio.h>
#include<stdlib.h>

/* A binary tree node structure */
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

/* A helper function that stores inorder traversal of a tree rooted
with node */
void storeInorder (struct node* node, int inorder[], int *index_ptr)
{
    // Base Case
    if (node == NULL)
        return;

    /* first store the left subtree */
    storeInorder (node->left, inorder, index_ptr);

    /* Copy the root's data */
    inorder[*index_ptr] = node->data;
    (*index_ptr)++; // increase index for next entry

    /* finally store the right subtree */
    storeInorder (node->right, inorder, index_ptr);
}

/* A helper function to count nodes in a Binary Tree */
int countNodes (struct node* root)
{
    if (root == NULL)
        return 0;
    return countNodes (root->left) +
           countNodes (root->right) + 1;
}
  
```

```
// Following function is needed for library function qsort()
int compare (const void * a, const void * b)
{
    return ( *(int*)a - *(int*)b );
}

/* A helper function that copies contents of arr[] to Binary Tree.
This function basically does Inorder traversal of Binary Tree and
one by one copy arr[] elements to Binary Tree nodes */
void arrayToBST (int *arr, struct node* root, int *index_ptr)
{
    // Base Case
    if (root == NULL)
        return;

    /* first update the left subtree */
    arrayToBST (arr, root->left, index_ptr);

    /* Now update root's data and increment index */
    root->data = arr[*index_ptr];
    (*index_ptr)++;

    /* finally update the right subtree */
    arrayToBST (arr, root->right, index_ptr);
}
```

```
// This function converts a given Binary Tree to BST
void binaryTreeToBST (struct node *root)
{
    // base case: tree is empty
    if(root == NULL)
        return;

    /* Count the number of nodes in Binary Tree so that
    we know the size of temporary array to be created */
    int n = countNodes (root);

    // Create a temp array arr[] and store inorder traversal of tree in it
    int *arr = new int[n];
    int i = 0;
    storeInorder (root, arr, &i);

    // Sort the array using library function for quick sort
    qsort (arr, n, sizeof(arr[0]), compare);

    // Copy array elements back to Binary Tree
    i = 0;
    arrayToBST (arr, root, &i);

    // delete dynamically allocated memory to avoid memory leak
    delete [] arr;
}
```

```
/* Utility function to create a new Binary Tree node */
struct node* newNode (int data)
{
    struct node *temp = new struct node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}
```

```

/* Utility function to print inorder traversal of Binary Tree */
void printInorder (struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder (node->left);

    /* then print the data of node */
    printf("%d ", node->data);

    /* now recur on right child */
    printInorder (node->right);
}

/* Driver function to test above functions */
int main()
{
    struct node *root = NULL;

    /* Constructing tree given in the above figure
    10
   /  \
  30   15
 /    \
20     5
*/
    root = newNode(10);
    root->left = newNode(30);
    root->right = newNode(15);
    root->left->left = newNode(20);
    root->right->right = newNode(5);

    // convert Binary Tree to BST
    binaryTreeToBST (root);

    printf("Following is Inorder Traversal of the converted BST: \n");
    printInorder (root);

    return 0;
}

```

Output:

```

Following is Inorder Traversal of the converted BST:
5 10 15 20 30

```

We will be covering another method for this problem which converts the tree using $O(\text{height of tree})$ extra space.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

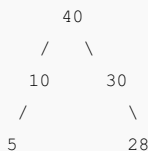
56. Construct Special Binary Tree from given Inorder traversal

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}`

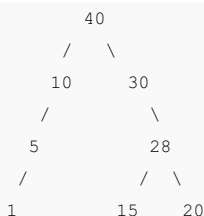
Output: root of following tree



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'.

Implementation: Following is C/C++ implementation of the above algorithm.

```
1 // C++ program to construct Special Binary Tree from Inorder traversal
```

```

/* program to construct tree from inorder traversal */
#include<stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Prototypes of a utility function to get the maximum
value in inorder[start..end] */
int max(int inorder[], int strt, int end);

/* A utility function to allocate memory for a node */
struct node* newNode(int data);

/* 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 subtreess */
    root->left = buildTree (inorder, start, i-1);
    root->right = buildTree (inorder, i+1, end);

    return root;
}

/* UTILITY FUNCTIONS */
/* Function to find index of the maximum value in arr[start...end] */
int max (int arr[], int strt, int end)
{
    int i, max = arr[strt], maxind = strt;
    for(i = strt+1; i <= end; i++)
    {
        if(arr[i] > max)
        {
            max = arr[i];
            maxind = i;
        }
    }
    return maxind;
}

```



```

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode (int data)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return node;
}

/* This function is here just to test buildTree() */
void printInorder (struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder (node->left);

    /* then print the data of node */
    printf("%d ", node->data);

    /* now recur on right child */
    printInorder (node->right);
}

/* Driver program to test above functions */
int main()
{
    /* Assume that inorder traversal of following tree is given
       40
      /  \
     10   30
    /  \  \
   5   28 28 */

    int inorder[] = {5, 10, 40, 30, 28};
    int len = sizeof(inorder)/sizeof(inorder[0]);
    struct node *root = buildTree(inorder, 0, len - 1);

    /* Let us test the built tree by printing Inorder traversal */
    printf("\n Inorder traversal of the constructed tree is \n");
    printInorder(root);
    return 0;
}

```

Output:

```

Inorder traversal of the constructed tree is
5 10 40 30 28

```

Time Complexity: $O(n^2)$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

57. Construct a special tree from given preorder traversal

Given an array 'pre[]' that represents Preorder traversal of a special 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.

Source: [Amazon Interview Question](#)

Example:

Input: pre[] = {10, 30, 20, 5, 15}, preLN[] = {'N', 'N', 'L', 'L', 'L'}

Output: Root of following tree



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 program to construct Binary Tree from preorder traversal */
#include<stdio.h>
```

```
/* A binary tree node structure */
```

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

```
/* Utility function to create a new Binary Tree node */
```

```
struct node* newNode (int data)
{
    struct node *temp = new struct node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}
```

```
/* 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 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
    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 m
    // the current index in pre[] and preLN[] arrays.
    int index = 0;

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

```

```

/* This function is used only for testing */
void printInorder (struct node* node)
{
    if (node == NULL)
        return;

    /* first recur on left child */
    printInorder (node->left);

    /* then print the data of node */
    printf("%d ", node->data);

    /* now recur on right child */
    printInorder (node->right);
}

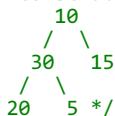
```

```

/* Driver function to test above functions */
int main()
{
    struct node *root = NULL;

```

/* Constructing tree given in the above figure



```

int pre[] = {10, 30, 20, 5, 15};
char preLN[] = {'N', 'N', 'L', 'L', 'L'};

```

```

int n = sizeof(pre)/sizeof(pre[0]);

// construct the above tree
root = constructTree (pre, preLN, n);

// Test the constructed tree
printf("Following is Inorder Traversal of the Constructed Binary T
printInorder (root);

return 0;
}

```

Output:

```

Following is Inorder Traversal of the Constructed Binary Tree:
20 30 5 10 15

```

Time Complexity: $O(n)$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

58. 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. Therefore, 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.

```
#include <stdio.h>
```

```
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;
}
```

```
// driver program to test above function
int main()
{
    int pre[] = {8, 3, 5, 7, 6};
    int size = sizeof(pre)/sizeof(pre[0]);
    if (hasOnlyOneChild(pre, size) == true )
        printf("Yes");
    else
        printf("No");
    return 0;
}
```

Output:

```
Yes
```

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.

```
#include <stdio.h>
```

```
int hasOnlyOneChild(int pre[], int size)
{
    // Initialize min and max using last two elements
    int min, max;
    if (pre[size-1] > pre[size-2])
    {
        max = pre[size-1];
        min = pre[size-2];
    }
    else
    {
        max = pre[size-2];
        min = pre[size-1];
    }

    // Every element must be either smaller than min or
    // greater than max
    for (int i=size-3; i>=0; i--)
    {
        if (pre[i] < min)
            min = pre[i];
        else if (pre[i] > max)
            max = pre[i];
        else
            return false;
    }
    return true;
}
```

```
// Driver program to test above function
int main()
{
    int pre[] = {8, 3, 5, 7, 6};
    int size = sizeof(pre)/sizeof(pre[0]);
    if (hasOnlyOneChild(pre,size))
        printf("Yes");
    else
        printf("No");
    return 0;
}
```

Output:

```
Yes
```

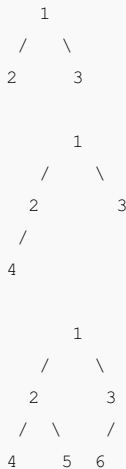
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

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

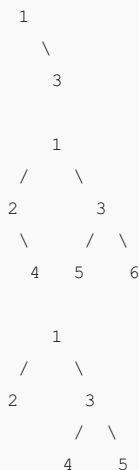
Given a Binary Tree, write a function to check whether the given Binary Tree is Complete Binary Tree or not.

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



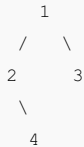
The following trees are examples of Non-Complete Binary Trees



Source: Write an algorithm to check if a tree is complete binary tree or not

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.



Thanks to Guddu Sharma for suggesting this simple and efficient approach.

```
// A program to check if a given binary tree is complete or not
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#define MAX_Q_SIZE 500

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* function prototypes for functions needed for Queue data
   structure. A queue is needed for level order traversal */
struct node** createQueue(int *, int *);
void enqueue(struct node **, int *, struct node *);
struct node *deQueue(struct node **, int *);
bool isEmptyQueue(int *front, int *rear);

/* 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(!isEmptyQueue(&front, &rear))
    {
        struct node *temp node = deQueue(queue, &front);
```



```

/* Ceck 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;

/* Ceck 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 left 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;
}

```

```

/*UTILITY FUNCTIONS*/
struct node** createQueue(int *front, int *rear)
{
    struct node **queue =
        (struct node **)malloc(sizeof(struct node)*MAX_Q_SIZE);

    *front = *rear = 0;
    return queue;
}

void enqueue(struct node **queue, int *rear, struct node *new_node)
{
    queue[*rear] = new_node;
    (*rear)++;
}

struct node *deQueue(struct node **queue, int *front)
{
    (*front)++;
    return queue[*front - 1];
}

bool isEmptyQueue(int *front, int *rear)
{
    return (*rear == *front);
}

```

```

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

/* Driver program to test above functions*/
int main()
{
    /* Let us construct the following Binary Tree which
       is not a complete Binary Tree
           1
          / \
         2   3
        / \   \
       4  5   6
      */

    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(6);

    if ( isCompleteBT(root) == true )
        printf ("Complete Binary Tree");
    else
        printf ("NOT Complete Binary Tree");

    return 0;
}

```

Output:

```
NOT Complete Binary Tree
```

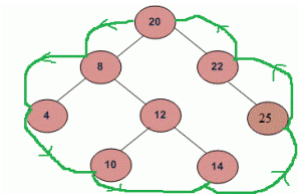
Time Complexity: $O(n)$ where n is the number of nodes in given Binary Tree

Auxiliary Space: $O(n)$ for queue.

Please write comments if you find any of the above codes/algorithms incorrect, or find other ways to solve the same problem.

60. 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:

```
/* program for boundary traversal of a binary tree */
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node *left, *right;
};

// A simple function to print leaf nodes of a binary tree
void printLeaves(struct node* root)
{
    if ( root )
    {
        printLeaves(root->left);

        // Print it if it is a leaf node
        if ( !(root->left) && !(root->right) )
            printf("%d ", root->data);

        printLeaves(root->right);
    }
}

// A function to print all left boundary nodes, except a leaf node.
// Print the nodes in TOP DOWN manner
void printBoundaryLeft(struct node* root)
{
    if (root)
    {
```

```

    if (root->left)
    {
        // to ensure top down order, print the node
        // before calling itself for left subtree
        printf("%d ", root->data);
        printBoundaryLeft(root->left);
    }
    else if( root->right )
    {
        printf("%d ", root->data);
        printBoundaryLeft(root->right);
    }
    // do nothing if it is a leaf node, this way we avoid
    // duplicates in output
}

// A function to print all right boundry nodes, except a leaf node
// Print the nodes in BOTTOM UP manner
void printBoundaryRight(struct node* root)
{
    if (root)
    {
        if ( root->right )
        {
            // to ensure bottom up order, first call for right
            // subtree, then print this node
            printBoundaryRight(root->right);
            printf("%d ", root->data);
        }
        else if ( root->left )
        {
            printBoundaryRight(root->left);
            printf("%d ", root->data);
        }
        // do nothing if it is a leaf node, this way we avoid
        // duplicates in output
    }
}

```

```

// 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);
    }
}

```

```

// A utility function to create a node
struct node* newNode( int data )

```

```

{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );

    temp->data = data;
    temp->left = temp->right = NULL;

    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree given in the above diagram
    struct node *root = newNode(20);
    root->left = newNode(8);
    root->left->left = newNode(4);
    root->left->right = newNode(12);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(14);
    root->right = newNode(22);
    root->right->right = newNode(25);

    printBoundary( root );

    return 0;
}

```

Output:

```
20 8 4 10 14 25 22
```

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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

61. Two nodes of a BST are swapped, correct the 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



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 auxiliary array elements back to the BST, keeping the structure of the BST same. Time complexity of this method is $O(n \log n)$ 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.

Following is C implementation of the given code.

```

// Two nodes in the BST's swapped, correct the BST.
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */

```

```

struct node
{
    int data;
    struct node *left, *right;
};

// A utility function to swap two integers
void swap( int* a, int* b )
{
    int t = *a;
    *a = *b;
    *b = t;
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node = (struct node *)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

// This function does inorder traversal to find out the two swapped nodes
// It sets three pointers, first, middle and last. If the swapped nodes are
// adjacent to each other, then first and middle contain the resultant nodes
// Else, first and last contain the resultant nodes
void correctBSTUtil( struct node* root, struct node** first,
                    struct node** middle, struct node** last,
                    struct node** prev )
{
    if( root )
    {
        // Recur for the left subtree
        correctBSTUtil( root->left, first, middle, last, prev );

        // If this node is smaller than the previous node, it's violating
        // the BST rule.
        if (*prev && root->data < (*prev)->data)
        {
            // If this is first violation, mark these two nodes as
            // 'first' and 'middle'
            if ( !*first )
            {
                *first = *prev;
                *middle = root;
            }

            // If this is second violation, mark this node as last
            else
                *last = root;
        }

        // Mark this node as previous
        *prev = root;

        // Recur for the right subtree
        correctBSTUtil( root->right, first, middle, last, prev );
    }
}

```

```

// A function to fix a given BST where two nodes are swapped. This
// function uses correctBSTUtil() to find out two nodes and swaps the
// nodes to fix the BST
void correctBST( struct node* root )
{
    // Initialize pointers needed for correctBSTUtil()
    struct node *first, *middle, *last, *prev;
    first = middle = last = prev = NULL;

    // Set the pointers to find out two nodes
    correctBSTUtil( root, &first, &middle, &last, &prev );

    // Fix (or correct) the tree
    if( first && last )
        swap( &(first->data), &(last->data) );
    else if( first && middle ) // Adjacent nodes swapped
        swap( &(first->data), &(middle->data) );

    // else nodes have not been swapped, passed tree is really BST.
}

/* A utility function to print Inorder traversal */
void printInorder(struct node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

/* Driver program to test above functions*/
int main()
{
    /*
        6
       / \
      10  2
     / \ / \
    1  3 7 12
    10 and 2 are swapped
    */

    struct node *root = newNode(6);
    root->left = newNode(10);
    root->right = newNode(2);
    root->left->left = newNode(1);
    root->left->right = newNode(3);
    root->right->right = newNode(12);
    root->right->left = newNode(7);

    printf("Inorder Traversal of the original tree \n");
    printInorder(root);

    correctBST(root);

    printf("\nInorder Traversal of the fixed tree \n");
    printInorder(root);

    return 0;
}

```


Output:

```
Inorder Traversal of the original tree
1 10 3 6 7 2 12
Inorder Traversal of the fixed tree
1 2 3 6 7 10 12
```

Time Complexity: $O(n)$

See [this](#) for different test cases of the above code.

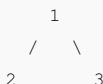
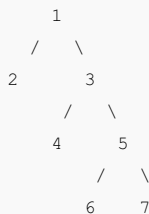
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

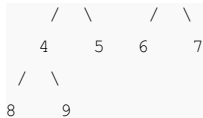
62. Construct Full Binary Tree from given preorder and postorder traversals

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.

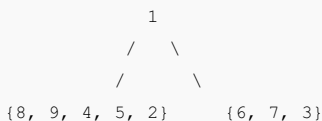




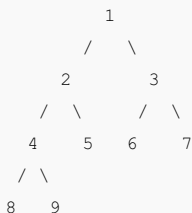
It is not possible to construct a general Binary Tree from preorder and postorder traversals (See [this](#)). But if know that the Binary Tree is Full, we can construct the tree without ambiguity. Let us understand this with the help of following example.

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.



```

/* program for construction of full binary tree */
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

// A utility function to create a node
struct node* newNode (int data)
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );

```

```

        struct node* temp = (struct node*) malloc( sizeof(struct node) );

        temp->data = data;
        temp->left = temp->right = NULL;

        return temp;
    }

// 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
    // preIndex from preorder and make it root, and increment preIndex
    struct node* root = newNode ( pre[*preIndex] );
    ++*preIndex;

    // If the current subarray 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
    // into 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
// 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);
}

// A utility function to print inorder traversal of a Binary Tree
void printInorder (struct node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

// Driver program to test above functions

```

```

int main ()
{
    int pre[] = {1, 2, 4, 8, 9, 5, 3, 6, 7};
    int post[] = {8, 9, 4, 5, 2, 6, 7, 3, 1};
    int size = sizeof( pre ) / sizeof( pre[0] );

    struct node *root = constructTree(pre, post, size);

    printf("Inorder traversal of the constructed tree: \n");
    printInorder(root);

    return 0;
}

```

Output:

```

Inorder traversal of the constructed tree:
8 4 9 2 5 1 6 3 7

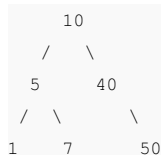
```

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

63. Construct BST from given preorder traversal | Set 1

Given preorder traversal of a binary search tree, construct the BST.

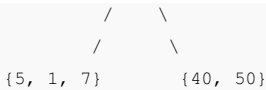
For example, if the given traversal is {10, 5, 1, 7, 40, 50}, then the output should be root of following tree.



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.



We recursively follow above steps for subarrays {5, 1, 7} and {40, 50}, and get the complete tree.

```

/* A O(n^2) program for construction of BST from preorder traversal */
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left child
and a pointer to right child */
struct node
{
    int data;
    struct node *left;
    struct node *right;
};

// A utility function to create a node
struct node* newNode (int data)
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );

    temp->data = data;
    temp->left = temp->right = NULL;

    return temp;
}

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

    // The first node in preorder traversal is root. So take the node
    // preIndex from pre[] and make it root, and increment preIndex
    struct node* root = newNode ( pre[*preIndex] );
    *preIndex = *preIndex + 1;

    // If the current subarray has only one element, no need to recur
    if (low == high)
        return root;

    // Search for the first element greater than root
    int i;
    for ( i = low; i <= high; ++i )
        if ( pre[ i ] > root->data )
            break;

    // Use the index of element found in postorder to divide postorder
    // two parts. Left subtree and right subtree
    root->left = constructTreeUtil ( pre, preIndex, *preIndex, i - 1,
    root->right = constructTreeUtil ( pre, preIndex, i, high, 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, 0, size - 1, size);
}

// A utility function to print inorder traversal of a Binary Tree
void printInorder (struct node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

// Driver program to test above functions
int main ()
{
    int pre[] = {10, 5, 1, 7, 40, 50};
    int size = sizeof( pre ) / sizeof( pre[0] );

    struct node *root = constructTree(pre, size);

    printf("Inorder traversal of the constructed tree: \n");
    printInorder(root);

    return 0;
}

```

Output:

```
1 5 7 10 40 50
```

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 O(n) program for construction of BST from preorder traversal */
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{

```

```

1
    int data;
    struct node *left;
    struct node *right;
};

// A utility function to create a node
struct node* newNode (int data)
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );

    temp->data = data;
    temp->left = temp->right = NULL;

    return temp;
}

// 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)
        {
            // Construct 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 );
}

// A utility function to print inorder traversal of a Binary Tree

```

```

void printInorder (struct node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

// Driver program to test above functions
int main ()
{
    int pre[] = {10, 5, 1, 7, 40, 50};
    int size = sizeof( pre ) / sizeof( pre[0] );

    struct node *root = constructTree(pre, size);

    printf("Inorder traversal of the constructed tree: \n");
    printInorder(root);

    return 0;
}

```

Output:

```
1 5 7 10 40 50
```

Time Complexity: $O(n)$

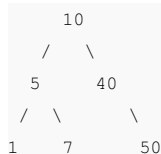
We will soon publish a $O(n)$ iterative solution as a separate post.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

64. Construct BST from given preorder traversal | Set 2

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.



We have discussed $O(n^2)$ and $O(n)$ recursive solutions in the [previous post](#). 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[].

```
/* A O(n) iterative program for construction of BST from preorder traversal
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
```

```
/* A binary tree node has data, pointer to left child
and a pointer to right child */
```

```
typedef struct Node
{
    int data;
    struct Node *left, *right;
} Node;
```

```
// A Stack has array of Nodes, capacity, and top
```

```
typedef struct Stack
{
    int top;
    int capacity;
    Node* *array;
} Stack;
```

```
// A utility function to create a new tree node
```

```
Node* newNode( int data )
{
    Node* temp = (Node *)malloc( sizeof( Node ) );
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}
```

```
// A utility function to create a stack of given capacity
```

```
Stack* createStack( int capacity )
{
    Stack* stack = (Stack *)malloc( sizeof( Stack ) );
    stack->top = -1;
    stack->capacity = capacity;
    stack->array = (Node **)malloc( stack->capacity * sizeof( Node* ) );
    return stack;
}
```

```
// A utility function to check if stack is full
```

```
int isFull( Stack* stack )
{
    return stack->top == stack->capacity - 1;
}
```

```
// A utility function to check if stack is empty
```

```

// A utility function to check if stack is empty
int isEmpty( Stack* stack )
{
    return stack->top == -1;
}

// A utility function to push an item to stack
void push( Stack* stack, Node* item )
{
    if( isFull( stack ) )
        return;
    stack->array[ ++stack->top ] = item;
}

// A utility function to remove an item from stack
Node* pop( Stack* stack )
{
    if( isEmpty( stack ) )
        return NULL;
    return stack->array[ stack->top-- ];
}

// A utility function to get top node of stack
Node* peek( Stack* stack )
{
    return stack->array[ stack->top ];
}

// The main function that constructs BST from 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 it
        // as the left child of the stack's top node. Push the new node
        else

```

```

    {
        peek( stack )->left = newNode( pre[i] );
        push( stack, peek( stack )->left );
    }
    return root;
}

// A utility function to print inorder traversal of a Binary Tree
void printInorder (Node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->data);
    printInorder(node->right);
}

// Driver program to test above functions
int main ()
{
    int pre[] = {10, 5, 1, 7, 40, 50};
    int size = sizeof( pre ) / sizeof( pre[0] );

    Node *root = constructTree(pre, size);

    printf("Inorder traversal of the constructed tree: \n");
    printInorder(root);

    return 0;
}

```

Output:

```
1 5 7 10 40 50
```

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 $O(n)$.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

65. 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 larger data than key value in left subtree, if not the root itself will be ceil node.

Here is code in C for ceil value.

```
// Program to find ceil of a given value in BST
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has key, left child and right child */
struct node
{
    int key;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the given key and
NULL left and right pointers.*/
struct node* newNode(int key)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

// Function to find ceil of a given input in BST. If input is more
// than the max key in BST, return -1
int Ceil(struct 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;
}
```

```
// Driver program to test above function
int main()
{
    node *root = newNode(8);

    root->left = newNode(4);
    root->right = newNode(12);

    root->left->left = newNode(2);
    root->left->right = newNode(6);

    root->right->left = newNode(10);
    root->right->right = newNode(14);

    for(int i = 0; i < 16; i++)
        printf("%d %d\n", i, Ceil(root, i));

    return 0;
}
```

Output:

```
0  2
1  2
2  2
3  4
4  4
5  6
6  6
7  8
8  8
9  10
10 10
11 12
12 12
13 14
14 14
15 -1
```

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.

— **Venki**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

66. 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.

```
#include <stdlib.h>
#include <stdio.h>
#include <iostream>
#include <stack>

using namespace std;

/* A binary tree node has data, left child and right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Helper function that allocates a new node with the given data and
NULL left and right pointers.*/
struct node* newNode(int data)
{
    struct node* node = new struct node;
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

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

    // Create an empty stack and push root to it
    stack<struct 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 b is done before c as right child has to be processed first)
    */
}
```

```

    c) push its left child
    Note that right child is pushed first so that left is processed fi
    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);
    }
}

```

// Driver program to test above functions

```

int main()
{
    /* Constructed binary tree is
        10
       /  \
      8    2
     /  \  /
    3   5 2
    */
    struct node *root = newNode(10);
    root->left = newNode(8);
    root->right = newNode(2);
    root->left->left = newNode(3);
    root->left->right = newNode(5);
    root->right->left = newNode(2);
    iterativePreorder(root);
    return 0;
}

```

Output:

```
10 8 3 5 2 2
```

This article is compiled by Saurabh Sharma and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

67. Convert a BST to a Binary Tree such that sum of all greater keys is added to every key

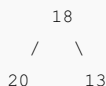
Given a Binary Search Tree (BST), convert it to a Binary Tree such that every key of the original BST is changed to key plus sum of all greater keys in BST.

Examples:

Input: Root of following BST



Output: The given BST is converted to following Binary Tree



Source: [Convert a BST](#)

Solution: Do reverse Inorder 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.

```
// Program to change a BST to Binary Tree such that key of a node becom
// original key plus sum of all greater keys in BST
```

```
#include <stdio.h>
#include <stdlib.h>
```

```
/* A BST node has key, left child and right child */
struct node
{
    int key;
    struct node* left;
    struct node* right;
};
```

```
/* Helper function that allocates a new node with the given key and
NULL left and right pointers.*/
```

```
struct node* newNode(int key)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    return (node);
}
```

```
// A recursive function that traverses the given BST in reverse inorde
// for every key, adds all greater keys to it
```

```
void addGreaterUtil(struct node *root, int *sum_ptr)
{
```

```
    // Base Case
    if (root == NULL)
        return;
```

```
    // Recur for right subtree first so that sum of all greater
    // nodes is stored at sum_ptr
    addGreaterUtil(root->right, sum_ptr);
```

```
    // Update the value at sum_ptr
```



```

    *sum_ptr = *sum_ptr + root->key;

    // Update key of this node
    root->key = *sum_ptr;

    // Recur for left subtree so that the updated sum is added
    // to smaller nodes
    addGreaterUtil(root->left, sum_ptr);
}

// A wrapper over addGreaterUtil(). It initializes sum and calls
// addGreaterUtil() to recursively update and use value of sum
void addGreater(struct node *root)
{
    int sum = 0;
    addGreaterUtil(root, &sum);
}

// A utility function to print inorder traversal of Binary Tree
void printInorder(struct node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    printf("%d ", node->key);
    printInorder(node->right);
}

// Driver program to test above function
int main()
{
    /* Create following BST
        5
       / \
      2  13
     */
    node *root = newNode(5);
    root->left = newNode(2);
    root->right = newNode(13);

    printf(" Inorder traversal of the given tree\n");
    printInorder(root);

    addGreater(root);

    printf("\n Inorder traversal of the modified tree\n");
    printInorder(root);

    return 0;
}

```

Output:

```

Inorder traversal of the given tree
2 5 13
Inorder traversal of the modified tree
20 18 13

```

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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

68. 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 C implementation of the above algorithm.

```
// C program for Morris Preorder traversal
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node *left, *right;
};

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* temp = (struct node*) malloc(sizeof(struct node));
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// 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 point
        // this node
        if (current->right == root)
        {
            current->right = NULL;
            root = root->right;
        }

        // If right child doesn't point to this node, then print t
        // node and make right child point to this node
        else
        {
            printf("%d ", root->data);
            current->right = root;
            root = root->left;
        }
    }
}
}
}
}

```

// Function for sStandard preorder traversal

```
void preorder(struct node* root)
```

```

{
    if (root)
    {
        printf( "%d ", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}

```

/* Driver program to test above functions*/

```

int main()
{
    struct node* root = NULL;

    root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);

    root->left->left = newNode(4);
    root->left->right = newNode(5);

    root->right->left = newNode(6);
    root->right->right = newNode(7);

    root->left->left->left = newNode(8);
    root->left->left->right = newNode(9);

    root->left->right->left = newNode(10);
    root->left->right->right = newNode(11);

    morrisTraversalPreorder(root);

    printf("\n");
}

```

```

preorder(root);

return 0;
}

```

Output:

```

1 2 4 8 9 5 10 11 3 6 7
1 2 4 8 9 5 10 11 3 6 7

```

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.

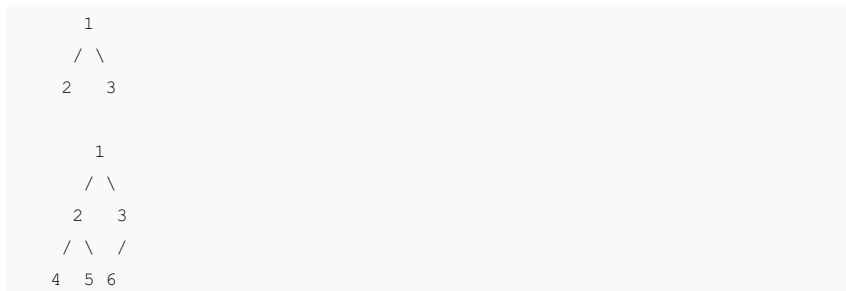
This article is compiled by **Aashish Barnwal** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

69. Linked complete binary tree & its creation

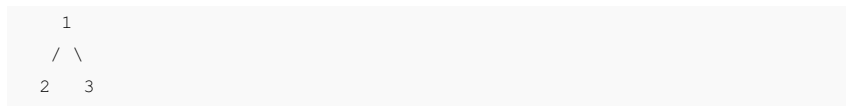
A complete binary tree is a binary tree where each level 'l' except the last has 2^l nodes and the nodes at the last level are all left aligned. Complete binary trees are mainly used in heap based data structures.

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.



Below binary trees are not complete:



```

/   /
4   5

      1
    /  \
   2    3
  / \  /
 4  5 6
 /
7

```

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.

Below is the implementation:

```

// Program for linked implementation of complete binary tree
#include <stdio.h>
#include <stdlib.h>

// For Queue Size
#define SIZE 50

// A tree node
struct node
{
    int data;
    struct node *right,*left;
};

```

```

..
// A queue node
struct Queue
{
    int front, rear;
    int size;
    struct node* *array;
};

// A utility function to create a new tree node
struct node* newNode(int data)
{
    struct node* temp = (struct node*) malloc(sizeof( struct node ));
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// A utility function to create a new Queue
struct Queue* createQueue(int size)
{
    struct Queue* queue = (struct Queue*) malloc(sizeof( struct Queue

    queue->front = queue->rear = -1;
    queue->size = size;

    queue->array = (struct node**) malloc(queue->size * sizeof( struct

    int i;
    for (i = 0; i < size; ++i)
        queue->array[i] = NULL;

    return queue;
}

// Standard Queue Functions
int isEmpty(struct Queue* queue)
{
    return queue->front == -1;
}

int isFull(struct Queue* queue)
{
    return queue->rear == queue->size - 1; }

int hasOnlyOneItem(struct Queue* queue)
{
    return queue->front == queue->rear; }

void Enqueue(struct node *root, struct Queue* queue)
{
    if (isFull(queue))
        return;

    queue->array[++queue->rear] = root;

    if (isEmpty(queue))
        ++queue->front;
}

struct node* Dequeue(struct Queue* queue)
{
    if (isEmpty(queue))
        return NULL;

```

```

    struct node* temp = queue->array[queue->front];

    if (hasOnlyOneItem(queue))
        queue->front = queue->rear = -1;
    else
        ++queue->front;

    return temp;
}

struct node* getFront(struct Queue* queue)
{ return queue->array[queue->front]; }

// A utility function to check if a tree node has both left and right child
int hasBothChild(struct node* temp)
{
    return temp && temp->left && temp->right;
}

// Function to insert a new node in complete binary tree
void insert(struct node **root, int data, struct Queue* queue)
{
    // Create a new node for given data
    struct node *temp = newNode(data);

    // If the tree is empty, initialize the root with new node.
    if (!*root)
        *root = temp;

    else
    {
        // get the front node of the queue.
        struct node* front = getFront(queue);

        // If the left child of this front node doesn't exist, set the
        // left child as the new node
        if (!front->left)
            front->left = temp;

        // If the right child of this front node doesn't exist, set the
        // right child as the new node
        else if (!front->right)
            front->right = temp;

        // If the front node has both the left child and right child,
        // Dequeue() it.
        if (hasBothChild(front))
            Dequeue(queue);
    }

    // Enqueue() the new node for later insertions
    Enqueue(temp, queue);
}

// Standard level order traversal to test above function
void levelOrder(struct node* root)
{
    struct Queue* queue = createQueue(SIZE);

    Enqueue(root, queue);

    while (!isEmpty(queue))

```

```

while (!isEmpty(queue))
{
    struct node* temp = Dequeue(queue);

    printf("%d ", temp->data);

    if (temp->left)
        Enqueue(temp->left, queue);

    if (temp->right)
        Enqueue(temp->right, queue);
}

// Driver program to test above functions
int main()
{
    struct node* root = NULL;
    struct Queue* queue = createQueue(SIZE);
    int i;

    for(i = 1; i <= 12; ++i)
        insert(&root, i, queue);

    levelOrder(root);

    return 0;
}

```

Output:

```
1 2 3 4 5 6 7 8 9 10 11 12
```

This article is compiled by **Aashish Barnwal** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

70. Ternary Search Tree

A ternary search tree is a special trie data structure where the child nodes of a standard trie are ordered as a binary search tree.

Representation of ternary search trees:

Unlike trie(standard) data structure where each node contains 26 pointers for its children, each node in a ternary search tree contains only 3 pointers:

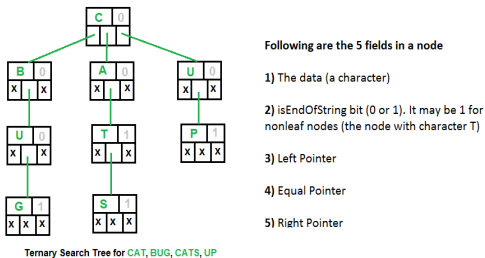
1. The left pointer points to the node whose value is less than the value in the current node.
2. The equal pointer points to the node whose value is equal to the value in the current node.
3. The right pointer points to the node whose value is greater than the value in the

current node.

Apart from above three pointers, each node has a field to indicate data(character in case of dictionary) and another field to mark end of a string.

So, more or less it is similar to BST which stores data based on some order. However, data in a ternary search tree is distributed over the nodes. e.g. It needs 4 nodes to store the word “Geek”.

Below figure shows how exactly the words in a ternary search tree are stored?



One of the advantage of using ternary search trees over tries is that ternary search trees are a more space efficient (involve only three pointers per node as compared to 26 in standard tries). Further, ternary search trees can be used any time a hashtable would be used to store strings.

Tries are suitable when there is a proper distribution of words over the alphabets so that spaces are utilized most efficiently. Otherwise ternary search trees are better. Ternary search trees are efficient to use(in terms of space) when the strings to be stored share a common prefix.

Applications of ternary search trees:

1. Ternary search trees are efficient for queries like “Given a word, find the next word in dictionary(near-neighbor lookups)” or “Find all telephone numbers starting with 9342 or “typing few starting characters in a web browser displays all website names with this prefix”(Auto complete feature)”.
2. Used in spell checks: Ternary search trees can be used as a dictionary to store all the words. Once the word is typed in an editor, the word can be parallely searched in the ternary search tree to check for correct spelling.

Implementation:

Following is C implementation of ternary search tree. The operations implemented are, search, insert and traversal.

```
// C program to demonstrate Ternary Search Tree (TST) insert, traverse
// and search operations
#include <stdio.h>
#include <stdlib.h>
#define MAX 50

// A node of ternary search tree
struct Node
```

```

struct Node
{
    char data;

    // True if this character is last character of one of the words
    unsigned isEndOfString: 1;

    struct Node *left, *eq, *right;
};

// A utility function to create a new ternary search tree node
struct Node* newNode(char data)
{
    struct Node* temp = (struct Node*) malloc(sizeof( struct Node ));
    temp->data = data;
    temp->isEndOfString = 0;
    temp->left = temp->eq = temp->right = NULL;
    return temp;
}

// Function to insert a new word in a Ternary Search Tree
void insert(struct Node** root, char *word)
{
    // Base Case: Tree is empty
    if (!(*root))
        *root = newNode(*word);

    // If current character of word is smaller than root's character,
    // then insert this word in left subtree of root
    if ((*word) < (*root)->data)
        insert(& (*root)->left , word);

    // If current character of word is greater than root's character,
    // then insert this word in right subtree of root
    else if ((*word) > (*root)->data)
        insert(& (*root)->right , word);

    // If current character of word is same as root's character,
    else
    {
        if (*(word+1))
            insert(& (*root)->eq , word+1);

        // the last character of the word
        else
            (*root)->isEndOfString = 1;
    }
}

// A recursive function to traverse Ternary Search Tree
void traverseTSTUtil(struct Node* root, char* buffer, int depth)
{
    if (root)
    {
        // First traverse the left subtree
        traverseTSTUtil(root->left, buffer, depth);

        // Store the character of this node
        buffer[depth] = root->data;
        if (root->isEndOfString)
        {
            buffer[depth+1] = '\0';
            printf(" %s\n", buffer);
        }
    }
}

```

```

    }

    // Traverse the subtree using equal pointer (middle subtree)
    traverseTSTUtil(root->eq, buffer, depth + 1);

    // Finally Traverse the right subtree
    traverseTSTUtil(root->right, buffer, depth);
}

// The main function to traverse a Ternary Search Tree.
// It mainly uses traverseTSTUtil()
void traverseTST(struct Node* root)
{
    char buffer[MAX];
    traverseTSTUtil(root, buffer, 0);
}

// Function to search a given word in TST
int searchTST(struct Node *root, char *word)
{
    if (!root)
        return 0;

    if (*word < (root)->data)
        return searchTST(root->left, word);

    else if (*word > (root)->data)
        return searchTST(root->right, word);

    else
    {
        if (*(word+1) == '\0')
            return root->isEndOfString;

        return searchTST(root->eq, word+1);
    }
}

// Driver program to test above functions
int main()
{
    struct Node *root = NULL;

    insert(&root, "cat");
    insert(&root, "cats");
    insert(&root, "up");
    insert(&root, "bug");

    printf("Following is traversal of ternary search tree\n");
    traverseTST(root);

    printf("\nFollowing are search results for cats, bu and cat respec
searchTST(root, "cats")? printf("Found\n"): printf("Not Found\n");
searchTST(root, "bu")? printf("Found\n"): printf("Not Found\n");
searchTST(root, "cat")? printf("Found\n"): printf("Not Found\n");

    return 0;
}

```

Output:

Following is traversal of ternary search tree

bug
cat
cats
up

Following are search results for cats, bu and cat respectively

Found

Not Found

Found

Time Complexity: The time complexity of the ternary search tree operations is similar to that of binary search tree. i.e. the insertion, deletion and search operations take time proportional to the height of the ternary search tree. The space is proportional to the length of the string to be stored.

Reference:

http://en.wikipedia.org/wiki/Ternary_search_tree

This article is compiled by **Aashish Barnwal** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

71. Segment Tree | Set 1 (Sum of given range)

Let us consider the following problem to understand Segment Trees.

We have an array $arr[0 \dots n-1]$. We should be able to

- 1 Find the sum of elements from index l to r where $0 \leq l \leq r \leq n-1$
- 2 Change value of a specified element of the array $arr[i] = x$ where $0 \leq i \leq n-1$.

A **simple solution** is to run a loop from l to r and calculate sum of elements in given range. To update a value, simply do $arr[i] = x$. The first operation takes $O(n)$ time and second operation takes $O(1)$ time.

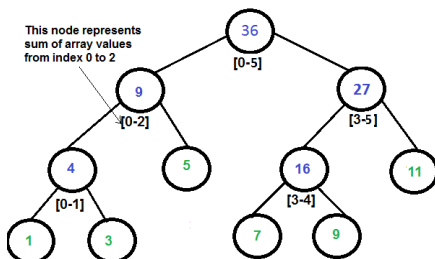
Another solution is to create another array and store sum from start to i at the i th index in this array. Sum of a given range can now be calculated in $O(1)$ time, but update operation takes $O(n)$ time now. This works well if the number of query operations are large and very few updates.

What if the number of query and updates are equal? **Can we perform both the operations in $O(\log n)$ time once given the array?** We can use a Segment Tree to do both operations in $O(\log n)$ time.

Representation of Segment trees

1. Leaf Nodes are the elements of the input array.
2. Each internal node represents some merging of the leaf nodes. The merging may be different for different problems. For this problem, merging is sum of leaves under a node.

An array representation of tree is used to represent Segment Trees. For each node at index i , the left child is at index $2*i+1$, right child at $2*i+2$ and the parent is at $\lfloor (i-1)/2 \rfloor$.



Segment Tree for input array {1, 3, 5, 7, 9, 11}

Construction of Segment Tree from given array

We start with a segment $arr[0 \dots n-1]$. and every time we divide the current segment into two halves (if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment we store the sum in corresponding node.

All levels of the constructed segment tree will be completely filled except the last level.

Also, the tree will be a **Full Binary Tree** because we always divide segments in two halves at every level. Since the constructed tree is always full binary tree with n leaves, there will be $n-1$ internal nodes. So total number of nodes will be $2*n - 1$.

Height of the segment tree will be $\lceil \log_2 n \rceil$. Since the tree is represented using array and relation between parent and child indexes must be maintained, size of memory allocated for segment tree will be $2 * 2^{\lceil \log_2 n \rceil} - 1$.

Query for Sum of given range

Once the tree is constructed, how to get the sum using the constructed segment tree. Following is algorithm to get the sum of elements.

```
int getSum(node, l, r)
{
    if range of node is within l and r
        return value in node
    else if range of node is completely outside l and r
        return 0
    else
        return getSum(node's left child, l, r) +
               getSum(node's right child, l, r)
```

```
}
```

Update a value

Like tree construction and query operations, update can also be done recursively. We are given an index which needs to be updated. Let *diff* be the value to be added. We start from root of the segment tree, and add *diff* to all nodes which have given index in their range. If a node doesn't have given index in its range, we don't make any changes to that node.

Implementation:

Following is implementation of segment tree. The program implements construction of segment tree for any given array. It also implements query and update operations.

```
// Program to show segment tree operations like construction, query and
#include <stdio.h>
#include <math.h>

// A utility function to get the middle index from corner indexes.
int getMid(int s, int e) { return s + (e - s)/2; }

/* A recursive function to get the sum of values in given range of the array.
The following are parameters for this function.

st    --> Pointer to segment tree
index --> Index of current node in the segment tree. Initially 0 is
         passed as root is always at index 0
ss & se --> Starting and ending indexes of the segment represented by
         current node, i.e., st[index]
qs & qe --> Starting and ending indexes of query range */
int getSumUtil(int *st, int ss, int se, int qs, int qe, int index)
{
    // If segment of this node is a part of given range, then return the
    // sum of the segment
    if (qs <= ss && qe >= se)
        return st[index];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
        return 0;

    // If a part of this segment overlaps with the given range
    int mid = getMid(ss, se);
    return getSumUtil(st, ss, mid, qs, qe, 2*index+1) +
           getSumUtil(st, mid+1, se, qs, qe, 2*index+2);
}

/* A recursive function to update the nodes which have the given index
in their range. The following are parameters
st, index, ss and se are same as getSumUtil()
i    --> index of the element to be updated. This index is in input array
diff --> Value to be added to all nodes which have i in range */
void updateValueUtil(int *st, int ss, int se, int i, int diff, int index)
{
    // Base Case: If the input index lies outside the range of this segment
    if (i < ss || i > se)
        return;

    // If the input index is in range of this node, then update the value
    // of the node and its children
```

```

    st[index] = st[index] + diff;
    if (se != ss)
    {
        int mid = getMid(ss, se);
        updateValueUtil(st, ss, mid, i, diff, 2*index + 1);
        updateValueUtil(st, mid+1, se, i, diff, 2*index + 2);
    }
}

// The function to update a value in input array and segment tree.
// It uses updateValueUtil() to update the value in segment tree
void updateValue(int arr[], int *st, int n, int i, int new_val)
{
    // Check for erroneous input index
    if (i < 0 || i > n-1)
    {
        printf("Invalid Input");
        return;
    }

    // Get the difference between new value and old value
    int diff = new_val - arr[i];

    // Update the value in array
    arr[i] = new_val;

    // Update the values of nodes in segment tree
    updateValueUtil(st, 0, n-1, i, diff, 0);
}

// Return sum of elements in range from index qs (query start) to
// qe (query end). It mainly uses getSumUtil()
int getSum(int *st, int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Invalid Input");
        return -1;
    }

    return getSumUtil(st, 0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for array[ss..se]
// si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int *st, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se)
    {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the sum of values in this node
    int mid = getMid(ss, se);
    st[si] = constructSTUtil(arr, ss, mid, st, si*2+1) +
        constructSTUtil(arr, mid+1, se, st, si*2+2);
    return st[si];
}

```

```

}

/* Function to construct segment tree from given array. This function
allocates memory for segment tree and calls constructSTUtil() to
fill the allocated memory */
int *constructST(int arr[], int n)
{
    // Allocate memory for segment tree
    int x = (int)(ceil(log2(n))); //Height of segment tree
    int max_size = 2*(int)pow(2, x) - 1; //Maximum size of segment tree
    int *st = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n-1, st, 0);

    // Return the constructed segment tree
    return st;
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Build segment tree from given array
    int *st = constructST(arr, n);

    // Print sum of values in array from index 1 to 3
    printf("Sum of values in given range = %d\n", getSum(st, n, 1, 3))

    // Update: set arr[1] = 10 and update corresponding segment
    // tree nodes
    updateValue(arr, st, n, 1, 10);

    // Find sum after the value is updated
    printf("Updated sum of values in given range = %d\n",
        getSum(st, n, 1, 3))

    return 0;
}

```

Output:

```

Sum of values in given range = 15
Updated sum of values in given range = 22

```

Time Complexity:

Time Complexity for tree construction is $O(n)$. There are total $2n-1$ nodes, and value of every node is calculated only once in tree construction.

Time complexity to query is $O(\text{Log}n)$. To query a sum, we process at most four nodes at every level and number of levels is $O(\text{Log}n)$.

The time complexity of update is also $O(\text{Log}n)$. To update a leaf value, we process one node at every level and number of levels is $O(\text{Log}n)$.

Segment Tree | Set 2 (Range Minimum Query)

References:

<http://www.cse.iitk.ac.in/users/aca/lop12/slides/06.pdf>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

72. Segment Tree | Set 2 (Range Minimum Query)

We have introduced [segment tree with a simple example](#) in the previous post. In this post, [Range Minimum Query](#) problem is discussed as another example where Segment Tree can be used. Following is problem statement.

We have an array $arr[0 \dots n-1]$. We should be able to efficiently find the minimum value from index qs (query start) to qe (query end) where $0 \leq qs \leq qe \leq n-1$. The array is static (elements are not deleted and inserted during the series of queries).

A **simple solution** is to run a loop from qs to qe and find minimum element in given range. This solution takes $O(n)$ time in worst case.

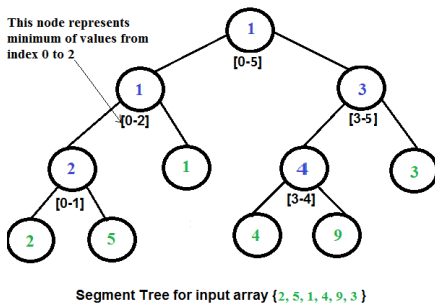
Another solution is to create a 2D array where an entry $[i, j]$ stores the minimum value in range $arr[i..j]$. Minimum of a given range can now be calculated in $O(1)$ time, but preprocessing takes $O(n^2)$ time. Also, this approach needs $O(n^2)$ extra space which may become huge for large input arrays.

Segment tree can be used to do preprocessing and query in moderate time. With segment tree, preprocessing time is $O(n)$ and time to for range minimum query is $O(\log n)$. The extra space required is $O(n)$ to store the segment tree.

Representation of Segment trees

1. Leaf Nodes are the elements of the input array.
2. Each internal node represents minimum of all leaves under it.

An array representation of tree is used to represent Segment Trees. For each node at index i , the left child is at index $2*i+1$, right child at $2*i+2$ and the parent is at $\lfloor (i-1)/2 \rfloor$.



Construction of Segment Tree from given array

We start with a segment $arr[0 \dots n-1]$. and every time we divide the current segment into two halves(if it has not yet become a segment of length 1), and then call the same procedure on both halves, and for each such segment, we store the minimum value in a segment tree node.

All levels of the constructed segment tree will be completely filled except the last level.

Also, the tree will be a **Full Binary Tree** because we always divide segments in two halves at every level. Since the constructed tree is always full binary tree with n leaves, there will be $n-1$ internal nodes. So total number of nodes will be $2*n - 1$.

Height of the segment tree will be $\lceil \log_2 n \rceil$. Since the tree is represented using array and relation between parent and child indexes must be maintained, size of memory allocated for segment tree will be $2 * 2^{\lceil \log_2 n \rceil} - 1$.

Query for minimum value of given range

Once the tree is constructed, how to do range minimum query using the constructed segment tree. Following is algorithm to get the minimum.

```
// qs --> query start index, qe --> query end index
int RMQ(node, qs, qe)
{
    if range of node is within qs and qe
        return value in node
    else if range of node is completely outside qs and qe
        return INFINITE
    else
        return min( RMQ(node's left child, qs, qe), RMQ(node's right child, qs, qe) )
}
```

Implementation:

```
// Program for range minimum query using segment tree
#include <stdio.h>
#include <math.h>
#include <limits.h>
```

```
// A utility function to get minimum of two numbers
int minVal(int x, int y) { return (x < y)? x: y; }
```

```

// A utility function to get the middle index from corner indexes.
int getMid(int s, int e) { return s + (e - s)/2; }

/* A recursive function to get the minimum value in a given range of
   indexes. The following are parameters for this function.

   st    --> Pointer to segment tree
   index --> Index of current node in the segment tree. Initially 0 is
             passed as root is always at index 0
   ss & se --> Starting and ending indexes of the segment represented
               current node, i.e., st[index]
   qs & qe --> Starting and ending indexes of query range */
int RMQUtil(int *st, int ss, int se, int qs, int qe, int index)
{
    // If segment of this node is a part of given range, then return the
    // min of the segment
    if (qs <= ss && qe >= se)
        return st[index];

    // If segment of this node is outside the given range
    if (se < qs || ss > qe)
        return INT_MAX;

    // If a part of this segment overlaps with the given range
    int mid = getMid(ss, se);
    return minVal(RMQUtil(st, ss, mid, qs, qe, 2*index+1),
                  RMQUtil(st, mid+1, se, qs, qe, 2*index+2));
}

// Return minimum of elements in range from index qs (query start) to
// qe (query end). It mainly uses RMQUtil()
int RMQ(int *st, int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n-1 || qs > qe)
    {
        printf("Invalid Input");
        return -1;
    }

    return RMQUtil(st, 0, n-1, qs, qe, 0);
}

// A recursive function that constructs Segment Tree for array[ss..se]
// si is index of current node in segment tree st
int constructSTUtil(int arr[], int ss, int se, int *st, int si)
{
    // If there is one element in array, store it in current node of
    // segment tree and return
    if (ss == se)
    {
        st[si] = arr[ss];
        return arr[ss];
    }

    // If there are more than one elements, then recur for left and
    // right subtrees and store the minimum of two values in this node
    int mid = getMid(ss, se);
    st[si] = minVal(constructSTUtil(arr, ss, mid, st, si*2+1),
                    constructSTUtil(arr, mid+1, se, st, si*2+2));
    return st[si];
}

```

```

/* Function to construct segment tree from given array. This function
allocates memory for segment tree and calls constructSTUtil() to
fill the allocated memory */
int *constructST(int arr[], int n)
{
    // Allocate memory for segment tree
    int x = (int)(ceil(log2(n))); //Height of segment tree
    int max_size = 2*(int)pow(2, x) - 1; //Maximum size of segment tree
    int *st = new int[max_size];

    // Fill the allocated memory st
    constructSTUtil(arr, 0, n-1, st, 0);

    // Return the constructed segment tree
    return st;
}

// Driver program to test above functions
int main()
{
    int arr[] = {1, 3, 2, 7, 9, 11};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Build segment tree from given array
    int *st = constructST(arr, n);

    int qs = 1; // Starting index of query range
    int qe = 5; // Ending index of query range

    // Print minimum value in arr[qs..qe]
    printf("Minimum of values in range [%d, %d] is = %d\n",
           qs, qe, RMQ(st, n, qs, qe));

    return 0;
}

```

Output:

```
Minimum of values in range [1, 5] is = 2
```

Time Complexity:

Time Complexity for tree construction is $O(n)$. There are total $2n-1$ nodes, and value of every node is calculated only once in tree construction.

Time complexity to query is $O(\log n)$. To query a range minimum, we process at most two nodes at every level and number of levels is $O(\log n)$.

Please refer following links for more solutions to range minimum query problem.

[http://community.topcoder.com/tc?](http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Range_Minimum_Query_(RMQ))

[module=Static&d1=tutorials&d2=lowestCommonAncestor#Range_Minimum_Query_\(RMQ\)](http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor#Range_Minimum_Query_(RMQ))

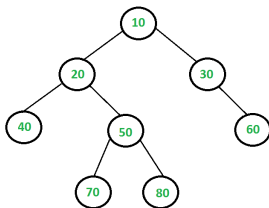
http://wcipeg.com/wiki/Range_minimum_query

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

73. Dynamic Programming | Set 26 (Largest Independent Set Problem)

Given a Binary Tree, find size of the **Largest Independent Set(LIS)** in it. A subset of all tree nodes is an independent set if there is no edge between any two nodes of the subset.

For example, consider the following binary tree. The largest independent set(LIS) is {10, 40, 60, 70, 80} and size of the LIS is 5.



A Dynamic Programming solution solves a given problem using solutions of subproblems in bottom up manner. Can the given problem be solved using solutions to subproblems? If yes, then what are the subproblems? Can we find largest independent set size (LISS) for a node X if we know LISS for all descendants of X? If a node is considered as part of LIS, then its children cannot be part of LIS, but its grandchildren can be. Following is optimal substructure property.

1) Optimal Substructure:

Let $LISS(X)$ indicates size of largest independent set of a tree with root X.

$$LISS(X) = \text{MAX} \{ (1 + \text{sum of LISS for all grandchildren of X}), (\text{sum of LISS for all children of X}) \}$$

The idea is simple, there are two possibilities for every node X, either X is a member of the set or not a member. If X is a member, then the value of $LISS(X)$ is 1 plus $LISS$ of all grandchildren. If X is not a member, then the value is sum of $LISS$ of all children.

2) Overlapping Subproblems

Following is recursive implementation that simply follows the recursive structure mentioned above.

```
// A naive recursive implementation of Largest Independent Set problem
#include <stdio.h>
#include <stdlib.h>

// A utility function to find max of two integers
int max(int x, int y) { return (x > y)? x: y; }
```

```

/* A binary tree node has data, pointer to left child and a pointer to
   right child */
struct node
{
    int data;
    struct node *left, *right;
};

// The function returns size of the largest independent set in a given
// binary tree
int LISS(struct node *root)
{
    if (root == NULL)
        return 0;

    // Caculate size excluding the current node
    int size_excl = LISS(root->left) + LISS(root->right);

    // Calculate size including the current node
    int size_incl = 1;
    if (root->left)
        size_incl += LISS(root->left->left) + LISS(root->left->right);
    if (root->right)
        size_incl += LISS(root->right->left) + LISS(root->right->right);

    // Return the maximum of two sizes
    return max(size_incl, size_excl);
}

// A utility function to create a node
struct node* newNode( int data )
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree given in the above diagram
    struct node *root = newNode(20);
    root->left = newNode(8);
    root->left->left = newNode(4);
    root->left->right = newNode(12);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(14);
    root->right = newNode(22);
    root->right->right = newNode(25);

    printf ("Size of the Largest Independent Set is %d ", LISS(root));

    return 0;
}

```

Output:

```
Size of the Largest Independent Set is 5
```

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. For example, LISS of node with value 50 is evaluated for node with values 10 and 20 as 50 is grandchild of 10 and child of 20.

Since same subproblems are called again, this problem has Overlapping Subproblems property. So LISS problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical [Dynamic Programming\(DP\) problems](#), recomputations of same subproblems can be avoided by storing the solutions to subproblems and solving problems in bottom up manner.

Following is C implementation of Dynamic Programming based solution. In the following solution, an additional field 'liss' is added to tree nodes. The initial value of 'liss' is set as 0 for all nodes. The recursive function LISS() calculates 'liss' for a node only if it is not already set.

```
/* Dynamic programming based program for Largest Independent Set problem */
#include <stdio.h>
#include <stdlib.h>

// A utility function to find max of two integers
int max(int x, int y) { return (x > y)? x: y; }

/* A binary tree node has data, pointer to left child and a pointer to right child */
struct node
{
    int data;
    int liss;
    struct node *left, *right;
};

// A memoization function returns size of the largest independent set
// a given binary tree
int LISS(struct node *root)
{
    if (root == NULL)
        return 0;

    if (root->liss)
        return root->liss;

    if (root->left == NULL && root->right == NULL)
        return (root->liss = 1);

    // Calculate size excluding the current node
    int liss_excl = LISS(root->left) + LISS(root->right);

    // Calculate size including the current node
    int liss_incl = 1;
    if (root->left)
        liss_incl += LISS(root->left->left) + LISS(root->left->right);
    if (root->right)
        liss_incl += LISS(root->right->left) + LISS(root->right->right);

    // Maximum of two sizes is LISS, store it for future uses.
    root->liss = max(liss_incl, liss_excl);

    return root->liss;
}
```

```

    return root->liss;
}

// A utility function to create a node
struct node* newNode(int data)
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );
    temp->data = data;
    temp->left = temp->right = NULL;
    temp->liss = 0;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree given in the above diagram
    struct node *root      = newNode(20);
    root->left              = newNode(8);
    root->left->left          = newNode(4);
    root->left->right         = newNode(12);
    root->left->right->left    = newNode(10);
    root->left->right->right   = newNode(14);
    root->right              = newNode(22);
    root->right->right        = newNode(25);

    printf ("Size of the Largest Independent Set is %d ", LISS(root));

    return 0;
}

```

Output

```
Size of the Largest Independent Set is 5
```

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

Following extensions to above solution can be tried as an exercise.

- 1) Extend the above solution for n -ary tree.
- 2) The above solution modifies the given tree structure by adding an additional field 'liss' to tree nodes. Extend the solution so that it doesn't modify the tree structure.
- 3) The above solution only returns size of LIS, it doesn't print elements of LIS. Extend the solution to print all nodes that are part of LIS.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

74. Iterative Postorder Traversal | Set 1 (Using Two Stacks)

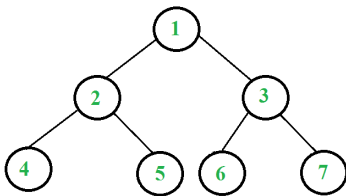
We have discussed [iterative inorder](#) and [iterative preorder](#) traversals. In this post, iterative postorder traversal is discussed which is more complex than the other two traversals (due to its nature of [non-tail recursion](#), there is an extra statement after the final recursive call to itself). 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

Let us consider the following tree



Following are the steps to print postorder traversal of the above tree using two stacks.

1. Push 1 to first stack.
First stack: 1
Second stack: Empty
2. Pop 1 from first stack and push it to second stack.
Push left and right children of 1 to first stack
First stack: 2, 3
Second stack: 1
3. Pop 3 from first stack and push it to second stack.

```

    Push left and right children of 3 to first stack
    First stack: 2, 6, 7
    Second stack:1, 3

4. Pop 7 from first stack and push it to second stack.
    First stack: 2, 6
    Second stack:1, 3, 7

5. Pop 6 from first stack and push it to second stack.
    First stack: 2
    Second stack:1, 3, 7, 6

6. Pop 2 from first stack and push it to second stack.
    Push left and right children of 2 to first stack
    First stack: 4, 5
    Second stack:1, 3, 7, 6, 2

7. Pop 5 from first stack and push it to second stack.
    First stack: 4
    Second stack: 1, 3, 7, 6, 2, 5

8. Pop 4 from first stack and push it to second stack.
    First stack: Empty
    Second stack: 1, 3, 7, 6, 2, 5, 4

```

The algorithm stops since there is no more item in first stack.
Observe that content of second stack is in postorder fashion. Print them.

Following is C implementation of iterative postorder traversal using two stacks.

```

#include <stdio.h>
#include <stdlib.h>

// Maximum stack size
#define MAX_SIZE 100

// A tree node
struct Node
{
    int data;
    struct Node *left, *right;
};

// Stack type
struct Stack
{
    int size;
    int top;
    struct Node* *array;
};

// A utility function to create a new tree node

```

```

// ...
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// A utility function to create a stack of given size
struct Stack* createStack(int size)
{
    struct Stack* stack =
        (struct Stack*) malloc(sizeof(struct Stack));
    stack->size = size;
    stack->top = -1;
    stack->array =
        (struct Node**) malloc(stack->size * sizeof(struct Node));
    return stack;
}

// BASIC OPERATIONS OF STACK
int isFull(struct Stack* stack)
{ return stack->top - 1 == stack->size; }

int isEmpty(struct Stack* stack)
{ return stack->top == -1; }

void push(struct Stack* stack, struct Node* node)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = node;
}

struct Node* pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top--];
}

// An iterative function to do post order traversal of a given binary
void postOrderIterative(struct Node* root)
{
    // 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);
    }
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree shown in above figure
    struct Node* root = NULL;
    root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);

    postOrderIterative(root);

    return 0;
}

```

Output:

```
4 5 2 6 7 3 1
```

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](#).

You may also like to see [a method which uses only one stack](#).

This article is compiled by [Aashish Barnwal](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

75. Iterative Postorder Traversal | Set 2 (Using One 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.

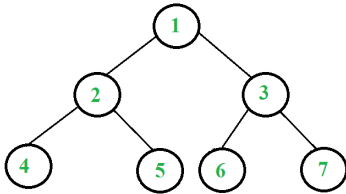
The idea is to move down to leftmost node using left pointer. While moving down, push

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.
```

Let us consider the following tree



Following are the steps to print postorder traversal of the above tree using one stack.

```
1. Right child of 1 exists.
   Push 3 to stack. Push 1 to stack. Move to left child.
   Stack: 3, 1

2. Right child of 2 exists.
   Push 5 to stack. Push 2 to stack. Move to left child.
   Stack: 3, 1, 5, 2

3. Right child of 4 doesn't exist. '
   Push 4 to stack. Move to left child.
   Stack: 3, 1, 5, 2, 4

4. Current node is NULL.
   Pop 4 from stack. Right child of 4 doesn't exist.
   Print 4. Set current node to NULL.
   Stack: 3, 1, 5, 2

5. Current node is NULL.
   Pop 2 from stack. Since right child of 2 equals stack top element,
```

```

    pop 5 from stack. Now push 2 to stack.
    Move current node to right child of 2 i.e. 5
    Stack: 3, 1, 2

6. Right child of 5 doesn't exist. Push 5 to stack. Move to left child.
    Stack: 3, 1, 2, 5

7. Current node is NULL. Pop 5 from stack. Right child of 5 doesn't exist.
    Print 5. Set current node to NULL.
    Stack: 3, 1, 2

8. Current node is NULL. Pop 2 from stack.
    Right child of 2 is not equal to stack top element.
    Print 2. Set current node to NULL.
    Stack: 3, 1

9. Current node is NULL. Pop 1 from stack.
    Since right child of 1 equals stack top element, pop 3 from stack.
    Now push 1 to stack. Move current node to right child of 1 i.e. 3
    Stack: 1

10. Repeat the same as above steps and Print 6, 7 and 3.
    Pop 1 and Print 1.

```

```

// C program for iterative postorder traversal using one stack
#include <stdio.h>
#include <stdlib.h>

// Maximum stack size
#define MAX_SIZE 100

// A tree node
struct Node
{
    int data;
    struct Node *left, *right;
};

// Stack type
struct Stack
{
    int size;
    int top;
    struct Node* *array;
};

// A utility function to create a new tree node
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

```

```

,

// A utility function to create a stack of given size
struct Stack* createStack(int size)
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack))
    stack->size = size;
    stack->top = -1;
    stack->array = (struct Node**) malloc(stack->size * sizeof(struct Node))
    return stack;
}

// BASIC OPERATIONS OF STACK
int isFull(struct Stack* stack)
{ return stack->top - 1 == stack->size; }

int isEmpty(struct Stack* stack)
{ return stack->top == -1; }

void push(struct Stack* stack, struct Node* node)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = node;
}

struct Node* pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top--];
}

struct Node* peek(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top];
}

// 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
        // processed yet, then make sure right child is processed before
        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));
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree shown in above figure
    struct Node* root = NULL;
    root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);

    postOrderIterative(root);

    return 0;
}

```

Output:

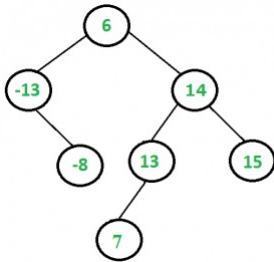
```
4 5 2 6 7 3 1
```

This article is compiled by [Aashish Barnwal](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

76. Find if there is a triplet in a Balanced BST that adds to zero

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 $O(\log n)$ extra space can be used. You can modify given Binary Search Tree. Note that height of a Balanced BST is always $O(\log n)$

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 $O(\text{Log}n)$ 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. To find the pair, we can use the approach used in `hasArrayTwoCandidates()` in method 1 of [this](#) post.

```

// A C++ program to check if there is a triplet with sum equal to 0 in
// a given BST
#include<stdio.h>

// A BST node has key, and left and right pointers
struct node
{
    int key;
    struct node *left;
    struct node *right;
};

// A function to convert given BST to Doubly Linked List. left pointer
// as previous pointer and right pointer is used as next pointer. The
// sets *head to point to first and *tail to point to last node of con
void convertBSTtoDLL(node* root, node** head, node** tail)
{
    // Base case
    if (root == NULL)
        return;

    // First convert the left subtree
    if (root->left)
        convertBSTtoDLL(root->left, head, tail);

    // Then change left of current root as last node of left subtree
    root->left = *tail;

```

```

// If tail is not NULL, then set right of tail as root, else current
// node is head
if (*tail)
    (*tail)->right = root;
else
    *head = root;

// Update tail
*tail = root;

// Finally, convert right subtree
if (root->right)
    convertBSTtoDLL(root->right, head, tail);
}

// This function returns true if there is pair in DLL with sum equal
// to given sum. The algorithm is similar to hasArrayTwoCandidates()
// in method 1 of http://tinyurl.com/dy6palr
bool isPresentInDLL(node* head, node* tail, int sum)
{
    while (head != tail)
    {
        int curr = head->key + tail->key;
        if (curr == sum)
            return true;
        else if (curr > sum)
            tail = tail->left;
        else
            head = head->right;
    }
    return false;
}

// 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 DLL
    node* head = NULL;
    node* tail = NULL;
    convertBSTtoDLL(root, &head, &tail);

    // Now iterate through every node and find if there is a pair with
    // equal to -1 * head->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;
}

```

```

}

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

// A utility function to insert a given key to BST
node* insert(node* root, int key)
{
    if (root == NULL)
        return newNode(key);
    if (root->key > key)
        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);
    return root;
}

// Driver program to test above functions
int main()
{
    node* root = NULL;
    root = insert(root, 6);
    root = insert(root, -13);
    root = insert(root, 14);
    root = insert(root, -8);
    root = insert(root, 15);
    root = insert(root, 13);
    root = insert(root, 7);
    if (isTripletPresent(root))
        printf("Present");
    else
        printf("Not Present");
    return 0;
}

```

Output:

```
Present
```

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 $O(\log n)$), the number of functions in call stack will never be more than $O(\log n)$.

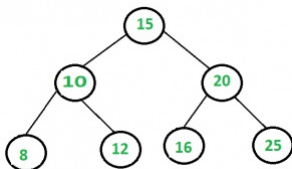
We can also find triplet in same time and extra space without modifying the tree. See [next](#) post. The code discussed there can be used to find triplet also.

This article is compiled by [Ashish Anand](#) and reviewed by GeeksforGeeks team. Please

write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

77. Find a pair with given sum in a Balanced 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 $O(\log n)$ extra space can be used. Any modification to Binary Search Tree is not allowed. Note that height of a Balanced BST is always $O(\log n)$.



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 $O(\log n)$ extra space, but it modifies the given BST.

The **solution discussed below takes $O(n)$ time, $O(\log n)$ 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. Following is C++ implementation of this approach.

```

/* In a balanced binary search tree isPairPresent two element which sum
   a given value time O(n) space O(logn) */
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 100

// A BST node
struct node
{
    int val;
    struct node *left, *right;
};

// Stack type
struct Stack
{
    int size;
    int top;
    struct node* array;
};

// A utility function to create a stack of given size
struct Stack* createStack(int size)
{
    struct Stack* stack =
        (struct Stack*) malloc(sizeof(struct Stack));
    stack->size = size;
    stack->top = -1;
    stack->array =
        (struct node**) malloc(stack->size * sizeof(struct node));
    return stack;
}

// BASIC OPERATIONS OF STACK
int isFull(struct Stack* stack)
{
    return stack->top - 1 == stack->size; }

int isEmpty(struct Stack* stack)
{
    return stack->top == -1; }

void push(struct Stack* stack, struct node* node)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = node;
}

struct node* pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top--];
}

// 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
// fix stack size as O(Logn) for balanced trees like AVL and Red B
// tree. We have used MAX_SIZE to keep the code simple

// done1, val1 and curr1 are used for normal inorder traversal usi
// done2, val2 and curr2 are used for reverse inorder traversal us
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 tw
// traversals is complete
while (1)
{
    // Find next node in normal Inorder traversal. See following p
    // http://www.geeksforgeeks.org/inorder-tree-traversal-without
    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

```

```

// 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
// normal inorder traversal
else if ((val1 + val2) < target)
    done1 = false;

// If sum of current values is greater, then move to next node
// 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;
}
}

```

// A utility function to create BST node

```

struct node * NewNode(int val)
{
    struct node *tmp = (struct node *)malloc(sizeof(struct node));
    tmp->val = val;
    tmp->right = tmp->left = NULL;
    return tmp;
}

```

// Driver program to test above functions

```

int main()
{
    /*
        15
       / \
      10  20
     / \ / \
    8  12 16 25
    */
    struct node *root = NewNode(15);
    root->left = NewNode(10);
    root->right = NewNode(20);
    root->left->left = NewNode(8);
    root->left->right = NewNode(12);
    root->right->left = NewNode(16);
    root->right->right = NewNode(25);

    int target = 33;
    if (isPairPresent(root, target) == false)
        printf("\n No such values are found\n");

    getchar();
    return 0;
}

```

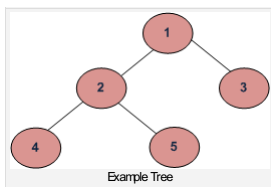
Output:

```
Pair Found: 8 + 25 = 33
```

This article is compiled by [Kumar](#) and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

78. 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.



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.

```
// A recursive C program to print REVERSE level order traversal
#include <stdio.h>
#include <stdlib.h>

/* A binary tree node has data, pointer to left and right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/*Function prototypes*/
void printGivenLevel(struct node* root, int level);
int height(struct node* node);
struct node* newNode(int data);

/* Function to print REVERSE level order traversal a tree*/
void reverseLevelOrder(struct node* root)
{
    int h = height(root);
```



```

    int h = height(root);
    int i;
    for (i=h; i>=1; i--) //THE ONLY LINE DIFFERENT FROM NORMAL LEVEL ORDER TRAVERSAL
        printGivenLevel(root, i);
}

/* Print nodes at a given level */
void printGivenLevel(struct node* root, int level)
{
    if (root == NULL)
        return;
    if (level == 1)
        printf("%d ", root->data);
    else if (level > 1)
    {
        printGivenLevel(root->left, level-1);
        printGivenLevel(root->right, level-1);
    }
}

```

```

/* Compute the "height" of a tree -- the number of
nodes along the longest path from the root node
down to the farthest leaf node.*/

```

```

int height(struct node* node)
{
    if (node==NULL)
        return 0;
    else
    {
        /* compute the height of each subtree */
        int lheight = height(node->left);
        int rheight = height(node->right);

        /* use the larger one */
        if (lheight > rheight)
            return(lheight+1);
        else return(rheight+1);
    }
}

```

```

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */

```

```

struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return(node);
}

```

```

/* Driver program to test above functions*/

```

```

int main()
{
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    printf("Level Order traversal of binary tree is \n");
}

```

```

        reverseLevelOrder(root);

    return 0;
}

```

Output:

```

Level Order traversal of binary tree is
4 5 2 3 1

```

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 $O(n) + O(n-1) + O(n-2) + \dots + O(1)$ 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.

```

// A C++ program to print REVERSE level order traversal using stack and queue
// This approach is adopted from following link
// http://tech-queries.blogspot.in/2008/12/level-order-tree-traversal-2.html
#include <iostream>
#include <stack>
#include <queue>
using namespace std;

/* A binary tree node has data, pointer to left and right children */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Given a binary tree, print its nodes in reverse level order */
void reverseLevelOrder(struct node* root)
{
    stack<node*> S;
    queue<node*> Q;
    Q.push(root);

    // Do something like normal level order traversal order. Following
    // differences with normal level order traversal
    // 1) Instead of printing a node, we push the node to stack
    // 2) Right subtree is visited before left subtree
    while (Q.empty() == false)
    {
        /* Dequeue node and make it root */
        root = Q.front();

```

```

        Q.pop();
        S.push(root);

        /* Enqueue right child */
        if (root->right)
            Q.push(root->right); // NOTE: RIGHT CHILD IS ENQUEUED BEFORE LEFT CHILD

        /* Enqueue left child */
        if (root->left)
            Q.push(root->left);
    }

    // Now pop all items from stack one by one and print them
    while (S.empty() == false)
    {
        root = S.top();
        cout << root->data << " ";
        S.pop();
    }
}

```

/* Helper function that allocates a new node with the given data and NULL left and right pointers. */

```
node* newNode(int data)
```

```
{
    node* temp = new node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;

```

```
    return (temp);
}
```

/* Driver program to test above functions*/

```
int main()
```

```
{
    struct node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);

    cout << "Level Order traversal of binary tree is \n";
    reverseLevelOrder(root);

    return 0;
}
```

Output:

```
Level Order traversal of binary tree is
4 5 6 7 2 3 1
```

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

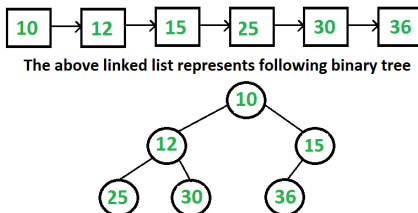
Please write comments if you find any bug in the above programs/algorithms or other ways to solve the same problem.

79. Construct Complete Binary Tree from its Linked List 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.



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.

Below is the code which implements the same in C++.

```
// C++ program to create a Complete Binary tree from its Linked List
// Representation
#include <iostream>
#include <string>
#include <queue>
using namespace std;
```

```

// Linked list node
struct ListNode
{
    int data;
    ListNode* next;
};

// Binary tree node structure
struct BinaryTreeNode
{
    int data;
    BinaryTreeNode *left, *right;
};

// Function to insert a node at the beginning of the Linked List
void push(struct ListNode** head_ref, int new_data)
{
    // allocate node and assign data
    struct ListNode* new_node = new ListNode;
    new_node->data = new_data;

    // link the old list off the new node
    new_node->next = (*head_ref);

    // move the head to point to the new node
    (*head_ref) = new_node;
}

// method to create a new binary tree node from the given data
BinaryTreeNode* newBinaryTreeNode(int data)
{
    BinaryTreeNode *temp = new BinaryTreeNode;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// converts a given linked list representing a complete binary tree in
// linked representation of binary tree.
void convertList2Binary(ListNode *head, BinaryTreeNode* &root)
{
    // queue to store the parent nodes
    queue<BinaryTreeNode *> q;

    // Base Case
    if (head == NULL)
    {
        root = NULL; // Note that root is passed by reference
        return;
    }

    // 1.) The first node is always the root node, and add it to the q
    root = newBinaryTreeNode(head->data);
    q.push(root);

    // advance the pointer to the next node
    head = head->next;

    // until the end of linked list is reached, do the following steps
    while (head)
    {
        // 2.) take the parent node from the q and remove it from q
    }
}

```

```

// 2.a) Take the parent node from the q and remove it from q
BinaryTreeNode* parent = q.front();
q.pop();

// 2.c) take next two nodes from the linked list. We will add
// them as children of the current parent node in step 2.b. Push
// into the queue so that they will be parents to the future nodes
BinaryTreeNode *leftChild = NULL, *rightChild = NULL;
leftChild = newBinaryTreeNode(head->data);
q.push(leftChild);
head = head->next;
if (head)
{
    rightChild = newBinaryTreeNode(head->data);
    q.push(rightChild);
    head = head->next;
}

// 2.b) assign the left and right children of parent
parent->left = leftChild;
parent->right = rightChild;
}
}

```

// Utility function to traverse the binary tree after conversion

```

void inorderTraversal(BinaryTreeNode* root)
{
    if (root)
    {
        inorderTraversal( root->left );
        cout << root->data << " ";
        inorderTraversal( root->right );
    }
}

```

// Driver program to test above functions

```

int main()
{
    // create a linked list shown in above diagram
    struct ListNode* head = NULL;
    push(&head, 36); /* Last node of Linked List */
    push(&head, 30);
    push(&head, 25);
    push(&head, 15);
    push(&head, 12);
    push(&head, 10); /* First node of Linked List */

    BinaryTreeNode *root;
    convertList2Binary(head, root);

    cout << "Inorder Traversal of the constructed Binary Tree is: \n";
    inorderTraversal(root);
    return 0;
}

```

Output:

```

Inorder Traversal of the constructed Binary Tree is:
25 12 30 10 36 15

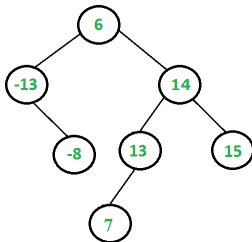
```

Time Complexity: Time complexity of the above solution is $O(n)$ where n is the number of nodes.

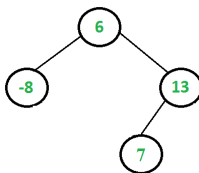
This article is compiled by **Ravi Chandra Enaganti**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

80. 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 than 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.

Following is C++ implementation of the above approach.

```
// A C++ program to remove BST keys outside the given range
#include<stdio.h>
#include <iostream>

using namespace std;

// A BST node has key, and left and right pointers
struct node
{
    int key;
    struct node *left;
    struct node *right;
};

// Removes all nodes having value outside the given range and returns
// 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 root
    // 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;
}

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

// A utility function to insert a given key to BST
node* insert(node* root, int key)
{
    if (root == NULL)
        return newNode(key);
    if (root->key > key)
```



```

        root->left = insert(root->left, key);
    else
        root->right = insert(root->right, key);
    return root;
}

// Utility function to traverse the binary tree after conversion
void inorderTraversal(node* root)
{
    if (root)
    {
        inorderTraversal( root->left );
        cout << root->key << " ";
        inorderTraversal( root->right );
    }
}

// Driver program to test above functions
int main()
{
    node* root = NULL;
    root = insert(root, 6);
    root = insert(root, -13);
    root = insert(root, 14);
    root = insert(root, -8);
    root = insert(root, 15);
    root = insert(root, 13);
    root = insert(root, 7);

    cout << "Inorder traversal of the given tree is: ";
    inorderTraversal(root);

    root = removeOutsideRange(root, -10, 13);

    cout << "\nInorder traversal of the modified tree is: ";
    inorderTraversal(root);

    return 0;
}

```

Output:

```

Inorder traversal of the given tree is: -13 -8 6 7 13 14 15
Inorder traversal of the modified tree is: -8 6 7 13

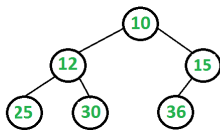
```

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

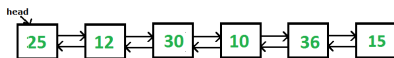
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

81. Convert a given Binary Tree to Doubly Linked List | Set 1

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.



The above tree should be in-place converted to following Doubly Linked List(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).

Below is the source code for above algorithm.

```
// A C++ program for in-place conversion of Binary Tree to DLL
#include <stdio.h>

/* A binary tree node has data, and left and right pointers */
struct node
{
    int data;
    node* left;
    node* right;
};

/* This is the core function to convert Tree to list. This function follows
steps 1 and 2 of the above algorithm */
node* bintree2listUtil(node* root)
```

```

{
    // Base case
    if (root == NULL)
        return root;

    // Convert the left subtree and link to root
    if (root->left != NULL)
    {
        // Convert the left subtree
        node* left = bintree2listUtil(root->left);

        // Find inorder predecessor. After this loop, left
        // will point to the inorder predecessor
        for (; left->right!=NULL; left=left->right);

        // Make root as next of the predecessor
        left->right = root;

        // Make predecessor as previous of root
        root->left = left;
    }

    // Convert the right subtree and link to root
    if (root->right!=NULL)
    {
        // Convert the right subtree
        node* right = bintree2listUtil(root->right);

        // Find inorder successor. After this loop, right
        // will point to the inorder successor
        for (; right->left!=NULL; right = right->left);

        // Make root as previous of successor
        right->left = root;

        // Make successor as next of root
        root->right = right;
    }

    return root;
}

// The main function that first calls bintree2listUtil(), then follows
// of the above algorithm
node* bintree2list(node *root)
{
    // Base case
    if (root == NULL)
        return root;

    // Convert to DLL using bintree2listUtil()
    root = bintree2listUtil(root);

    // bintree2listUtil() returns root node of the converted
    // DLL. We need pointer to the leftmost node which is
    // head of the constructed DLL, so move to the leftmost node
    while (root->left != NULL)
        root = root->left;

    return (root);
}

```

```

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
node* newNode(int data)
{
    node* new_node = new node;
    new_node->data = data;
    new_node->left = new_node->right = NULL;
    return (new_node);
}

/* Function to print nodes in a given doubly linked list */
void printList(node *node)
{
    while (node!=NULL)
    {
        printf("%d ", node->data);
        node = node->right;
    }
}

/* Driver program to test above functions*/
int main()
{
    // Let us create the tree shown in above diagram
    node *root      = newNode(10);
    root->left       = newNode(12);
    root->right      = newNode(15);
    root->left->left  = newNode(25);
    root->left->right = newNode(30);
    root->right->left = newNode(36);

    // Convert to DLL
    node *head = bintree2list(root);

    // Print the converted list
    printList(head);

    return 0;
}

```

Output:

```
25 12 30 10 36 15
```

This article is compiled by **Ashish Mangla** and reviewed by GeeksforGeeks team. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

You may also like to see [Convert a given Binary Tree to Doubly Linked List | Set 2](#) for another simple and efficient solution.

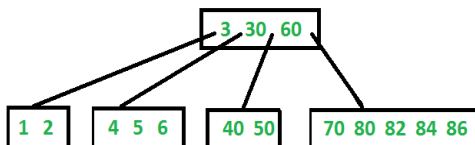
82. B-Tree | Set 1 (Introduction)

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red Black Trees), it is assumed that everything is in main memory. To understand use of B-Trees, we must think of huge amount of data that cannot fit in main memory. When the number of keys is high, the data is read from disk in the form of blocks. Disk access time is very high compared to main memory access time. The main idea of using B-Trees is to reduce the number of disk accesses. Most of the tree operations (search, insert, delete, max, min, ..etc) require $O(h)$ disk accesses where h is height of the tree. B-tree is a fat tree. Height of B-Trees is kept low by putting maximum possible keys in a B-Tree node. Generally, a B-Tree node size is kept equal to the disk block size. Since h is low for B-Tree, total disk accesses for most of the operations are reduced significantly compared to balanced Binary Search Trees like AVL Tree, Red Black Tree, ..etc.

Properties of B-Tree

- 1) All leaves are at same level.
- 2) A B-Tree is defined by the term *minimum degree* 't'. The value of t depends upon disk block size.
- 3) Every node except root must contain at least $t-1$ keys. Root may contain minimum 1 key.
- 4) All nodes (including root) may contain at most $2t - 1$ keys.
- 5) Number of children of a node is equal to the number of keys in it plus 1.
- 6) All keys of a node are sorted in increasing order. The child between two keys k_1 and k_2 contains all keys in range from k_1 and k_2 .
- 7) B-Tree grows and shrinks from root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
- 8) Like other balanced Binary Search Trees, time complexity to search, insert and delete is $O(\text{Log}n)$.

Following is an example B-Tree of minimum degree 3. Note that in practical B-Trees, the value of minimum degree is much more than 3.



Search

Search is similar to search in Binary Search Tree. Let the key to be searched be k . We start from root and recursively traverse down. For every visited non-leaf node, if the node has key, we simply return the node. Otherwise we recur down to the appropriate child (The child which is just before the first greater key) of the node. If we reach a leaf node and don't find k in the leaf node, we return NULL.

Traverse

Traversal is also similar to Inorder traversal of Binary Tree. We start from the leftmost

child, recursively print the leftmost child, then repeat the same process for remaining children and keys. In the end, recursively print the rightmost child.

```
// C++ implementation of search() and traverse() methods
#include<iostream>
using namespace std;

// A BTree node
class BTreeNode
{
    int *keys; // An array of keys
    int t; // Minimum degree (defines the range for number of keys)
    BTreeNode **C; // An array of child pointers
    int n; // Current number of keys
    bool leaf; // Is true when node is leaf. Otherwise false
public:
    BTreeNode(int _t, bool _leaf); // Constructor

    // A function to traverse all nodes in a subtree rooted with this node.
    void traverse();

    // A function to search a key in subtree rooted with this node.
    BTreeNode *search(int k); // returns NULL if k is not present.

// Make BTree friend of this so that we can access private members of
// class in BTree functions
friend class BTree;
};

// A BTree
class BTree
{
    BTreeNode *root; // Pointer to root node
    int t; // Minimum degree
public:
    // Constructor (Initializes tree as empty)
    BTree(int _t)
    { root = NULL; t = _t; }

    // function to traverse the tree
    void traverse()
    { if (root != NULL) root->traverse(); }

    // function to search a key in this tree
    BTreeNode* search(int k)
    { return (root == NULL)? NULL : root->search(k); }
};

// Constructor for BTreeNode class
BTreeNode::BTreeNode(int _t, bool _leaf)
{
    // Copy the given minimum degree and leaf property
    t = _t;
    leaf = _leaf;

    // Allocate memory for maximum number of possible keys
    // and child pointers
    keys = new int[2*t-1];
    C = new BTreeNode *[2*t];

    // Initialize the number of keys as 0
    n = 0;
```

```

}

// Function to traverse all nodes in a subtree rooted with this node
void BTreeNode::traverse()
{
    // There are n keys and n+1 children, travers through n keys
    // and first n children
    int i;
    for (i = 0; i < n; i++)
    {
        // If this is not leaf, then before printing key[i],
        // traverse the subtree rooted with child C[i].
        if (leaf == false)
            C[i]->traverse();
        cout << " " << keys[i];
    }

    // Print the subtree rooted with last child
    if (leaf == false)
        C[i]->traverse();
}

// Function to search key k in subtree rooted with this node
BTreeNode *BTreeNode::search(int k)
{
    // Find the first key greater than or equal to k
    int i = 0;
    while (i < n && k > keys[i])
        i++;

    // If the found key is equal to k, return this node
    if (keys[i] == k)
        return this;

    // If key is not found here and this is a leaf node
    if (leaf == true)
        return NULL;

    // Go to the appropriate child
    return C[i]->search(k);
}

```

The above code doesn't contain driver program. We will be covering the complete program in our next post on B-Tree Insertion.

There are two conventions to define a B-Tree, one is to define by minimum degree (followed in [Cormen book](#)), second is define by order. We have followed the minimum degree convention and will be following same in coming posts on B-Tree. The variable names used in the above program are also kept same as Cormen book for better readability.

Insertion and Deletion

B-Tree Insertion

B-Tree Deletion

References:

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

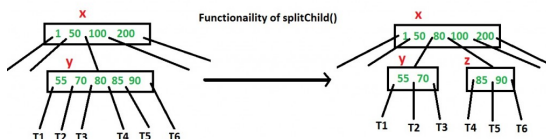
83. B-Tree | Set 2 (Insert)

In the [previous post](#), we introduced B-Tree. We also discussed `search()` and `traverse()` functions.

In this post, `insert()` operation is discussed. A new key is always inserted at leaf node. Let the key to be inserted be k . Like BST, we start from root and traverse down till we reach a leaf node. Once we reach a leaf node, we insert the key in that leaf node. Unlike BSTs, we have a predefined range on number of keys that a node can contain. So before inserting a key to node, we make sure that the node has extra space.

How to make sure that a node has space available for key before the key is inserted?

We use an operation called `splitChild()` that is used to split a child of a node. See the following diagram to understand split. In the following diagram, child y of x is being split into two nodes y and z . Note that the `splitChild` operation moves a key up and this is the reason B-Trees grow up unlike BSTs which grow down.



As discussed above, to insert a new key, we go down from root to leaf. Before traversing down to a node, we first check if the node is full. If the node is full, we split it to create space. Following is complete algorithm.

Insertion

- 1) Initialize x as root.
- 2) While x is not leaf, do following
 - ..a) Find the child of x that is going to be traversed next. Let the child be y .
 - ..b) If y is not full, change x to point to y .
 - ..c) If y is full, split it and change x to point to one of the two parts of y . If k is smaller than mid key in y , then set x as first part of y . Else second part of y . When we split y , we move a key from y to its parent x .
- 3) The loop in step 2 stops when x is leaf. x must have space for 1 extra key as we have been splitting all nodes in advance. So simply insert k to x .


Note that the algorithm follows the Cormen book. It is actually a proactive insertion

algorithm where before going down to a node, we split it if it is full. The advantage of splitting before is, we never traverse a node twice. If we don't split a node before going down to it and split it only if new key is inserted (reactive), we may end up traversing all nodes again from leaf to root. This happens in cases when all nodes on the path from root to leaf are full. So when we come to the leaf node, we split it and move a key up. Moving a key up will cause a split in parent node (because parent was already full). This cascading effect never happens in this proactive insertion algorithm. There is a disadvantage of this proactive insertion though, we may do unnecessary splits.

Let us understand the algorithm with an example tree of minimum degree 't' as 3 and a sequence of integers 10, 20, 30, 40, 50, 60, 70, 80 and 90 in an initially empty B-Tree.


Initially root is NULL. Let us first insert 10.

Insert 10



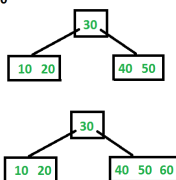
Let us now insert 20, 30, 40 and 50. They all will be inserted in root because maximum number of keys a node can accommodate is $2*t - 1$ which is 5.

Insert 20, 30, 40 and 50



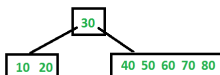
Let us now insert 60. Since root node is full, it will first split into two, then 60 will be inserted into the appropriate child.

Insert 60




Let us now insert 70 and 80. These new keys will be inserted into the appropriate leaf without any split.

Insert 70 and 80



Let us now insert 90. This insertion will cause a split. The middle key will go up to the parent.

Insert 90



See [this](#) for more examples.

Following is C++ implementation of the above proactive algorithm.

```
// C++ program for B-Tree insertion
#include<iostream>
```

```

using namespace std;

// A BTree node
class BTreeNode
{
    int *keys; // An array of keys
    int t; // Minimum degree (defines the range for number of keys)
    BTreeNode **C; // An array of child pointers
    int n; // Current number of keys
    bool leaf; // Is true when node is leaf. Otherwise false
public:
    BTreeNode(int _t, bool _leaf); // Constructor

    // A utility function to insert a new key in the subtree rooted with
    // this node. The assumption is, the node must be non-full when the
    // function is called
    void insertNonFull(int k);

    // A utility function to split the child y of this node. i is index of
    // child array C[]. The Child y must be full when this function is
    // called
    void splitChild(int i, BTreeNode *y);

    // A function to traverse all nodes in a subtree rooted with this node
    void traverse();

    // A function to search a key in subtree rooted with this node.
    BTreeNode *search(int k); // returns NULL if k is not present.

// Make BTree friend of this so that we can access private members of
// class in BTree functions
friend class BTree;
};

// A BTree
class BTree
{
    BTreeNode *root; // Pointer to root node
    int t; // Minimum degree
public:
    // Constructor (Initializes tree as empty)
    BTree(int _t)
    { root = NULL; t = _t; }

    // function to traverse the tree
    void traverse()
    { if (root != NULL) root->traverse(); }

    // function to search a key in this tree
    BTreeNode* search(int k)
    { return (root == NULL)? NULL : root->search(k); }

    // The main function that inserts a new key in this B-Tree
    void insert(int k);
};

// Constructor for BTreeNode class
BTreeNode::BTreeNode(int t1, bool leaf1)
{
    // Copy the given minimum degree and leaf property
    t = t1;
    leaf = leaf1;

    // Allocate memory for maximum number of possible keys

```

```

// Allocate memory for maximum number of possible keys
// and child pointers
keys = new int[2*t-1];
C = new BTreeNode *[2*t];

// Initialize the number of keys as 0
n = 0;
}

// Function to traverse all nodes in a subtree rooted with this node
void BTreeNode::traverse()
{
    // There are n keys and n+1 children, travers through n keys
    // and first n children
    int i;
    for (i = 0; i < n; i++)
    {
        // If this is not leaf, then before printing key[i],
        // traverse the subtree rooted with child C[i].
        if (leaf == false)
            C[i]->traverse();
        cout << " " << keys[i];
    }

    // Print the subtree rooted with last child
    if (leaf == false)
        C[i]->traverse();
}

// Function to search key k in subtree rooted with this node
BTreeNode *BTreeNode::search(int k)
{
    // Find the first key greater than or equal to k
    int i = 0;
    while (i < n && k > keys[i])
        i++;

    // If the found key is equal to k, return this node
    if (keys[i] == k)
        return this;

    // If key is not found here and this is a leaf node
    if (leaf == true)
        return NULL;

    // Go to the appropriate child
    return C[i]->search(k);
}

// The main function that inserts a new key in this B-Tree
void BTree::insert(int k)
{
    // If tree is empty
    if (root == NULL)
    {
        // Allocate memory for root
        root = new BTreeNode(t, true);
        root->keys[0] = k; // Insert key
        root->n = 1; // Update number of keys in root
    }
    else // If tree is not empty
    {
        // If root is full, then tree grows in height

```

```

if (root->n == 2*t-1)
{
    // Allocate memory for new root
    BTreeNode *s = new BTreeNode(t, false);

    // Make old root as child of new root
    s->C[0] = root;

    // Split the old root and move 1 key to the new root
    s->splitChild(0, root);

    // New root has two children now. Decide which of the
    // two children is going to have new key
    int i = 0;
    if (s->keys[0] < k)
        i++;
    s->C[i]->insertNonFull(k);

    // Change root
    root = s;
}
else // If root is not full, call insertNonFull for root
    root->insertNonFull(k);
}
}

// A utility function to insert a new key in this node
// The assumption is, the node must be non-full when this
// function is called
void BTreeNode::insertNonFull(int k)
{
    // Initialize index as index of rightmost element
    int i = n-1;

    // If this is a leaf node
    if (leaf == true)
    {
        // The following loop does two things
        // a) Finds the location of new key to be inserted
        // b) Moves all greater keys to one place ahead
        while (i >= 0 && keys[i] > k)
        {
            keys[i+1] = keys[i];
            i--;
        }

        // Insert the new key at found location
        keys[i+1] = k;
        n = n+1;
    }
    else // If this node is not leaf
    {
        // Find the child which is going to have the new key
        while (i >= 0 && keys[i] > k)
            i--;

        // See if the found child is full
        if (C[i+1]->n == 2*t-1)
        {
            // If the child is full, then split it
            splitChild(i+1, C[i+1]);

```

```

        // After split, the middle key of C[i] goes up and
        // C[i] is splitted into two. See which of the two
        // is going to have the new key
        if (keys[i+1] < k)
            i++;
    }
    C[i+1]->insertNonFull(k);
}
}

```

```

// A utility function to split the child y of this node
// Note that y must be full when this function is called
void BTreeNode::splitChild(int i, BTreeNode *y)
{
    // Create a new node which is going to store (t-1) keys
    // of y
    BTreeNode *z = new BTreeNode(y->t, y->leaf);
    z->n = t - 1;

    // Copy the last (t-1) keys of y to z
    for (int j = 0; j < t-1; j++)
        z->keys[j] = y->keys[j+t];

    // Copy the last t children of y to z
    if (y->leaf == false)
    {
        for (int j = 0; j < t; j++)
            z->C[j] = y->C[j+t];
    }

    // Reduce the number of keys in y
    y->n = t - 1;

    // Since this node is going to have a new child,
    // create space of new child
    for (int j = n; j >= i+1; j--)
        C[j+1] = C[j];

    // Link the new child to this node
    C[i+1] = z;

    // A key of y will move to this node. Find location of
    // new key and move all greater keys one space ahead
    for (int j = n-1; j >= i; j--)
        keys[j+1] = keys[j];

    // Copy the middle key of y to this node
    keys[i] = y->keys[t-1];

    // Increment count of keys in this node
    n = n + 1;
}

```

```

// Driver program to test above functions
int main()
{
    BTree t(3); // A B-Tree with minium degree 3
    t.insert(10);
    t.insert(20);
    t.insert(5);
    t.insert(6);
    t.insert(12);
    t.insert(30);
}

```

```

t.insert(5);
t.insert(7);
t.insert(17);

cout << "Traversal of the constucted tree is ";
t.traverse();

int k = 6;
(t.search(k) != NULL)? cout << "\nPresent" : cout << "\nNot Present";

k = 15;
(t.search(k) != NULL)? cout << "\nPresent" : cout << "\nNot Present";

return 0;
}

```

Output:

```

Traversal of the constucted tree is  5 6 7 10 12 17 20 30
Present
Not Present

```

References:

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest
<http://www.cs.utexas.edu/users/djimenez/utsa/cs3343/lecture17.html>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

84. Longest prefix matching – A Trie based solution in Java

Given a dictionary of words and an input string, find the longest prefix of the string which is also a word in dictionary.

Examples:

Let the dictionary contains the following words:
{are, area, base, cat, cater, children, basement}

Below are some input/output examples:

Input String	Output
caterer	cater
basemexy	base
child	< Empty >

Solution

We build a Trie of all dictionary words. Once the Trie is built, traverse through it using characters of input string. If prefix matches a dictionary word, store current length and look for a longer match. Finally, return the longest match.

Following is Java implementation of the above solution based.

```
import java.util.HashMap;

// Trie Node, which stores a character and the children in a HashMap
class TrieNode {
    public TrieNode(char ch) {
        value = ch;
        children = new HashMap<>();
        bIsEnd = false;
    }
    public HashMap<Character,TrieNode> getChildren() { return children; }
    public char getValue() { return value; }
    public void setIsEnd(boolean val) { bIsEnd = val; }
    public boolean isEnd() { return bIsEnd; }

    private char value;
    private HashMap<Character,TrieNode> children;
    private boolean bIsEnd;
}

// Implements the actual Trie
class Trie {
    // Constructor
    public Trie() { root = new TrieNode((char)0); }

    // Method to insert a new word to Trie
    public void insert(String word) {

        // Find length of the given word
        int length = word.length();
        TrieNode crawl = root;

        // Traverse through all characters of given word
        for( int level = 0; level < length; level++)
        {
            HashMap<Character,TrieNode> child = crawl.getChildren();
            char ch = word.charAt(level);

            // If there is already a child for current character of given word
            if( child.containsKey(ch))
```

```

        crawl = child.get(ch);
    else    // Else create a child
    {
        TrieNode temp = new TrieNode(ch);
        child.put( ch, temp );
        crawl = temp;
    }
}

// Set bIsEnd true for last character
crawl.setIsEnd(true);
}

// The main method that finds out the longest string 'input'
public String getMatchingPrefix(String input) {
    String result = ""; // Initialize resultant string
    int length = input.length(); // Find length of the input string

    // Initialize reference to traverse through Trie
    TrieNode crawl = root;

    // Iterate through all characters of input string 'str' and traverse
    // down the Trie
    int level, prevMatch = 0;
    for( level = 0 ; level < length; level++ )
    {
        // Find current character of str
        char ch = input.charAt(level);

        // HashMap of current Trie node to traverse down
        HashMap<Character,TrieNode> child = crawl.getChildren();

        // See if there is a Trie edge for the current character
        if( child.containsKey(ch) )
        {
            result += ch;           //Update result
            crawl = child.get(ch); //Update crawl to move down in Trie

            // If this is end of a word, then update prevMatch
            if( crawl.isEnd() )
                prevMatch = level + 1;
        }
        else break;
    }
}

```



```

        // If the last processed character did not match end of a word,
        // return the previously matching prefix
        if( !crawl.isEnd() )
            return result.substring(0, prevMatch);

        else return result;
    }

    private TrieNode root;
}

// Testing class
public class Test {
    public static void main(String[] args) {
        Trie dict = new Trie();
        dict.insert("are");
        dict.insert("area");
        dict.insert("base");
        dict.insert("cat");
        dict.insert("cater");
        dict.insert("basement");

        String input = "caterer";
        System.out.print(input + ": ");
        System.out.println(dict.getMatchingPrefix(input));

        input = "basement";
        System.out.print(input + ": ");
        System.out.println(dict.getMatchingPrefix(input));

        input = "are";
        System.out.print(input + ": ");
        System.out.println(dict.getMatchingPrefix(input));

        input = "arex";
        System.out.print(input + ": ");
        System.out.println(dict.getMatchingPrefix(input));

        input = "basemexz";
        System.out.print(input + ": ");
        System.out.println(dict.getMatchingPrefix(input));

        input = "xyz";
        System.out.print(input + ": ");
    }
}

```

```

        System.out.println(dict.getMatchingPrefix(input));
    }
}

```

Output:

```

caterer:    cater
basement:   basement
are:        are
arex:       are
basemexz:   base
xyz:

```

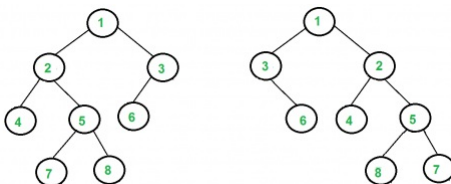
Time Complexity: Time complexity of finding the longest prefix is $O(n)$ where n is length of the input string. Refer [this](#) for time complexity of building the Trie.

This article is compiled by **Ravi Chandra Enaganti**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

85. 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.

```
// A C++ program to check if two given trees are isomorphic
#include <iostream>
using namespace std;

/* A binary tree node has data, pointer to left and right children */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* 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))
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
node* newNode(int data)
{
    node* temp = new node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;

    return (temp);
}

/* Driver program to test above functions*/
int main()
{
    // Let us create trees shown in above diagram
    struct node *n1 = newNode(1);
    n1->left = newNode(2);
    n1->right = newNode(3);
    n1->left->left = newNode(4);
    n1->left->right = newNode(5);
    n1->right->left = newNode(6);
```

```

n1->left->right->left = newNode(7);
n1->left->right->right = newNode(8);

struct node *n2 = newNode(1);
n2->left          = newNode(3);
n2->right         = newNode(2);
n2->right->left    = newNode(4);
n2->right->right   = newNode(5);
n2->left->right    = newNode(6);
n2->right->right->left = newNode(8);
n2->right->right->right = newNode(7);

if (isIsomorphic(n1, n2) == true)
    cout << "Yes";
else
    cout << "No";

return 0;
}

```

Output:

Yes

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.

This article is contributed by **Ciphe**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above. If you like GeeksforGeeks and would like to contribute, you can also write an article and mail your article to contribute@geeksforgeeks.org. See your article appearing on the GeeksforGeeks main page and help other Geeks.

86. Find all possible interpretations of an array of digits

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}

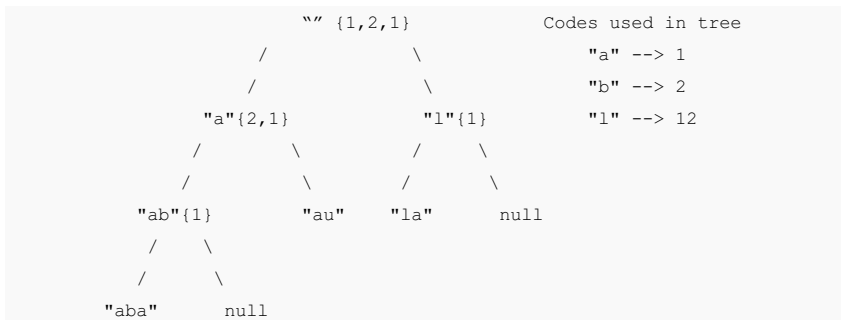
On first look it looks like a problem of permutation/combination. But on closer look you will figure out that this is an interesting tree problem.

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

1. Process 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.

Following is Java implementation of above algorithm.

```
// A Java program to print all interpretations of an integer array
import java.util.Arrays;
```

```
// A Binary Tree node
class Node {

    String dataString;
    Node left;
    Node right;

    Node(String dataString) {
        this.dataString = dataString;
        //Be default left and right child are null.
    }

    public String getDataString() {
        return dataString;
    }
}

public class arrayToAllInterpretations {

    // 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
}

// For simplicity I am taking it as string array. Char Array will save space
private static final String[] alphabet = {"", "a", "b", "c", "d", "e",
    "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r",
    "s", "t", "u", "v", "w", "x", "y", "z"};

// Driver method to test above methods
public static void main(String args[]) {

```

```

        // aacd(1,1,3,4) amd(1,13,4) kcd(11,3,4)
        // Note : 1,1,34 is not valid as we don't have values corresponding
        // to 34 in alphabet
        int[] arr = {1, 1, 3, 4};
        printAllInterpretations(arr);

        // aaa(1,1,1) ak(1,11) ka(11,1)
        int[] arr2 = {1, 1, 1};
        printAllInterpretations(arr2);

        // bf(2,6) z(26)
        int[] arr3 = {2, 6};
        printAllInterpretations(arr3);

        // ab(1,2), 1(12)
        int[] arr4 = {1, 2};
        printAllInterpretations(arr4);

        // a(1,0} j(10)
        int[] arr5 = {1, 0};
        printAllInterpretations(arr5);

        // "" empty string output as array is empty
        int[] arr6 = {};
        printAllInterpretations(arr6);

        // abba abu ava lba lu
        int[] arr7 = {1, 2, 2, 1};
        printAllInterpretations(arr7);
    }
}

```

Output:

```

aacd amd kcd
aaa ak ka
bf z
ab 1
a j

abba abu ava lba lu

```

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?

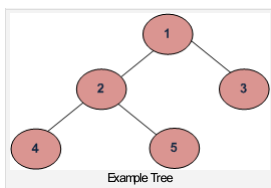
This article is compiled by [Varun Jain](#). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

87. 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.

In this post, the first convention is followed. For example, height of the below tree is 3.



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.

Following is detailed algorithm to find level order traversal using queue.

```
Create a queue.
Push root into the queue.
height = 0
Loop
    nodeCount = size of queue

    // If number of nodes at this level is 0, return height
    if nodeCount is 0
        return Height;
    else
        increase Height

    // Remove nodes of this level and add nodes of
    // next level
```

```

while (nodeCount > 0)
    pop node from front
    push its children to queue
    decrease nodeCount

    // At this point, queue has nodes of next level

```

Following is C++ implementation of above algorithm.

```

/* Program to find height of the tree by Iterative Method */
#include <iostream>
#include <queue>
using namespace std;

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

// Iterative method to find height of Binary Tree
int treeHeight(node *root)
{
    // Base Case
    if (root == NULL)
        return 0;

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

    // Enqueue Root and initialize height
    q.push(root);
    int height = 0;

    while (1)
    {
        // nodeCount (queue size) indicates number of nodes
        // at current level.
        int nodeCount = q.size();
        if (nodeCount == 0)
            return height;

        height++;

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

```

```
// Utility function to create a new tree node
node* newNode(int data)
{
    node *temp = new node;
    temp->data = data;
    temp->left = NULL;
    temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree shown in above diagram
    node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);

    cout << "Height of tree is " << treeHeight(root);
    return 0;
}
```

Output:

```
Height of tree is 3
```

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

This article is contributed by **Rahul Kumar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

88. Custom Tree Problem

You are given a set of links, e.g.

```
a ---> b
b ---> c
b ---> d
a ---> e
```

Print the tree that would form when each pair of these links that has the same character as start and end point is joined together. You have to maintain fidelity w.r.t. the height of nodes, i.e. nodes at height n from root should be printed at same row or column. For set of links given above, tree printed should be –

```
-->a
  |-->b
```

```
|   |-->c
|   |-->d
|-->e
```

Note that these links need not form a single tree; they could form, ahem, a forest.
Consider the following links

```
a ----> b
a ----> g
b ----> c
c ----> d
d ----> e
c ----> f
z ----> y
y ----> x
x ----> w
```

The output would be following forest.

```
-->a
|-->b
|   |-->c
|   |   |-->d
|   |   |   |-->e
|   |   |   |-->f
|   |-->g

-->z
|-->y
|   |-->x
|   |   |-->w
```

You can assume that given links can form a tree or forest of trees only, and there are no duplicates among links.

Solution: The idea is to maintain two arrays, one array for tree nodes and other for trees themselves (we call this array forest). An element of the node array contains the `TreeNode` object that corresponds to respective character. An element of the forest array contains `Tree` object that corresponds to respective root of tree.

It should be obvious that the crucial part is creating the forest here, once it is created, printing it out in required format is straightforward. To create the forest, following procedure is used –

Do following for each input link,

1. If start of link is not present in node array
Create `TreeNode` objects for start character

- Add entries of start in both arrays.
- 2. If end of link is not present in node array
 - Create TreeNode objects for start character
 - Add entry of end in node array.
- 3. If end of link is present in node array.
 - If end of link is present in forest array, then remove it from there.
- 4. Add an edge (in tree) between start and end nodes of link.

It should be clear that this procedure runs in linear time in number of nodes as well as of links – it makes only one pass over the links. It also requires linear space in terms of alphabet size.

Following is Java implementation of above algorithm. In the following implementation characters are assumed to be only lower case characters from 'a' to 'z'.

```
// Java program to create a custom tree from a given set of links.

// The main class that represents tree and has main method
public class Tree {

    private TreeNode root;

    /* Returns an array of trees from links input. Links are assumed to
    be Strings of the form "<s> <e>" where <s> and <e> are starting
    and ending points for the link. The returned array is of size 26
    and has non-null values at indexes corresponding to roots of trees
    in output */
    public Tree[] buildFromLinks(String [] links) {

        // Create two arrays for nodes and forest
        TreeNode[] nodes = new TreeNode[26];
        Tree[] forest = new Tree[26];

        // Process each link
        for (String link : links) {

            // Find the two ends of current link
            String[] ends = link.split(" ");
            int start = (int) (ends[0].charAt(0) - 'a'); // Start node
            int end = (int) (ends[1].charAt(0) - 'a'); // End node

            // If start of link not seen before, add it to both arrays
            if (nodes[start] == null)
            {
                nodes[start] = new TreeNode((char) (start + 'a'));
```

```

        // Note that it may be removed later when this character is
        // last character of a link. For example, let we first see
        // a--->b, then c--->a. We first add 'a' to array of trees
        // and when we see link c--->a, we remove it from trees array.
        forest[start] = new Tree(nodes[start]);
    }

    // If end of link is not seen before, add it to the nodes array
    if (nodes[end] == null)
        nodes[end] = new TreeNode((char) (end + 'a'));

    // If end of link is seen before, remove it from forest if
    // it exists there.
    else forest[end] = null;

    // Establish Parent-Child Relationship between Start and End
    nodes[start].addChild(nodes[end], end);
}

return forest;
}

// Constructor
public Tree(TreeNode root) { this.root = root; }

public static void printForest(String[] links)
{
    Tree t = new Tree(new TreeNode('\0'));
    for (Tree t1 : t.buildFromLinks(links)) {
        if (t1 != null)
        {
            t1.root.printTreeIdented("");
            System.out.println("");
        }
    }
}

// Driver method to test
public static void main(String[] args) {
    String [] links1 = {"a b", "b c", "b d", "a e"};
    System.out.println("----- Forest 1 -----");
    printForest(links1);

    String [] links2 = {"a b", "a g", "b c", "c d", "d e", "c f",
                        "z y", "y x", "x w"};

```

```

        System.out.println("----- Forest 2 -----");
        printForest(links2);
    }
}

// Class to represent a tree node
class TreeNode {
    TreeNode []children;
    char c;

    // Adds a child 'n' to this node
    public void addChild(TreeNode n, int index) { this.children[index] = n;}

    // Constructor
    public TreeNode(char c) { this.c = c; this.children = new TreeNode[26];}

    // Recursive method to print indented tree rooted with this node.
    public void printTreeIndented(String indent) {
        System.out.println(indent + "-->" + c);
        for (TreeNode child : children) {
            if (child != null)
                child.printTreeIndented(indent + "    |");
        }
    }
}

```

Output:

```

----- Forest 1 -----
-->a
    |-->b
        |-->c
        |-->d
    |-->e

----- Forest 2 -----
-->a
    |-->b
        |-->c
            |-->d
                |-->e
                |-->f
            |-->g
-->z

```

```
|-->y
|   |-->x
|   |   |-->w
```

Exercise:

In the above implementation, endpoints of input links are assumed to be from set of only 26 characters. Extend the implementation where endpoints are strings of any length.

This article is contributed by **Ciphe**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

89. 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
```

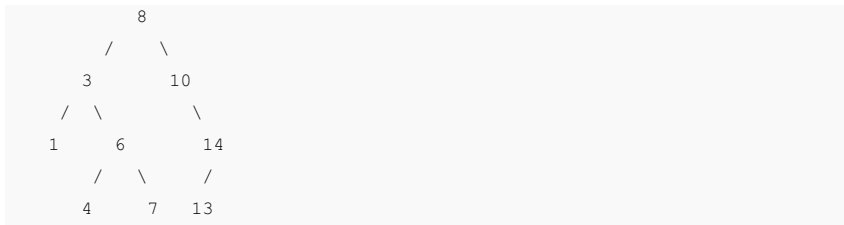
So the output is "True"

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"

**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 same 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 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.

```

// A C program to check for Identical BSTs without building the trees
#include<stdio.h>
#include<limits.h>
#include<stdbool.h>

/* The main function that checks if two arrays a[] and b[] of size n can
   same BST. The two values 'min' and 'max' decide whether the call is for
   left subtree or right subtree of a parent element. The indexes i1 and i2
   the indexes in (a[] and b[]) after which we search the left or right
   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
       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 (decided 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
       subtrees of it. Note that we can also pass a[k] in place of a[j] if
       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
}

// A wrapper over isSameBSTUtil()
bool isSameBST(int a[], int b[], int n)
{
    return isSameBSTUtil(a, b, n, 0, 0, INT_MIN, INT_MAX);
}

// Driver program to test above functions
int main()
{
    int a[] = {8, 3, 6, 1, 4, 7, 10, 14, 13};
    int b[] = {8, 10, 14, 3, 6, 4, 1, 7, 13};
    int n=sizeof(a)/sizeof(a[0]);
    printf("%s\n", isSameBST(a, b, n)?
           "BSTs are same":"BSTs not same");
    return 0;
}

```

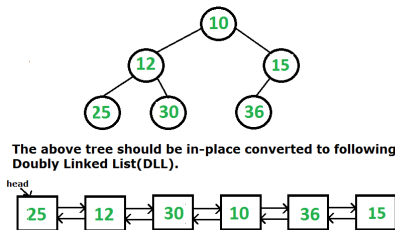
Output:

BSTs are same

This article is compiled by **Arnit Jain**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

90. Convert a given Binary Tree to Doubly Linked List | Set 2

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.



A solution to this problem is discussed in [this post](#).

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.

```
// A simple inorder traversal based program to convert a Binary Tree to  
#include<stdio.h>  
#include<stdlib.h>
```

```
// A tree node
```

```

struct node
{
    int data;
    struct node *left, *right;
};

// A utility function to create a new tree node
struct node *newNode(int data)
{
    struct node *node = (struct node *)malloc(sizeof(struct node));
    node->data = data;
    node->left = node->right = NULL;
    return(node);
}

// Standard Inorder traversal of tree
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%t%d", root->data);
        inorder(root->right);
    }
}

// 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 pointer.
    // 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 *BToDLL(struct node *root)
{
    // Set the previous pointer
    fixPrevPtr(root);

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

// Traverses the DLL from left to right
void printList(struct node *root)
{
    while (root != NULL)
    {
        printf("%t%d", root->data);
        root = root->right;
    }
}

// Driver program to test above functions
int main(void)
{
    // Let us create the tree shown in above diagram
    struct node *root = newNode(10);
    root->left = newNode(12);
    root->right = newNode(15);
    root->left->left = newNode(25);
    root->left->right = newNode(30);
    root->right->left = newNode(36);

    printf("\n\t\tInorder Tree Traversal\n\n");
    inorder(root);

    struct node *head = BToDLL(root);

    printf("\n\n\t\tDLL Traversal\n\n");
    printList(head);
    return 0;
}

```

Output:

```

                Inorder Tree Traversal

        25      12      30      10      36      15

                DLL Traversal

        25      12      30      10      36      15

```

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.

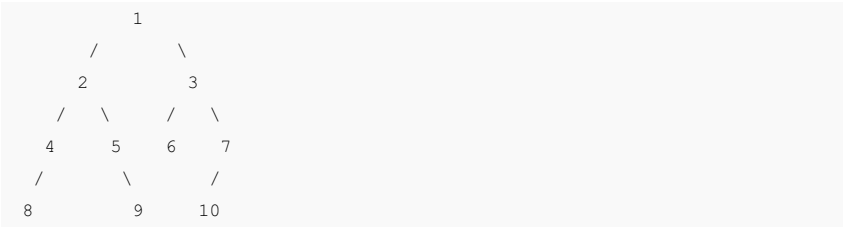
This article is contributed by **Bala**. Please write comments if you find anything incorrect,

or you want to share more information about the topic discussed above

91. 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.

For example, consider the following Binary Tree



Following are different input keys and their ancestors in the above tree

Input Key	List of Ancestors
1	
2	1
3	1
4	2 1
5	2 1
6	3 1
7	3 1
8	4 2 1
9	5 2 1
10	7 3 1

Recursive solution for this problem is discussed [here](#).

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.

Following is C implementation of the above approach.

```

// C program to print all ancestors of a given key
#include <stdio.h>
#include <stdlib.h>

// Maximum stack size
#define MAX_SIZE 100

// Structure for a tree node
struct Node
{
    int data;
    struct Node *left, *right;
};

// Structure for Stack
struct Stack
{
    int size;
    int top;
    struct Node* *array;
};

// A utility function to create a new tree node
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// A utility function to create a stack of given size
struct Stack* createStack(int size)
{
    struct Stack* stack = (struct Stack*) malloc(sizeof(struct Stack))
    stack->size = size;
    stack->top = -1;
    stack->array = (struct Node**) malloc(stack->size * sizeof(struct Node*))
    return stack;
}

// BASIC OPERATIONS OF STACK
int isFull(struct Stack* stack)
{
    return ((stack->top + 1) == stack->size);
}
int isEmpty(struct Stack* stack)
{
    return stack->top == -1;
}
void push(struct Stack* stack, struct Node* node)
{
    if (isFull(stack))
        return;
    stack->array[++stack->top] = node;
}
struct Node* pop(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top--];
}

```

```

struct Node* peek(struct Stack* stack)
{
    if (isEmpty(stack))
        return NULL;
    return stack->array[stack->top];
}

// 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 in
        // 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 it
            // as well. Left child of the top must have processed before
            // Consider the following tree for example and key = 3. If
            // 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 green there in tree
    while (!isEmpty(stack))
        printf("%d ", pop(stack)->data);
}

```



```

    printf("%d\n", pop(stack) / data);
}

// Driver program to test above functions
int main()
{
    // Let us construct a binary tree
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->left->left->left = newNode(8);
    root->left->right->right = newNode(9);
    root->right->right->left = newNode(10);

    printf("Following are all keys and their ancestors\n");
    for (int key = 1; key <= 10; key++)
    {
        printf("%d: ", key);
        printAncestors(root, key);
        printf("\n");
    }

    getchar();
    return 0;
}

```

Output:

```

Following are all keys and their ancestors
1:
2: 1
3: 1
4: 2 1
5: 2 1
6: 3 1
7: 3 1
8: 4 2 1
9: 5 2 1
10: 7 3 1

```

Exercise

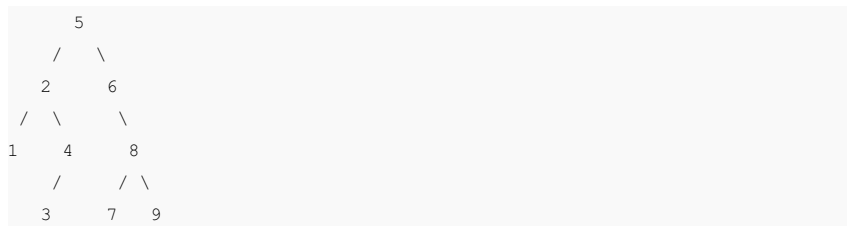
Note that the above solution assumes that the given key is present in the given Binary Tree. It may go in infinite loop if key is not present. Extend the above solution to work even when the key is not present in tree.

This article is contributed by **Chandra Prakash**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

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

Given 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. See following for implementation of this approach.

C implementation of level order traversal based approach to find the difference.

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. Following is C implementation of this approach.

```

// A recursive program to find difference between sum of nodes at
// odd level and sum at even level
#include <stdio.h>
#include <stdlib.h>

// Binary Tree node
struct node
{
    int data;
    struct node* left, *right;
};

// A utility function to allocate a new tree node with given data
struct node* newNode(int data)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// The main function that return difference between odd and even level
// nodes
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);
}

// Driver program to test above functions
int main()
{
    struct node *root = newNode(5);
    root->left = newNode(2);
    root->right = newNode(6);
    root->left->left = newNode(1);
    root->left->right = newNode(4);
    root->left->right->left = newNode(3);
    root->right->right = newNode(8);
    root->right->right->right = newNode(9);
    root->right->right->left = newNode(7);
    printf("%d is the required difference\n", getLevelDiff(root));
    getchar();
    return 0;
}

```

Output:

```
-9 is the required difference
```

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

This article is contributed by **Chandra Prakash**. Please write comments if you find

anything incorrect, or you want to share more information about the topic discussed above.

93. Print Postorder traversal from given Inorder and Preorder traversals

Given Inorder and Preorder traversals of a binary tree, print Postorder traversal.

Example:

Input:

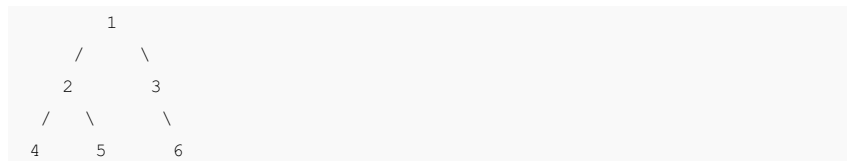
Inorder traversal in[] = {4, 2, 5, 1, 3, 6}

Preorder traversal pre[] = {1, 2, 4, 5, 3, 6}

Output:

Postorder traversal is {4, 5, 2, 6, 3, 1}

Traversals 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.

```

// C++ program to print postorder traversal from preorder and inorder
#include <iostream>
using namespace std;

// A utility function to search x in arr[] of size n
int search(int arr[], int x, int n)
{
    for (int i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}

// Prints postorder traversal from given inorder and preorder traversal
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] << " ";
}

// Driver program to test above functions
int main()
{
    int in[] = {4, 2, 5, 1, 3, 6};
    int pre[] = {1, 2, 4, 5, 3, 6};
    int n = sizeof(in)/sizeof(in[0]);
    cout << "Postorder traversal " << endl;
    printPostOrder(in, pre, n);
    return 0;
}

```

Output

```

Postorder traversal
4 5 2 6 3 1

```

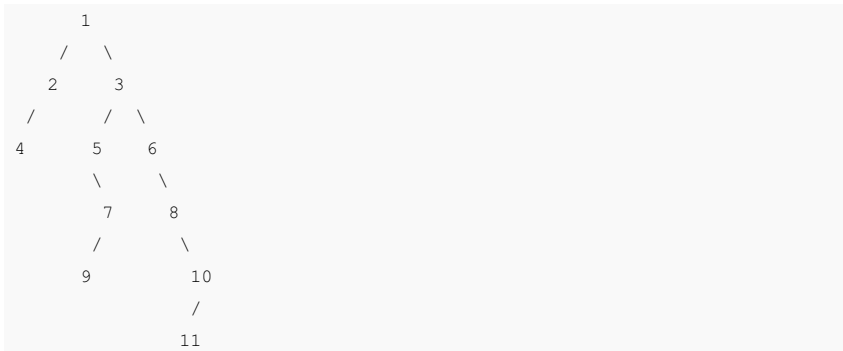
Time Complexity: The above function visits every node in array. For every visit, it calls search which takes $O(n)$ time. Therefore, overall time complexity of the function is $O(n^2)$

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

94. Find depth of the deepest odd level leaf 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.



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

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.

```
// C program to find depth of the deepest odd level leaf node
#include <stdio.h>
#include <stdlib.h>

// A utility function to find maximum of two integers
int max(int x, int y) { return (x > y)? x : y; }

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

// A utility function to allocate a new tree node
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// A recursive function to find depth of the deepest odd level leaf
```

```
// 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%2)
        return level;

    // If not leaf, return the maximum value from left and right subtree
    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);
}
```

```
// Driver program to test above functions
int main()
{
    struct Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->right->left = newNode(5);
    root->right->right = newNode(6);
    root->right->left->right = newNode(7);
    root->right->right->right = newNode(8);
    root->right->left->right->left = newNode(9);
    root->right->right->right->right = newNode(10);
    root->right->right->right->right->left = newNode(11);

    printf("%d is the required depth\n", depthOfOddLeaf(root));
    getchar();
    return 0;
}
```

Output:

```
5 is the required depth
```

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

This article is contributed by **Chandra Prakash**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

95. Check if all leaves are at same level

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

```
      12
     /  \
    5    7
   /      \
  3         1
Leaves are at same level
```

```
      12
     /  \
    5    7
   /
  3
Leaves are Not at same level
```

```
      12
     /
    5
   /  \
  3    9
 /      \
1        2
Leaves are at same level
```

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

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.

```
// C program to check if all leaves are at same level
#include <stdio.h>
#include <stdlib.h>

// A binary tree node
struct Node
{
    int data;
    struct Node *left, *right;
};

// A utility function to allocate a new tree node
```



```

struct Node* newNode(int data)
{
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

/* 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 leaves are at same level.
   It mainly uses checkUtil() */
bool check(struct Node *root)
{
    int level = 0, leafLevel = 0;
    return checkUtil(root, level, &leafLevel);
}

// Driver program to test above function
int main()
{
    // Let us create tree shown in third example
    struct Node *root = newNode(12);
    root->left = newNode(5);
    root->left->left = newNode(3);
    root->left->right = newNode(9);
    root->left->left->left = newNode(1);
    root->left->right->left = newNode(1);
    if (check(root))
        printf("Leaves are at same level\n");
    else
        printf("Leaves are not at same level\n");
    getchar();
    return 0;
}

```

Output:

Leaves are at same level

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

This article is contributed by **Chandra Prakash**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

96. Print Left View of a 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.



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). Following is C implementation of this approach.

```

// C program to print left view of Binary Tree
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int data;
    struct node *left, *right;
};

// A utility function to create a new Binary Tree node
struct node *newNode(int item)
{
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

// 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);
}

// Driver Program to test above functions
int main()
{
    struct node *root = newNode(12);
    root->left = newNode(10);
    root->right = newNode(30);
    root->right->left = newNode(25);
    root->right->right = newNode(40);

    leftView(root);

    return 0;
}

```

Output:

```

12      10      25

```

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

This article is contributed by Ramsai Chinthamani. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

97. B-Tree | Set 3 (Delete)

It is recommended to refer following posts as prerequisite of this post.

[B-Tree | Set 1 \(Introduction\)](#)

[B-Tree | Set 2 \(Insert\)](#)

B-Tree is a type of a multi-way search tree. So, if you are not familiar with multi-way search trees in general, it is better to take a look at [this video lecture from IIT-Delhi](#), before proceeding further. Once you get the basics of a multi-way search tree clear, B-Tree operations will be easier to understand.

Source of the following explanation and algorithm is [Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest](#)

Deletion process:

Deletion from a B-tree is more complicated than insertion, because we can delete a key from any node—not just a leaf—and when we delete a key from an internal node, we will have to rearrange the node's children.

As in insertion, we must make sure the deletion doesn't violate the [B-tree properties](#). Just as we had to ensure that a node didn't get too big due to insertion, we must ensure that a node doesn't get too small during deletion (except that the root is allowed to have fewer than the minimum number $t-1$ of keys). Just as a simple insertion algorithm might have to back up if a node on the path to where the key was to be inserted was full, a simple approach to deletion might have to back up if a node (other than the root) along the path to where the key is to be deleted has the minimum number of keys.

The deletion procedure deletes the key k from the subtree rooted at x . This procedure guarantees that whenever it calls itself recursively on a node x , the number of keys in x is at least the minimum degree t . Note that this condition requires one more key than the minimum required by the usual B-tree conditions, so that sometimes a key may have to be moved into a child node before recursion descends to that child. This strengthened condition allows us to delete a key from the tree in one downward pass without having to "back up" (with one exception, which we'll explain). You should interpret the following specification for deletion from a B-tree with the understanding that if the root node x

ever becomes an internal node having no keys (this situation can occur in cases 2c and 3b then we delete x , and x 's only child $x.c_1$ becomes the new root of the tree, decreasing the height of the tree by one and preserving the property that the root of the tree contains at least one key (unless the tree is empty).

We sketch how deletion works with various cases of deleting keys from a B-tree.

1. If the key k is in node x and x is a leaf, delete the key k from x .

2. If the key k is in node x and x is an internal node, do the following.

a) If the child y that precedes k in node x has at least t keys, then find the predecessor k_0 of k in the sub-tree rooted at y . Recursively delete k_0 , and replace k by k_0 in x . (We can find k_0 and delete it in a single downward pass.)

b) If y has fewer than t keys, then, symmetrically, examine the child z that follows k in node x . If z has at least t keys, then find the successor k_0 of k in the subtree rooted at z . Recursively delete k_0 , and replace k by k_0 in x . (We can find k_0 and delete it in a single downward pass.)

c) Otherwise, if both y and z have only $t-1$ keys, merge k and all of z into y , so that x loses both k and the pointer to z , and y now contains $2t-1$ keys. Then free z and recursively delete k from y .

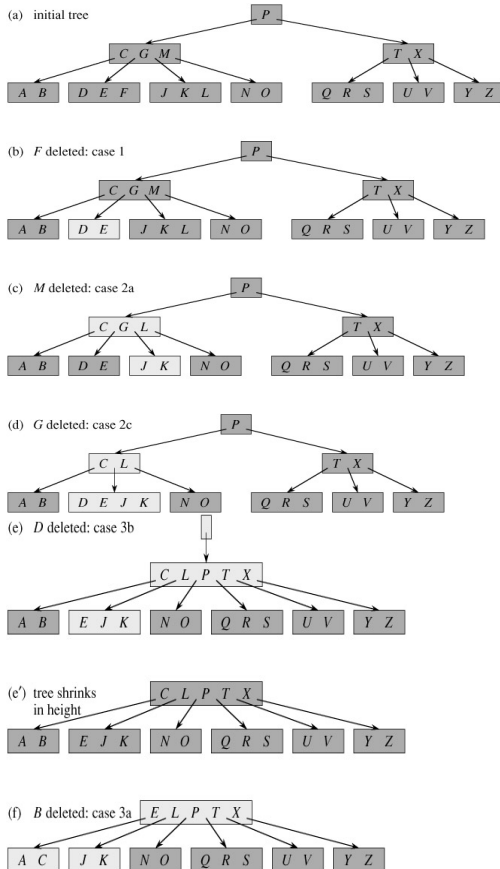
3. If the key k is not present in internal node x , determine the root $x.c(i)$ of the appropriate subtree that must contain k , if k is in the tree at all. If $x.c(i)$ has only $t-1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least t keys. Then finish by recursing on the appropriate child of x .

a) If $x.c(i)$ has only $t-1$ keys but has an immediate sibling with at least t keys, give $x.c(i)$ an extra key by moving a key from x down into $x.c(i)$, moving a key from $x.c(i)$'s immediate left or right sibling up into x , and moving the appropriate child pointer from the sibling into $x.c(i)$.

b) If $x.c(i)$ and both of $x.c(i)$'s immediate siblings have $t-1$ keys, merge $x.c(i)$ with one sibling, which involves moving a key from x down into the new merged node to become the median key for that node.

Since most of the keys in a B-tree are in the leaves, deletion operations are most often used to delete keys from leaves. The recursive delete procedure then acts in one downward pass through the tree, without having to back up. When deleting a key in an internal node, however, the procedure makes a downward pass through the tree but may have to return to the node from which the key was deleted to replace the key with its predecessor or successor (cases 2a and 2b).

The following figures from [CLRS book](#) explain the deletion process.



Implementation:

Following is C++ implementation of deletion process.

/* The following program performs deletion on a B-Tree. It contains functions specific for deletion along with all the other functions provided in previous articles on B-Trees. See <http://www.geeksforgeeks.org/b-tree/> for previous article.

The deletion function has been compartmentalized into 8 functions for better understanding and clarity

The following functions are exclusive for deletion

In class BTreeNode:

- 1) remove
- 2) removeFromLeaf
- 3) removeFromNonLeaf
- 4) getPred
- 5) getSucc
- 6) borrowFromPrev
- 7) borrowFromNext
- 8) merge
- 9) findKey

In class BTree:

1) remove

The removal of a key from a B-Tree is a fairly complicated process. It involves all the 6 different cases that might arise while removing a key.

Testing: The code has been tested using the B-Tree provided in the C++ code (in the main function) along with other cases.

Reference: CLRS3 - Chapter 18 - (499-502)

It is advised to read the material in CLRS before taking a look at the code.

```
#include<iostream>
using namespace std;

// A BTree node
class BTreeNode
{
    int *keys; // An array of keys
    int t; // Minimum degree (defines the range for number of keys)
    BTreeNode **C; // An array of child pointers
    int n; // Current number of keys
    bool leaf; // Is true when node is leaf. Otherwise false

public:
    BTreeNode(int _t, bool _leaf); // Constructor

    // A function to traverse all nodes in a subtree rooted with this node
    void traverse();

    // A function to search a key in subtree rooted with this node.
    BTreeNode *search(int k); // returns NULL if k is not present.

    // A function that returns the index of the first key that is greater
    // or equal to k
    int findKey(int k);

    // A utility function to insert a new key in the subtree rooted with
    // this node. The assumption is, the node must be non-full when the
    // function is called
    void insertNonFull(int k);

    // A utility function to split the child y of this node. i is index
    // of y in child array C[]. The Child y must be full when this
    // function is called
    void splitChild(int i, BTreeNode *y);

    // A wrapper function to remove the key k in subtree rooted with
    // this node.
    void remove(int k);

    // A function to remove the key present in idx-th position in
    // this node which is a leaf
    void removeFromLeaf(int idx);

    // A function to remove the key present in idx-th position in
    // this node which is a non-leaf node
    void removeFromNonLeaf(int idx);

    // A function to get the predecessor of the key- where the key
    // is present in the idx-th position in the node
    int getPredecessor(int idx);
```

```

    int getPred(int idx);

    // A function to get the successor of the key- where the key
    // is present in the idx-th position in the node
    int getSucc(int idx);

    // A function to fill up the child node present in the idx-th
    // position in the C[] array if that child has less than t-1 keys
    void fill(int idx);

    // A function to borrow a key from the C[idx-1]-th node and place
    // it in C[idx]th node
    void borrowFromPrev(int idx);

    // A function to borrow a key from the C[idx+1]-th node and place
    // in C[idx]th node
    void borrowFromNext(int idx);

    // A function to merge idx-th child of the node with (idx+1)th child
    // the node
    void merge(int idx);

    // Make BTree friend of this so that we can access private members
    // this class in BTree functions
    friend class BTree;
};

class BTree
{
    BTreeNode *root; // Pointer to root node
    int t; // Minimum degree
public:

    // Constructor (Initializes tree as empty)
    BTree(int _t)
    {
        root = NULL;
        t = _t;
    }

    void traverse()
    {
        if (root != NULL) root->traverse();
    }

    // function to search a key in this tree
    BTreeNode* search(int k)
    {
        return (root == NULL)? NULL : root->search(k);
    }

    // The main function that inserts a new key in this B-Tree
    void insert(int k);

    // The main function that removes a new key in this B-Tree
    void remove(int k);
};

BTreeNode::BTreeNode(int t1, bool leaf1)
{
    // Copy the given minimum degree and leaf property
    t = t1;
    leaf = leaf1;
}

```



```

    t = t-1,
    leaf = leaf1;

    // Allocate memory for maximum number of possible keys
    // and child pointers
    keys = new int[2*t-1];
    C = new BTreeNode *[2*t];

    // Initialize the number of keys as 0
    n = 0;
}

// A utility function that returns the index of the first key that is
// greater than or equal to k
int BTreeNode::findKey(int k)
{
    int idx=0;
    while (idx<n && keys[idx] < k)
        ++idx;
    return idx;
}

// A function to remove the key k from the sub-tree rooted with this node
void BTreeNode::remove(int k)
{
    int idx = findKey(k);

    // The key to be removed is present in this node
    if (idx < n && keys[idx] == k)
    {
        // If the node is a leaf node - removeFromLeaf is called
        // Otherwise, removeFromNonLeaf function is called
        if (leaf)
            removeFromLeaf(idx);
        else
            removeFromNonLeaf(idx);
    }
    else
    {
        // If this node is a leaf node, then the key is not present in
        if (leaf)
        {
            cout << "The key "<< k << " is does not exist in the tree\n";
            return;
        }

        // The key to be removed is present in the sub-tree rooted with
        // The flag indicates whether the key is present in the sub-tree
        // with the last child of this node
        bool flag = ( (idx==n)? true : false );

        // If the child where the key is supposed to exist has less than t
        // we fill that child
        if (C[idx]->n < t)
            fill(idx);

        // If the last child has been merged, it must have merged with
        // child and so we recurse on the (idx-1)th child. Else, we recurse
        // (idx)th child which now has atleast t keys
        if (flag && idx > n)
            C[idx-1]->remove(k);
    }
}

```

```

        else
            C[idx]->remove(k);
    }
    return;
}

// A function to remove the idx-th key from this node - which is a leaf
void BTreeNode::removeFromLeaf (int idx)
{
    // Move all the keys after the idx-th pos one place backward
    for (int i=idx+1; i<n; ++i)
        keys[i-1] = keys[i];

    // Reduce the count of keys
    n--;

    return;
}

// A function to remove the idx-th key from this node - which is a non
void BTreeNode::removeFromNonLeaf(int idx)
{
    int k = keys[idx];

    // If the child that precedes k (C[idx]) has atleast t keys,
    // find the predecessor 'pred' of k in the subtree rooted at
    // C[idx]. Replace k by pred. Recursively delete pred
    // in C[idx]
    if (C[idx]->n >= t)
    {
        int pred = getPred(idx);
        keys[idx] = pred;
        C[idx]->remove(pred);
    }

    // If the child C[idx] has less than t keys, examine C[idx+1].
    // If C[idx+1] has atleast t keys, find the successor 'succ' of k :
    // the subtree rooted at C[idx+1]
    // Replace k by succ
    // Recursively delete succ in C[idx+1]
    else if (C[idx+1]->n >= t)
    {
        int succ = getSucc(idx);
        keys[idx] = succ;
        C[idx+1]->remove(succ);
    }

    // If both C[idx] and C[idx+1] has less than t keys, merge k and all
    // into C[idx]
    // Now C[idx] contains 2t-1 keys
    // Free C[idx+1] and recursively delete k from C[idx]
    else
    {
        merge(idx);
        C[idx]->remove(k);
    }
    return;
}

// A function to get predecessor of keys[idx]

```

```

int BTreeNode::getPred(int idx)
{
    // Keep moving to the right most node until we reach a leaf
    BTreeNode *cur=C[idx];
    while (!cur->leaf)
        cur = cur->C[cur->n];

    // Return the last key of the leaf
    return cur->keys[cur->n-1];
}

int BTreeNode::getSucc(int idx)
{
    // Keep moving the left most node starting from C[idx+1] until we reach a leaf
    BTreeNode *cur = C[idx+1];
    while (!cur->leaf)
        cur = cur->C[0];

    // Return the first key of the leaf
    return cur->keys[0];
}

// A function to fill child C[idx] which has less than t-1 keys
void BTreeNode::fill(int idx)
{
    // If the previous child(C[idx-1]) has more than t-1 keys, borrow a key
    // from that child
    if (idx!=0 && C[idx-1]->n>=t)
        borrowFromPrev(idx);

    // If the next child(C[idx+1]) has more than t-1 keys, borrow a key
    // from that child
    else if (idx!=n && C[idx+1]->n>=t)
        borrowFromNext(idx);

    // Merge C[idx] with its sibling
    // If C[idx] is the last child, merge it with its previous sibling
    // Otherwise merge it with its next sibling
    else
    {
        if (idx != n)
            merge(idx);
        else
            merge(idx-1);
    }
    return;
}

// A function to borrow a key from C[idx-1] and insert it
// into C[idx]
void BTreeNode::borrowFromPrev(int idx)
{
    BTreeNode *child=C[idx];
    BTreeNode *sibling=C[idx-1];

    // The last key from C[idx-1] goes up to the parent and key[idx-1]
    // from parent is inserted as the first key in C[idx]. Thus, the
    // sibling loses one key and child gains one key

    // Moving all keys in C[idx] one step ahead

```

```

// moving all key in C[idx] one step ahead
for (int i=child->n-1; i>=0; --i)
    child->keys[i+1] = child->keys[i];

// If C[idx] is not a leaf, move all its child pointers one step al
if (!child->leaf)
{
    for(int i=child->n; i>=0; --i)
        child->C[i+1] = child->C[i];
}

// Setting child's first key equal to keys[idx-1] from the current
child->keys[0] = keys[idx-1];

// Moving sibling's last child as C[idx]'s first child
if (!leaf)
    child->C[0] = sibling->C[sibling->n];

// Moving the key from the sibling to the parent
// This reduces the number of keys in the sibling
keys[idx-1] = sibling->keys[sibling->n-1];

child->n += 1;
sibling->n -= 1;

return;
}

// A function to borrow a key from the C[idx+1] and place
// it in C[idx]
void BTreeNode::borrowFromNext(int idx)
{
    BTreeNode *child=C[idx];
    BTreeNode *sibling=C[idx+1];

    // keys[idx] is inserted as the last key in C[idx]
    child->keys[(child->n)] = keys[idx];

    // Sibling's first child is inserted as the last child
    // into C[idx]
    if (!(child->leaf))
        child->C[(child->n)+1] = sibling->C[0];

    //The first key from sibling is inserted into keys[idx]
    keys[idx] = sibling->keys[0];

    // Moving all keys in sibling one step behind
    for (int i=1; i<sibling->n; ++i)
        sibling->keys[i-1] = sibling->keys[i];

    // Moving the child pointers one step behind
    if (!sibling->leaf)
    {
        for(int i=1; i<=sibling->n; ++i)
            sibling->C[i-1] = sibling->C[i];
    }

    // Increasing and decreasing the key count of C[idx] and C[idx+1]
    // respectively
    child->n += 1;
    sibling->n -= 1;
}

```

```

    return;
}

// A function to merge C[idx] with C[idx+1]
// C[idx+1] is freed after merging
void BTreeNode::merge(int idx)
{
    BTreeNode *child = C[idx];
    BTreeNode *sibling = C[idx+1];

    // Pulling a key from the current node and inserting it into (t-1)
    // position of C[idx]
    child->keys[t-1] = keys[idx];

    // Copying the keys from C[idx+1] to C[idx] at the end
    for (int i=0; i<sibling->n; ++i)
        child->keys[i+t] = sibling->keys[i];

    // Copying the child pointers from C[idx+1] to C[idx]
    if (!child->leaf)
    {
        for(int i=0; i<=sibling->n; ++i)
            child->C[i+t] = sibling->C[i];
    }

    // Moving all keys after idx in the current node one step before -
    // to fill the gap created by moving keys[idx] to C[idx]
    for (int i=idx+1; i<n; ++i)
        keys[i-1] = keys[i];

    // Moving the child pointers after (idx+1) in the current node one
    // step before
    for (int i=idx+2; i<=n; ++i)
        C[i-1] = C[i];

    // Updating the key count of child and the current node
    child->n += sibling->n+1;
    n--;

    // Freeing the memory occupied by sibling
    delete(sibling);
    return;
}

// The main function that inserts a new key in this B-Tree
void BTree::insert(int k)
{
    // If tree is empty
    if (root == NULL)
    {
        // Allocate memory for root
        root = new BTreeNode(t, true);
        root->keys[0] = k; // Insert key
        root->n = 1; // Update number of keys in root
    }
    else // If tree is not empty
    {
        // If root is full, then tree grows in height
        if (root->n == 2*t-1)
        {
            // Allocate memory for new root
            BTreeNode *s = new BTreeNode(t, false);

```

```

        // Make old root as child of new root
        s->C[0] = root;

        // Split the old root and move 1 key to the new root
        s->splitChild(0, root);

        // New root has two children now. Decide which of the
        // two children is going to have new key
        int i = 0;
        if (s->keys[0] < k)
            i++;
        s->C[i]->insertNonFull(k);

        // Change root
        root = s;
    }
    else // If root is not full, call insertNonFull for root
        root->insertNonFull(k);
}

// A utility function to insert a new key in this node
// The assumption is, the node must be non-full when this
// function is called
void BTreeNode::insertNonFull(int k)
{
    // Initialize index as index of rightmost element
    int i = n-1;

    // If this is a leaf node
    if (leaf == true)
    {
        // The following loop does two things
        // a) Finds the location of new key to be inserted
        // b) Moves all greater keys to one place ahead
        while (i >= 0 && keys[i] > k)
        {
            keys[i+1] = keys[i];
            i--;
        }

        // Insert the new key at found location
        keys[i+1] = k;
        n = n+1;
    }
    else // If this node is not leaf
    {
        // Find the child which is going to have the new key
        while (i >= 0 && keys[i] > k)
            i--;

        // See if the found child is full
        if (C[i+1]->n == 2*t-1)
        {
            // If the child is full, then split it
            splitChild(i+1, C[i+1]);

            // After split, the middle key of C[i] goes up and
            // C[i] is splitted into two. See which of the two
            // is going to have the new key
            if (keys[i+1] < k)
                i++;
        }
    }
}

```

```

        ...
    }
    C[i+1]->insertNonFull(k);
}

// A utility function to split the child y of this node
// Note that y must be full when this function is called
void BTreeNode::splitChild(int i, BTreeNode *y)
{
    // Create a new node which is going to store (t-1) keys
    // of y
    BTreeNode *z = new BTreeNode(y->t, y->leaf);
    z->n = t - 1;

    // Copy the last (t-1) keys of y to z
    for (int j = 0; j < t-1; j++)
        z->keys[j] = y->keys[j+t];

    // Copy the last t children of y to z
    if (y->leaf == false)
    {
        for (int j = 0; j < t; j++)
            z->C[j] = y->C[j+t];
    }

    // Reduce the number of keys in y
    y->n = t - 1;

    // Since this node is going to have a new child,
    // create space of new child
    for (int j = n; j >= i+1; j--)
        C[j+1] = C[j];

    // Link the new child to this node
    C[i+1] = z;

    // A key of y will move to this node. Find location of
    // new key and move all greater keys one space ahead
    for (int j = n-1; j >= i; j--)
        keys[j+1] = keys[j];

    // Copy the middle key of y to this node
    keys[i] = y->keys[t-1];

    // Increment count of keys in this node
    n = n + 1;
}

// Function to traverse all nodes in a subtree rooted with this node
void BTreeNode::traverse()
{
    // There are n keys and n+1 children, travers through n keys
    // and first n children
    int i;
    for (i = 0; i < n; i++)
    {
        // If this is not leaf, then before printing key[i],
        // traverse the subtree rooted with child C[i].
        if (leaf == false)
            C[i]->traverse();
        cout << " " << keys[i];
    }
}

```

```

// Print the subtree rooted with last child
if (leaf == false)
    C[i]->traverse();
}

// Function to search key k in subtree rooted with this node
BTreeNode *BTreeNode::search(int k)
{
    // Find the first key greater than or equal to k
    int i = 0;
    while (i < n && k > keys[i])
        i++;

    // If the found key is equal to k, return this node
    if (keys[i] == k)
        return this;

    // If key is not found here and this is a leaf node
    if (leaf == true)
        return NULL;

    // Go to the appropriate child
    return C[i]->search(k);
}

void BTree::remove(int k)
{
    if (!root)
    {
        cout << "The tree is empty\n";
        return;
    }

    // Call the remove function for root
    root->remove(k);

    // If the root node has 0 keys, make its first child as the new root
    // if it has a child, otherwise set root as NULL
    if (root->n==0)
    {
        BTreeNode *tmp = root;
        if (root->leaf)
            root = NULL;
        else
            root = root->C[0];

        // Free the old root
        delete tmp;
    }
    return;
}

// Driver program to test above functions
int main()
{
    BTree t(3); // A B-Tree with minium degree 3

    t.insert(1);
    t.insert(3);
    t.insert(7);
    t.insert(10);
}

```



```

t.insert(11);
t.insert(13);
t.insert(14);
t.insert(15);
t.insert(18);
t.insert(16);
t.insert(19);
t.insert(24);
t.insert(25);
t.insert(26);
t.insert(21);
t.insert(4);
t.insert(5);
t.insert(20);
t.insert(22);
t.insert(2);
t.insert(17);
t.insert(12);
t.insert(6);

cout << "Traversal of tree constructed is\n";
t.traverse();
cout << endl;

t.remove(6);
cout << "Traversal of tree after removing 6\n";
t.traverse();
cout << endl;

t.remove(13);
cout << "Traversal of tree after removing 13\n";
t.traverse();
cout << endl;

t.remove(7);
cout << "Traversal of tree after removing 7\n";
t.traverse();
cout << endl;

t.remove(4);
cout << "Traversal of tree after removing 4\n";
t.traverse();
cout << endl;

t.remove(2);
cout << "Traversal of tree after removing 2\n";
t.traverse();
cout << endl;

t.remove(16);
cout << "Traversal of tree after removing 16\n";
t.traverse();
cout << endl;

return 0;
}

```

Output:

```

Traversal of tree constructed is
1 2 3 4 5 6 7 10 11 12 13 14 15 16 17 18 19 20 21 22 24 25 26

```

```

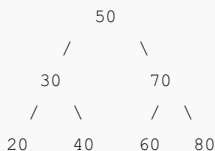
Traversal of tree after removing 6
1 2 3 4 5 7 10 11 12 13 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 13
1 2 3 4 5 7 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 7
1 2 3 4 5 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 4
1 2 3 5 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 2
1 3 5 10 11 12 14 15 16 17 18 19 20 21 22 24 25 26
Traversal of tree after removing 16
1 3 5 10 11 12 14 15 17 18 19 20 21 22 24 25 26

```

This article is contributed by **Balasubramanian.N** . Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

98. Add all greater values to every node in a 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.



The above tree should be modified to following



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

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.

```
// C program to add all greater values in every node of BST
#include<stdio.h>
#include<stdlib.h>

struct node
{
    int data;
    struct node *left, *right;
};

// A utility function to create a new BST node
struct node *newNode(int item)
{
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Recursive function to add all greater values in every node
void modifyBSTUtil(struct node *root, int *sum)
{
    // Base Case
    if (root == NULL) return;

    // Recur for right subtree
    modifyBSTUtil(root->right, sum);

    // Now *sum has sum of nodes in right subtree, add
    // root->data to sum and update root->data
    *sum = *sum + root->data;
    root->data = *sum;

    // Recur for left subtree
    modifyBSTUtil(root->left, sum);
}

// A wrapper over modifyBSTUtil()
void modifyBST(struct node *root)
{
    int sum = 0;
    modifyBSTUtil(root, &sum);
}

// A utility function to do inorder traversal of BST
void inorder(struct node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d ", root->data);
        inorder(root->right);
    }
}

/* A utility function to insert a new node with given data in BST */
struct node* insert(struct node* node, int data)
{
    /* If the tree is empty. return a new node */

```

```

/* If the given element is empty, return a new node */
if (node == NULL) return newNode(data);

/* Otherwise, recur down the tree */
if (data <= node->data)
    node->left = insert(node->left, data);
else
    node->right = insert(node->right, data);

/* return the (unchanged) node pointer */
return node;
}

// Driver Program to test above functions
int main()
{
    /* Let us create following BST
            50
           /  \
          30   70
         /  \  /  \
        20  40 60  80 */
    struct node *root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    modifyBST(root);

    // print inorder traversal of the modified BST
    inorder(root);

    return 0;
}

```

Output

```
350 330 300 260 210 150 80
```

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 k th largest element in a BST.

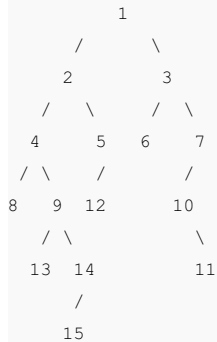
This article is contributed by **Chandra Prakash**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

99. Remove all nodes which don't lie in any path with $\text{sum} \geq k$

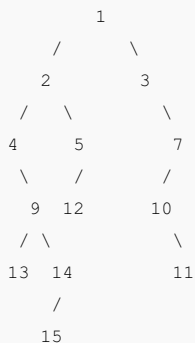
Given a binary tree, a complete path is defined as a path from root to a leaf. The sum of all nodes on that path is defined as the sum of that path. Given a number K, you have to remove (prune the tree) all nodes which don't lie in any path with $\text{sum} \geq k$.

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.

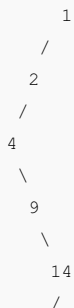
Consider the following Binary Tree



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.



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

The idea is to traverse the tree and delete nodes in bottom up manner. While traversing the tree, recursively calculate the sum of nodes from root to leaf node of each path. For each visited node, checks the total calculated sum against given sum "k". If sum is less than k, then free(delete) that node (leaf node) and return the sum back to the previous node. Since the path is from root to leaf and nodes are deleted in bottom up manner, a node is deleted only when all of its descendants are deleted. Therefore, when a node is deleted, it must be a leaf in the current Binary Tree.

Following is C implementation of the above approach.

```
#include <stdio.h>
#include <stdlib.h>

// A utility function to get maximum of two integers
int max(int l, int r) { return (l > r ? l : r); }

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

// A utility function to create a new Binary Tree node with given data
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// print the tree in LVR (Inorder traversal) way.
void print(struct Node *root)
{
    if (root != NULL)
    {
        print(root->left);
        printf("%d ", root->data);
        print(root->right);
    }
}

/* Main function which truncates the binary tree. */
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);
}

// Driver program to test above function
int main()
{
    int k = 45;
    struct Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->left->left->left = newNode(8);
    root->left->left->right = newNode(9);
    root->left->right->left = newNode(12);
    root->right->right->left = newNode(10);
    root->right->right->left->right = newNode(11);
    root->left->left->right->left = newNode(13);
    root->left->left->right->right = newNode(14);
    root->left->left->right->right->left = newNode(15);

    printf("Tree before truncation\n");
    print(root);

    root = prune(root, k); // k is 45

    printf("\n\nTree after truncation\n");
    print(root);

    return 0;
}

```

Output:

```
Tree before truncation
```

```
8 4 13 9 15 14 2 12 5 1 6 3 10 11 7
```

Tree after truncation

```
4 9 15 14 2 1
```

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

A Simpler Solution:

The above code can be simplified using 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.

Thanks to vicky for suggesting this solution in below comments.

```
#include <stdio.h>
#include <stdlib.h>

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

// A utility function to create a new Binary
// Tree node with given data
struct Node* newNode(int data)
{
    struct Node* node =
        (struct Node*) malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// print the tree in LVR (Inorder traversal) way.
void print(struct Node *root)
{
    if (root != NULL)
    {
        print(root->left);
        printf("%d ", root->data);
        print(root->right);
    }
}

/* Main function which truncates the binary tree. */
struct Node *prune(struct Node *root, int sum)
{
    // Base Case
    if (root == NULL) return NULL;

    // Recur for left and right subtrees
    root->left = prune(root->left, sum - root->data);
    root->right = prune(root->right, sum - root->data);
}
```



```

// If we reach leaf whose data is smaller than sum,
// we delete the leaf. An important thing to note
// is a non-leaf node can become leaf when its
// children are deleted.
if (root->left==NULL && root->right==NULL)
{
    if (root->data < sum)
    {
        free(root);
        return NULL;
    }
}

return root;
}

```

// Driver program to test above function

```

int main()
{
    int k = 45;
    struct Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->left->left->left = newNode(8);
    root->left->left->right = newNode(9);
    root->left->right->left = newNode(12);
    root->right->right->left = newNode(10);
    root->right->right->left->right = newNode(11);
    root->left->left->right->left = newNode(13);
    root->left->left->right->right = newNode(14);
    root->left->left->right->right->left = newNode(15);

    printf("Tree before truncation\n");
    print(root);

    root = prune(root, k); // k is 45

    printf("\n\nTree after truncation\n");
    print(root);

    return 0;
}

```

Output:

Tree before truncation

8 4 13 9 15 14 2 12 5 1 6 3 10 11 7

Tree after truncation

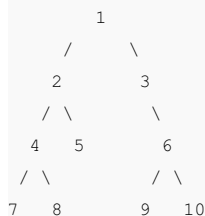
4 9 15 14 2 1

This article is contributed by **Chandra Prakash**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

100. Extract Leaves of a Binary Tree in a 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

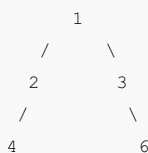


Output:

Doubly Linked List

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

Modified Tree:



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

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.

```
// C program to extract leaves of a Binary Tree in a Doubly Linked List
#include <stdio.h>
#include <stdlib.h>
```

```

// Structure for tree and linked list
struct Node
{
    int data;
    struct Node *left, *right;
};

// 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
// head of doubly linked list. left pointer of tree is used as prev in
// 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;
}

// Utility function for allocating node for Binary Tree.
struct Node* newNode(int data)
{
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// Utility function for printing tree in In-Order.
void print(struct Node *root)
{
    if (root != NULL)
    {
        print(root->left);
        printf("%d ", root->data);
        print(root->right);
    }
}

// Utility function for printing double linked list.
void printList(struct Node *head)

```

```

void printList(struct Node *head)
{
    while (head)
    {
        printf("%d ", head->data);
        head = head->right;
    }
}

// Driver program to test above function
int main()
{
    struct Node *head = NULL;
    struct Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(6);
    root->left->left->left = newNode(7);
    root->left->left->right = newNode(8);
    root->right->right->left = newNode(9);
    root->right->right->right = newNode(10);

    printf("Inorder Traversal of given Tree is:\n");
    print(root);

    root = extractLeafList(root, &head);

    printf("\nExtracted Double Linked list is:\n");
    printList(head);

    printf("\nInorder traversal of modified tree is:\n");
    print(root);
    return 0;
}

```

Output:

```

Inorder Traversal of given Tree is:
7 4 8 2 5 1 3 9 6 10
Extracted Double Linked list is:
7 8 5 9 10
Inorder traversal of modified tree is:
4 2 1 3 6

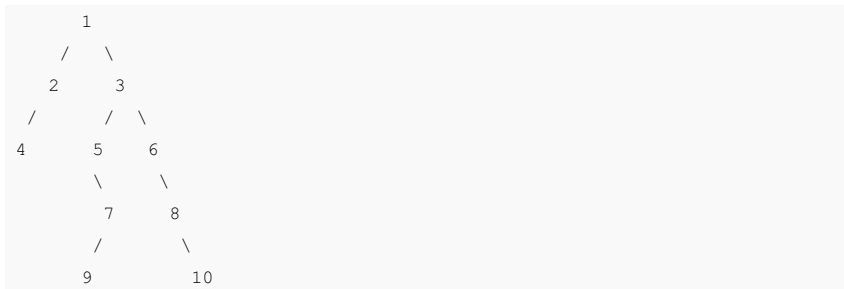
```

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

This article is contributed by **Chandra Prakash**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

101. 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.



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

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. Thanks to [Coder011](#) for suggesting this approach.

```
// A C++ program to find the deepest left leaf in a given binary tree
#include <stdio.h>
#include <iostream>
using namespace std;
```

```
struct Node
{
    int val;
    struct Node *left, *right;
};

Node *newNode(int data)
{
    Node *temp = new Node;
    temp->val = data;
    temp->left = temp->right = NULL;
    return temp;
}
```

```
// 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;
}

// Driver program to test above function
int main()
{
    Node* root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->right->left = newNode(5);
    root->right->right = newNode(6);
    root->right->left->right = newNode(7);
    root->right->right->right = newNode(8);
    root->right->left->right->left = newNode(9);
    root->right->right->right->right = newNode(10);

    Node *result = deepestLeftLeaf(root);
    if (result)
        cout << "The deepest left child is " << result->val;
    else
        cout << "There is no left leaf in the given tree";

    return 0;
}

```

Output:

```
The deepest left child is 9
```

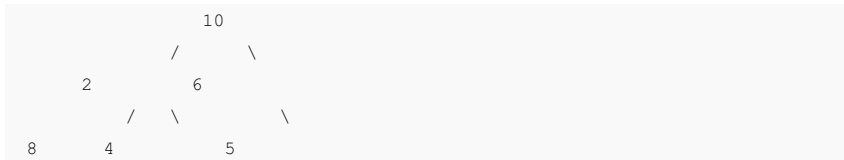
Time Complexity: The function does a simple traversal of the tree, so the complexity is $O(n)$.

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

102. 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.



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

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.

```
/* Program to find next right of a given key */
#include <iostream>
#include <queue>
using namespace std;

// A Binary Tree Node
struct node
{
    struct node *left, *right;
    int key;
};

// Method to find next right of given key k, it returns NULL if k is
// not present in tree or k is the rightmost node of its level
node* nextRight(node *root, int k)
{
    // Base Case
    if (root == NULL)
        return 0;

    // Create an empty queue for level order traversal
    queue<node*> qn; // A queue to store node addresses
    queue<int> ql;   // Another queue to store node levels

    int level = 0; // Initialize level as 0

    // Enqueue Root and its level
    qn.push(root);
    ql.push(level);

    // A standard BFS loop
    while (qn.size())
```

```

{
    // dequeue an node from qn and its level from ql
    node *node = qn.front();
    level = ql.front();
    qn.pop();
    ql.pop();

    // If the dequeued node has the given key k
    if (node->key == k)
    {
        // If there are no more items in queue or given node is
        // the rightmost node of its level, then return NULL
        if (ql.size() == 0 || ql.front() != level)
            return NULL;

        // Otherwise return next node from queue of nodes
        return qn.front();
    }

    // Standard BFS steps: enqueue children of this node
    if (node->left != NULL)
    {
        qn.push(node->left);
        ql.push(level+1);
    }
    if (node->right != NULL)
    {
        qn.push(node->right);
        ql.push(level+1);
    }
}

// We reach here if given key x doesn't exist in tree
return NULL;
}

```

// Utility function to create a new tree node
node* newNode(int key)

```

{
    node *temp = new node;
    temp->key = key;
    temp->left = temp->right = NULL;
    return temp;
}

```

// A utility function to test above functions

```

void test(node *root, int k)
{
    node *nr = nextRight(root, k);
    if (nr != NULL)
        cout << "Next Right of " << k << " is " << nr->key << endl;
    else
        cout << "No next right node found for " << k << endl;
}

```

// Driver program to test above functions

```

int main()
{
    // Let us create binary tree given in the above example
    node *root = newNode(10);
    root->left = newNode(2);
    root->right = newNode(6);
    root->right->right = newNode(5);
}

```



```

root->right->right = newNode(3);
root->left->left = newNode(8);
root->left->right = newNode(4);

test(root, 10);
test(root, 2);
test(root, 6);
test(root, 5);
test(root, 8);
test(root, 4);
return 0;
}

```

Output:

```

No next right node found for 10
Next Right of 2 is 6
No next right node found for 6
No next right node found for 5
Next Right of 8 is 4
Next Right of 4 is 5

```

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.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

103. Splay Tree | Set 1 (Search)

The worst case time complexity of Binary Search Tree (BST) operations like search, delete, insert is $O(n)$. The worst case occurs when the tree is skewed. We can get the worst case time complexity as $O(\log n)$ with [AVL](#) and Red-Black Trees.

Can we do better than AVL or Red-Black trees in practical situations?

Like [AVL](#) and Red-Black Trees, Splay tree is also [self-balancing BST](#). The main idea of splay tree is to bring the recently accessed item to root of the tree, this makes the recently searched item to be accessible in $O(1)$ time if accessed again. The idea is to use locality of reference (In a typical application, 80% of the access are to 20% of the items). Imagine a situation where we have millions or billions of keys and only few of them are accessed frequently, which is very likely in many practical applications.

All splay tree operations run in $O(\log n)$ time on average, where n is the number of

entries in the tree. Any single operation can take $\Theta(n)$ time in the worst case.

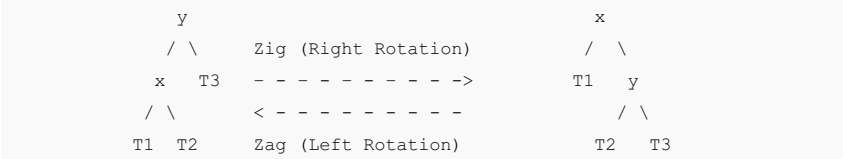
Search Operation

The search operation in Splay tree does the standard BST search, in addition to search, it also splays (move a node to the root). If the search is successful, then the node that is found is splayed and becomes the new root. Else the last node accessed prior to reaching the NULL is splayed and becomes the new root.

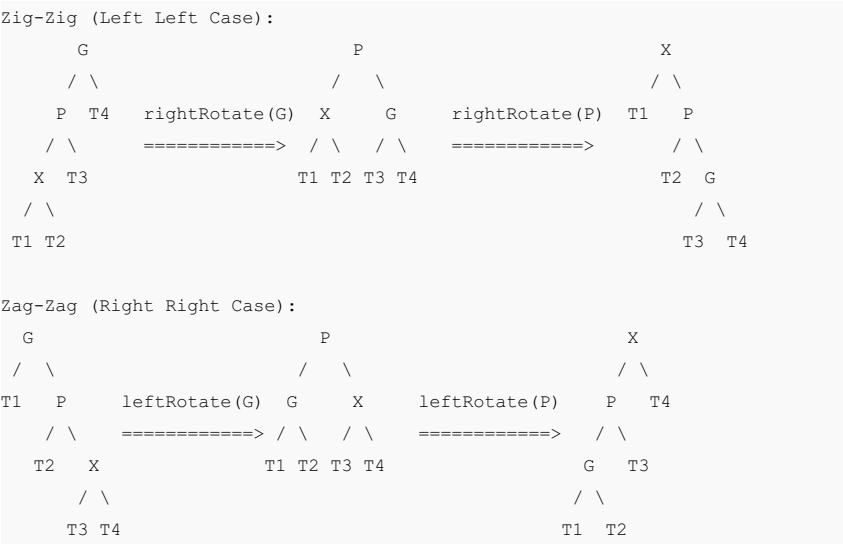
There are following cases for the node being accessed.

1) Node is root We simply return the root, don't do anything else as the accessed node is already root.

2) Zig: Node is child of root (the node has no grandparent). Node is either a left child of root (we do a right rotation) or node is a right child of its parent (we do a left rotation). T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side)



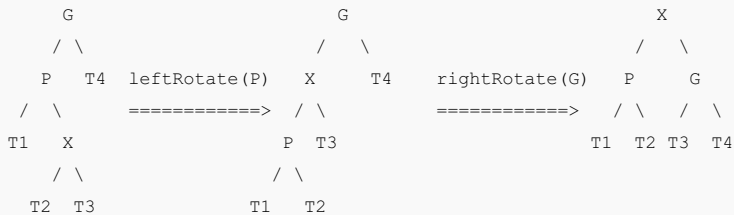
3) Node has both parent and grandparent. There can be following subcases.
.....**3.a) Zig-Zig and Zag-Zag** Node is left child of parent and parent is also left child of grand parent (Two right rotations) OR node is right child of its parent and parent is also right child of grand parent (Two Left Rotations).



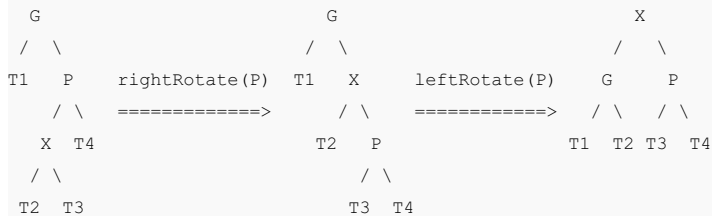
.....**3.b) Zig-Zag and Zag-Zig** Node is left child of parent and parent is right child of

grand parent (Left Rotation followed by right rotation) OR node is right child of its parent and parent is left child of grand parent (Right Rotation followed by left rotation).

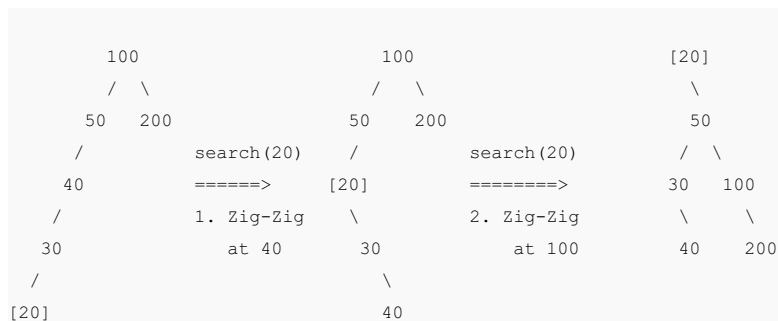
Zig-Zag (Left Right Case):



Zag-Zig (Right Left Case):



Example:



The important thing to note is, the search or splay operation not only brings the searched key to root, but also balances the BST. For example in above case, height of BST is reduced by 1.

Implementation:

```

// The code is adopted from http://goo.gl/SDH9hH
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct node
{
    int key;
    struct node *left, *right;
};
  
```

```

},

/* Helper function that allocates a new node with the given key and
   NULL left and right pointers. */
struct node* newNode(int key)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->key = key;
    node->left = node->right = NULL;
    return (node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct node *rightRotate(struct node *x)
{
    struct node *y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}

// This function brings the key at root if key is present in tree.
// If key is not present, then it brings the last accessed item at
// root. This function modifies the tree and returns the new root
struct node *splay(struct node *root, int key)
{
    // Base cases: root is NULL or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key lies in left subtree
    if (root->key > key)
    {
        // Key is not in tree, we are done
        if (root->left == NULL) return root;

        // Zig-Zig (Left Left)
        if (root->left->key > key)
        {
            // First recursively bring the key as root of left-left
            root->left->left = splay(root->left->left, key);

            // Do first rotation for root, second rotation is done after
            root = rightRotate(root);
        }
        // else if (root->left->key < key) // Zig-Zag (Left Right)
        {
            // First recursively bring the key as root of left-right
            root->left->right = splay(root->left->right, key);

            // Do first rotation for root->left
            if (root->left->right != NULL)

```

```

        root->left = leftRotate(root->left);
    }

    // Do second rotation for root
    return (root->left == NULL)? root: rightRotate(root);
}
else // Key lies in right subtree
{
    // Key is not in tree, we are done
    if (root->right == NULL) return root;

    // Zag-Zig (Right Left)
    if (root->right->key > key)
    {
        // Bring the key as root of right-left
        root->right->left = splay(root->right->left, key);

        // Do first rotation for root->right
        if (root->right->left != NULL)
            root->right = rightRotate(root->right);
    }
    else if (root->right->key < key) // Zag-Zag (Right Right)
    {
        // Bring the key as root of right-right and do first rotation
        root->right->right = splay(root->right->right, key);
        root = leftRotate(root);
    }

    // Do second rotation for root
    return (root->right == NULL)? root: leftRotate(root);
}
}

// The search function for Splay tree. Note that this function
// returns the new root of Splay Tree. If key is present in tree
// then, it is moved to root.
struct node *search(struct node *root, int key)
{
    return splay(root, key);
}

// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(struct node *root)
{
    if (root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Driver program to test above function*/
int main()
{
    struct node *root = newNode(100);
    root->left = newNode(50);
    root->right = newNode(200);
    root->left->left = newNode(40);
    root->left->left->left = newNode(30);
    root->left->left->left->left = newNode(20);

```

```

    root = search(root, 20);
    printf("Preorder traversal of the modified Splay tree is \n");
    preOrder(root);
    return 0;
}

```

Output:

```

Preorder traversal of the modified Splay tree is
20 50 30 40 100 200

```

Summary

- 1) Splay trees have excellent locality properties. Frequently accessed items are easy to find. Infrequent items are out of way.
- 2) All splay tree operations take $O(\log n)$ time on average. Splay trees can be rigorously shown to run in $O(\log n)$ average time per operation, over any sequence of operations (assuming we start from an empty tree)
- 3) Splay trees are simpler compared to **AVL** and Red-Black Trees as no extra field is required in every tree node.
- 4) Unlike **AVL tree**, a splay tree can change even with read-only operations like search.

Applications of Splay Trees

Splay trees have become the most widely used basic data structure invented in the last 30 years, because they're the fastest type of balanced search tree for many applications.

Splay trees are used in Windows NT (in the virtual memory, networking, and file system code), the gcc compiler and GNU C++ library, the sed string editor, Fore Systems network routers, the most popular implementation of Unix malloc, Linux loadable kernel modules, and in much other software (Source:

<http://www.cs.berkeley.edu/~jrs/61b/lec/36>)

We will soon be discussing insert and delete operations on splay trees.

References:

<http://www.cs.berkeley.edu/~jrs/61b/lec/36>

<http://www.cs.cornell.edu/courses/cs3110/2009fa/recitations/rec-splay.html>

<http://courses.cs.washington.edu/courses/cse326/01au/lectures/SplayTrees.ppt>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

It is recommended to refer following post as prerequisite of this post.

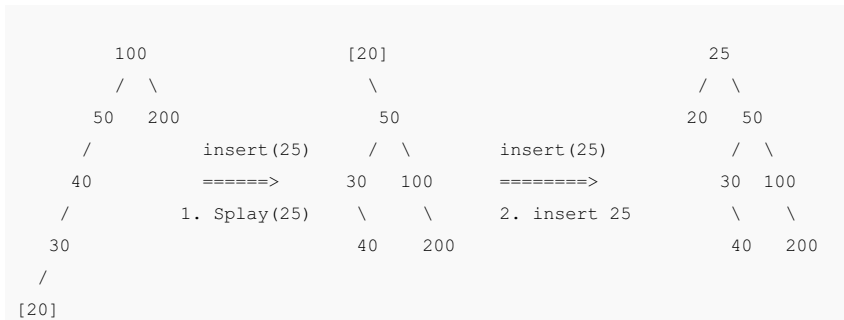
Splay Tree | Set 1 (Search)

As discussed in the [previous post](#), Splay tree is a self-balancing data structure where the last accessed key is always at root. The insert operation is similar to Binary Search Tree insert with additional steps to make sure that the newly inserted key becomes the new root.

Following are different cases to insert a key k in splay tree.

- 1) Root is NULL: We simply allocate a new node and return it as root.
- 2) **Splay** the given key k. If k is already present, then it becomes the new root. If not present, then last accessed leaf node becomes the new root.
- 3) If new root's key is same as k, don't do anything as k is already present.
- 4) Else allocate memory for new node and compare root's key with k.
 -**4.a)** If k is smaller than root's key, make root as right child of new node, copy left child of root as left child of new node and make left child of root as NULL.
 -**4.b)** If k is greater than root's key, make root as left child of new node, copy right child of root as right child of new node and make right child of root as NULL.
- 5) Return new node as new root of tree.

Example:



```
// This code is adopted from http://algs4.cs.princeton.edu/33balanced/:
#include<stdio.h>
#include<stdlib.h>

// An AVL tree node
struct node
{
    int key;
    struct node *left, *right;
};

/* Helper function that allocates a new node with the given key and
```

```

    NULL left and right pointers. */
struct node* newNode(int key)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->key = key;
    node->left = node->right = NULL;
    return (node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct node *rightRotate(struct node *x)
{
    struct node *y = x->left;
    x->left = y->right;
    y->right = x;
    return y;
}

// A utility function to left rotate subtree rooted with x
// See the diagram given above.
struct node *leftRotate(struct node *x)
{
    struct node *y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}

// This function brings the key at root if key is present in tree.
// If key is not present, then it brings the last accessed item at
// root. This function modifies the tree and returns the new root
struct node *splay(struct node *root, int key)
{
    // Base cases: root is NULL or key is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key lies in left subtree
    if (root->key > key)
    {
        // Key is not in tree, we are done
        if (root->left == NULL) return root;

        // Zig-Zig (Left Left)
        if (root->left->key > key)
        {
            // First recursively bring the key as root of left-left
            root->left->left = splay(root->left->left, key);

            // Do first rotation for root, second rotation is done after
            root = rightRotate(root);
        }
        else if (root->left->key < key) // Zig-Zag (Left Right)
        {
            // First recursively bring the key as root of left-right
            root->left->right = splay(root->left->right, key);

            // Do first rotation for root->left
            if (root->left->right != NULL)
                root->left = leftRotate(root->left);
        }
    }
}

```



```

        // Do second rotation for root
        return (root->left == NULL)? root: rightRotate(root);
    }
    else // Key lies in right subtree
    {
        // Key is not in tree, we are done
        if (root->right == NULL) return root;

        // Zig-Zag (Right Left)
        if (root->right->key > key)
        {
            // Bring the key as root of right-left
            root->right->left = splay(root->right->left, key);

            // Do first rotation for root->right
            if (root->right->left != NULL)
                root->right = rightRotate(root->right);
        }
        else if (root->right->key < key) // Zag-Zag (Right Right)
        {
            // Bring the key as root of right-right and do first rotation
            root->right->right = splay(root->right->right, key);
            root = leftRotate(root);
        }

        // Do second rotation for root
        return (root->right == NULL)? root: leftRotate(root);
    }
}

```

```

// Function to insert a new key k in splay tree with given root
struct node *insert(struct node *root, int k)
{
    // Simple Case: If tree is empty
    if (root == NULL) return newNode(k);

    // Bring the closest leaf node to root
    root = splay(root, k);

    // If key is already present, then return
    if (root->key == k) return root;

    // Otherwise allocate memory for new node
    struct node *newnode = newNode(k);

    // If root's key is greater, make root as right child
    // of newnode and copy the left child of root to newnode
    if (root->key > k)
    {
        newnode->right = root;
        newnode->left = root->left;
        root->left = NULL;
    }

    // If root's key is smaller, make root as left child
    // of newnode and copy the right child of root to newnode
    else
    {
        newnode->left = root;
        newnode->right = root->right;
        root->right = NULL;
    }
}

```

```

    }
    return newnode; // newnode becomes new root
}

// A utility function to print preorder traversal of the tree.
// The function also prints height of every node
void preOrder(struct node *root)
{
    if (root != NULL)
    {
        printf("%d ", root->key);
        preOrder(root->left);
        preOrder(root->right);
    }
}

/* Driver program to test above function*/
int main()
{
    struct node *root = newNode(100);
    root->left = newNode(50);
    root->right = newNode(200);
    root->left->left = newNode(40);
    root->left->left->left = newNode(30);
    root->left->left->left->left = newNode(20);
    root = insert(root, 25);
    printf("Preorder traversal of the modified Splay tree is \n");
    preOrder(root);
    return 0;
}

```

Output:

```

Preorder traversal of the modified Splay tree is
25 20 50 30 40 100 200

```

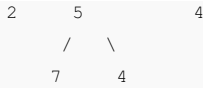
This article is compiled by **Abhay Rath**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

105. Sum of all the numbers that are formed from root to 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

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

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.

```

// C program to find sum of all paths from root to leaves
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node *left, *right;
};

// function to allocate new node with given data
struct node* newNode(int data)
{
    struct node* node = (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// 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);
}

// Driver function to test the above functions
int main()
{
    struct node *root = newNode(6);
    root->left = newNode(3);
    root->right = newNode(5);
    root->right->right = newNode(7);
    root->left->left = newNode(2);
    root->left->right = newNode(5);
    root->right->right = newNode(4);
    root->left->right->left = newNode(7);
    root->left->right->right = newNode(4);
    printf("Sum of all paths is %d", treePathsSum(root));
    return 0;
}

```

Output:

```
Sum of all paths is 13997
```

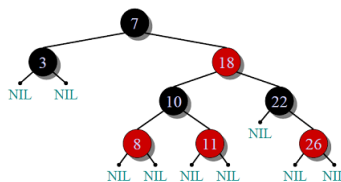
Time Complexity: The above code is a simple preorder traversal code which visits every exactly once. Therefore, the time complexity is $O(n)$ where n is the number of nodes in the given binary tree.

This article is contributed by **Ramchand R**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

106. Red-Black Tree | Set 1 (Introduction)

Red-Black Tree is a self-balancing Binary Search Tree (BST) where every node follows following rules.

- 1) Every node has a color either red or black.
- 2) Root of tree is always black.
- 3) There are no two adjacent red nodes (A red node cannot have a red parent or red child).
- 4) Every path from root to a NULL node has same number of black nodes.



Why Red-Black Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take $O(h)$ time where h is the height of the BST. The cost of these operations may become $O(n)$ for a skewed Binary tree. If we make sure that height of the tree remains $O(\log n)$ after every insertion and deletion, then we can guarantee an upper bound of $O(\log n)$ for all these operations. The height of a Red Black tree is always $O(\log n)$ where n is the number of nodes in the tree.

Comparison with AVL Tree

The AVL trees are more balanced compared to Red Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves many frequent insertions and deletions, then Red Black trees should be preferred. And if the insertions and deletions are less frequent and search is more frequent operation, then AVL tree should be preferred over Red Black Tree.

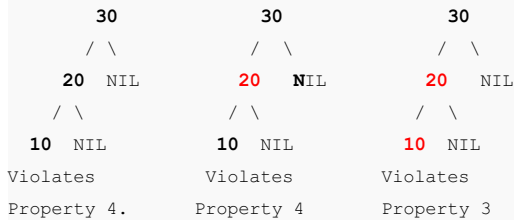
How does a Red-Black Tree ensure balance?

A simple example to understand balancing is, a chain of 3 nodes is not possible in red black tree. We can try any combination of colors and see all of them violate Red-Black

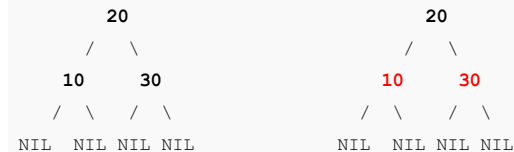
tree property.

A chain of 3 nodes is not possible in Red-Black Trees.

Following are **NOT** Red-Black Trees



Following are different possible Red-Black Trees with above 3 keys



From the above examples, we get some idea how Red-Black trees ensure balance.

Following is an important fact about balancing in Red-Black Trees.

Every Red Black Tree with n nodes has height $\leq 2\log_2(n+1)$

This can be proved using following facts:

- 1) For a general Binary Tree, let k be the minimum number of nodes on all root to NULL paths, then $n \geq 2^k - 1$ (Ex. If k is 3, then n is atleast 7). This expression can also be written as $k \leq 2\log_2(n+1)$
- 2) From property 4 of Red-Black trees and above claim, we can say in a Red-Black Tree with n nodes, there is a root to leaf path with at-most $\log_2(n+1)$ black nodes.
- 3) From property 3 of Red-Black trees, we can claim that the number black nodes in a Red-Black tree is at least $\lfloor n/2 \rfloor$ where n is total number of nodes.

From above 2 points, we can conclude the fact that Red Black Tree with n nodes has height $\leq 2\log_2(n+1)$

In this post, we introduced Red-Black trees and discussed how balance is ensured. The hard part is to maintain balance when keys are added and removed. We will soon be discussing insertion and deletion operations in coming posts on Red-Black tree.

Exercise:

- 1) Is it possible to have all black nodes in a Red-Black tree?
- 2) Draw a Red-Black Tree that is not an AVL tree structure wise?

Insertion and Deletion

Red Black Tree Insertion

Red-Black Tree Deletion

References:

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E.

Leiserson, Ronald L. Rivest

http://en.wikipedia.org/wiki/Red%E2%80%93black_tree

Video Lecture on Red-Black Tree by Tim Roughgarden

MIT Video Lecture on Red-Black Tree

MIT Lecture Notes on Red Black Tree

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

107. Red-Black Tree | Set 2 (Insert)

In the [previous post](#), we discussed introduction to Red-Black Trees. In this post, insertion is discussed.

In [AVL tree insertion](#), we used rotation as a tool to do balancing after insertion caused imbalance. In Red-Black tree, we use two tools to do balancing.

1) Recoloring

2) Rotation

We try recoloring first, if recoloring doesn't work, then we go for rotation. Following is detailed algorithm. The algorithm has mainly two cases depending upon the color of uncle. If uncle is red, we do recoloring. If uncle is black, we do rotations and/or recoloring.

Color of a NULL node is considered as BLACK.

Let x be the newly inserted node.

1) Perform [standard BST insertion](#) and make the color of newly inserted nodes as RED.

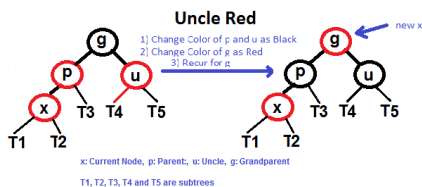
2) Do following if color of x's parent is not BLACK or x is not root.

.....a) If x's uncle is RED (Grand parent must have been black from [property 4](#))

.....(i) Change color of parent and uncle as BLACK.

.....(ii) color of grand parent as RED.

.....(iii) Change x = x's grandparent, repeat steps 2 and 3 for new x.



....b) If x's uncle is **BLACK**, then there can be four configurations for x, x's parent (p) and x's grandparent (g) (This is similar to **AVL Tree**)

.....i) Left Left Case (p is left child of g and x is left child of p)

.....ii) Left Right Case (p is left child of g and x is right child of p)

.....iii) Right Right Case (Mirror of case a)

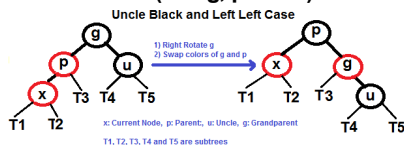
.....iv) Right Left Case (Mirror of case c)

3) If x is root, change color of x as **BLACK** (Black height of complete tree increases by 1).

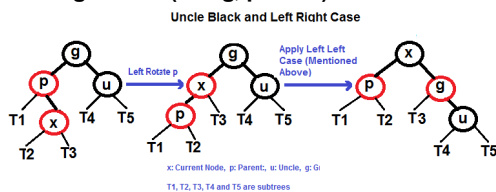
Following are operations to be performed in four subcases when uncle is **BLACK**.

AVL Tree BLACK

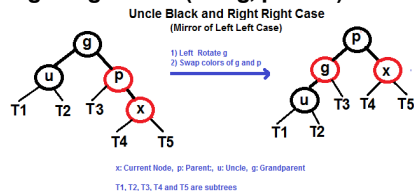
Left Left Case (See g, p and x)



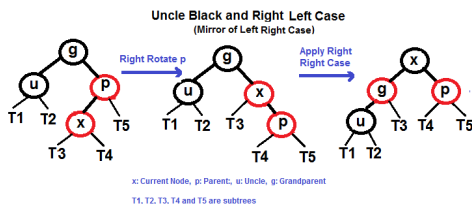
Left Right Case (See g, p and x)



Right Right Case (See g, p and x)

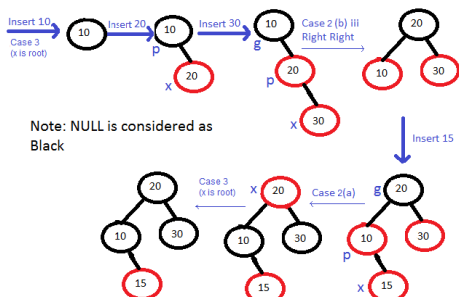


Right Left Case (See g, p and x)



Example

Insert 10, 20, 30 and 15 in an empty tree

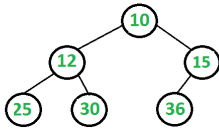


Please refer [C Program for Red Black Tree Insertion](#) for complete implementation of above algorithm.

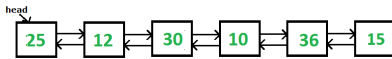
Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

108. Convert a given Binary Tree to Doubly Linked List | Set 3

Given a Binary Tree (BT), convert it to a Doubly Linked List(DLL) In-Place. 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.



The above tree should be in-place converted to following Doubly Linked List(DLL).



Following two different solutions have been discussed for this problem.

[Convert a given Binary Tree to Doubly Linked List | Set 1](#)

[Convert a given Binary Tree to Doubly Linked List | Set 2](#)

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*.

Thanks to rahul, wishall and all other readers for their useful comments on the above two posts.

Following is C++ implementation of this solution.

```

// A C++ program for in-place conversion of Binary Tree to DLL
#include <iostream>
using namespace std;

/* A binary tree node has data, and left and right pointers */
struct node
{
    int data;
    node* left;
    node* right;
};

// 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
        prev->right = root;
    root->left = prev;
    prev = root;

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

```

    {
        root->left = prev;
        prev->right = root;
    }
    prev = root;

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

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
node* newNode(int data)
{
    node* new_node = new node;
    new_node->data = data;
    new_node->left = new_node->right = NULL;
    return (new_node);
}

/* Function to print nodes in a given doubly linked list */
void printList(node *node)
{
    while (node!=NULL)
    {
        cout << node->data << " ";
        node = node->right;
    }
}

/* Driver program to test above functions*/
int main()
{
    // Let us create the tree shown in above diagram
    node *root
        = newNode(10);
    root->left
        = newNode(12);
    root->right
        = newNode(15);
    root->left->left
        = newNode(25);
    root->left->right
        = newNode(30);
    root->right->left
        = newNode(36);

    // Convert to DLL
    node *head = NULL;
    BinaryTree2DoubleLinkedList(root, &head);

    // Print the converted list
    printList(head);

    return 0;
}

```

Output:

```
25 12 30 10 36 15
```

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.

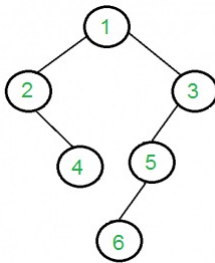
Time Complexity: The above program does a simple inorder traversal, so time complexity is $O(n)$ where n is the number of nodes in given binary tree.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

109. Print all nodes that don't have sibling

Given a Binary Tree, print all nodes that don't have a sibling (a sibling is a node that has same parent. In a Binary Tree, there can be at most one sibling). Root should not be printed as root cannot have a sibling.

For example, the output should be "4 5 6" for the following tree.



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

This is a typical tree traversal question. We start from root and check if the node has one child, if yes then print the only child of that node. If node has both children, then recur for both the children.

```
/* Program to find singles in a given binary tree */
#include <iostream>
using namespace std;

// A Binary Tree Node
struct node
{
    struct node *left, *right;
    int key;
};

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

```

// Function to print all non-root nodes that don't have a sibling
void printSingles(struct node *root)
{
    // Base case
    if (root == NULL)
        return;

    // If this is an internal node, recur for left
    // and right subtrees
    if (root->left != NULL && root->right != NULL)
    {
        printSingles(root->left);
        printSingles(root->right);
    }

    // If left child is NULL and right is not, print right child
    // and recur for right child
    else if (root->right != NULL)
    {
        cout << root->right->key << " ";
        printSingles(root->right);
    }

    // If right child is NULL and left is not, print left child
    // and recur for left child
    else if (root->left != NULL)
    {
        cout << root->left->key << " ";
        printSingles(root->left);
    }
}

```

```

// Driver program to test above functions
int main()
{
    // Let us create binary tree given in the above example
    node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->right = newNode(4);
    root->right->left = newNode(5);
    root->right->left->left = newNode(6);
    printSingles(root);
    return 0;
}

```

Output:

```
4 5 6
```

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

This article is compiled by **Aman Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

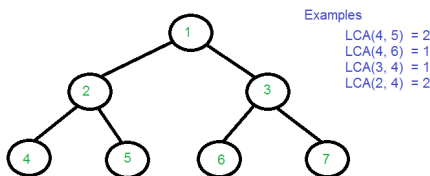
110. Lowest Common Ancestor in a Binary Tree | Set 1

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.

Following is C++ implementation of above algorithm.

```
// A O(n) solution to find LCA of two given values n1 and n2
#include <iostream>
#include <vector>
using namespace std;

// A Binary Tree node
struct Node
{
    int key;
    struct Node *left, *right;
};

// Utility function creates a new binary tree node with given key
```

```

// Utility function creates a new binary tree node with given key
Node * newNode(int k)
{
    Node *temp = new Node;
    temp->key = k;
    temp->left = temp->right = NULL;
    return temp;
}

// Finds the path from root node to given root of the tree, Stores the
// path in a vector path[], returns true if path exists otherwise false
bool findPath(Node *root, vector<int> &path, int k)
{
    // base case
    if (root == NULL) return false;

    // Store this node in path vector. The node will be removed if
    // not in path from root to k
    path.push_back(root->key);

    // See if the k is same as root's key
    if (root->key == k)
        return true;

    // Check if k is found in left or right sub-tree
    if ( (root->left && findPath(root->left, path, k)) ||
        (root->right && findPath(root->right, path, k)) )
        return true;

    // If not present in subtree rooted with root, remove root from
    // path[] and return false
    path.pop_back();
    return false;
}

```

```

// Returns LCA if node n1, n2 are present in the given binary tree,
// otherwise return -1
int findLCA(Node *root, int n1, int n2)
{
    // to store paths to n1 and n2 from the root
    vector<int> path1, path2;

    // Find paths from root to n1 and root to n1. If either n1 or n2
    // is not present, return -1
    if ( !findPath(root, path1, n1) || !findPath(root, path2, n2))
        return -1;

    /* Compare the paths to get the first different value */
    int i;
    for (i = 0; i < path1.size() && i < path2.size() ; i++)
        if (path1[i] != path2[i])
            break;
    return path1[i-1];
}

```

```

// Driver program to test above functions
int main()
{
    // Let us create the Binary Tree shown in above diagram.
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
}

```

```

root->left->right = newNode(5);
root->right->left = newNode(6);
root->right->right = newNode(7);
cout << "LCA(4, 5) = " << findLCA(root, 4, 5);
cout << "\nLCA(4, 6) = " << findLCA(root, 4, 6);
cout << "\nLCA(3, 4) = " << findLCA(root, 3, 4);
cout << "\nLCA(2, 4) = " << findLCA(root, 2, 4);
return 0;
}

```

Output:

```

LCA(4, 5) = 2
LCA(4, 6) = 1
LCA(3, 4) = 1
LCA(2, 4) = 2

```

Time Complexity: Time complexity of the above solution is $O(n)$. The tree is traversed twice, and then path arrays are compared.

Thanks to *Ravi Chandra Enaganti* for suggesting the initial solution based on this method.

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.

```

/* Program to find LCA of n1 and n2 using one traversal of Binary Tree
#include <iostream>
using namespace std;

// A Binary Tree Node
struct Node
{
    struct Node *left, *right;
    int key;
};

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

```


J

```
// 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;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree given in the above example
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    cout << "LCA(4, 5) = " << findLCA(root, 4, 5)->key;
    cout << "\nLCA(4, 6) = " << findLCA(root, 4, 6)->key;
    cout << "\nLCA(3, 4) = " << findLCA(root, 3, 4)->key;
    cout << "\nLCA(2, 4) = " << findLCA(root, 2, 4)->key;
    return 0;
}
```

Output:

```
LCA(4, 5) = 2
LCA(4, 6) = 1
LCA(3, 4) = 1
LCA(2, 4) = 2
```

Thanks to *Atul Singh* for suggesting this solution.

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.

```
/* Program to find LCA of n1 and n2 using one traversal of Binary Tree
   It handles all cases even when n1 or n2 is not there in Binary Tree
#include <iostream>
using namespace std;

// A Binary Tree Node
struct Node
{
    struct Node *left, *right;
    int key;
};

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

// 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,
{
    // 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;
}

// Driver program to test above functions
int main()
{
    // Let us create binary tree given in the above example
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    Node *lca = findLCA(root, 4, 5);
    if (lca != NULL)
        cout << "LCA(4, 5) = " << lca->key;
    else
        cout << "Keys are not present ";

    lca = findLCA(root, 4, 10);
    if (lca != NULL)
        cout << "\nLCA(4, 10) = " << lca->key;
    else
        cout << "\nKeys are not present ";

    return 0;
}

```

Output:

```
LCA(4, 5) = 2
```

```
Keys are not present
```

Thanks to Dhruv for suggesting this extended solution.

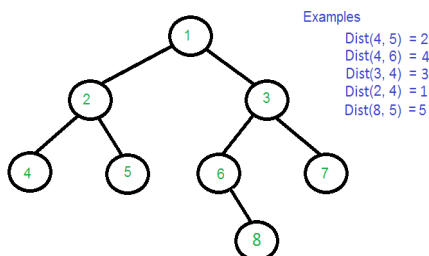
We will soon be discussing more solutions to this problem. Solutions considering the following.

- 1) If there are many LCA queries and we can take some extra preprocessing time to reduce the time taken to find LCA.
- 2) If parent pointer is given with every node.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

111. Find distance between two given keys of a Binary Tree

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.



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

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 C++ implementation of above approach. The implementation is adopted from last code provided in [Lowest Common Ancestor Post](#).

```

/* Program to find distance between n1 and n2 using one traversal */
#include <iostream>
using namespace std;

// A Binary Tree Node
struct Node
{
    struct Node *left, *right;
    int key;
};

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

// Returns level of key k if it is present in tree, otherwise returns
int findLevel(Node *root, int k, int level)
{
    // Base Case
    if (root == NULL)
        return -1;

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

    int l = findLevel(root->left, k, level+1);
    return (l != -1)? l : findLevel(root->right, k, level+1);
}

// 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, 1);
Node *right_lca = findDistUtil(root->right, n1, n2, d1, d2, dist, 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;
}

```

// Driver program to test above functions

```

int main()
{
    // Let us create binary tree given in the above example
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);
    cout << "Dist(4, 5) = " << findDistance(root, 4, 5);
}

```

```

    cout << "\nDist(4, 5) = " << findDistance(root, 4, 5);
    cout << "\nDist(4, 6) = " << findDistance(root, 4, 6);
    cout << "\nDist(3, 4) = " << findDistance(root, 3, 4);
    cout << "\nDist(2, 4) = " << findDistance(root, 2, 4);
    cout << "\nDist(8, 5) = " << findDistance(root, 8, 5);
    return 0;
}

```

Output:

```

Dist(4, 5) = 2
Dist(4, 6) = 4
Dist(3, 4) = 3
Dist(2, 4) = 1
Dist(8, 5) = 5

```

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

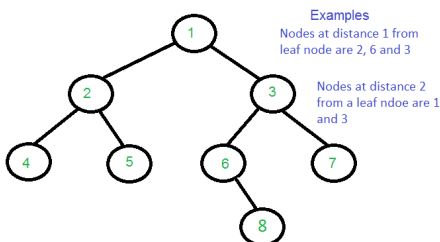
Thanks to **Atul Singh** for providing the initial solution for this post.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

112. Print all nodes that are at distance k from a 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.



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

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[].

```
/* Program to print all nodes which are at distance k from a leaf */
#include <iostream>
using namespace std;
#define MAX_HEIGHT 10000

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

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

/* This function prints all nodes that are distance k from a leaf node
path[] --> Store ancestors of a node
visited[] --> Stores true if a node is printed as output. A node may be
distance away from many leaves, we want to print it only once
void kDistantFromLeafUtil(Node* node, int path[], bool visited[],
                           int pathLen, int k)
{
    // Base case
    if (node==NULL) return;

    /* append this Node to the path array */
    path[pathLen] = node->key;
    visited[pathLen] = false;
    pathLen++;

    /* it's a leaf, so print the ancestor at distance k only
    if the ancestor is not already printed */
    if (node->left == NULL && node->right == NULL &&
        pathLen-k-1 >= 0 && visited[pathLen-k-1] == false)
    {
        cout << path[pathLen-k-1] << " ";
        visited[pathLen-k-1] = true;
        return;
    }

    /* If not leaf node, recur for left and right subtrees */
    kDistantFromLeafUtil(node->left, path, visited, pathLen, k);
    kDistantFromLeafUtil(node->right, path, visited, pathLen, k);
}

/* Given a binary tree and a number k, print all nodes that are k
distant from a leaf*/
void printKDistantfromLeaf(Node* node, int k)
{
    int path[MAX_HEIGHT];
    bool visited[MAX_HEIGHT] = {false};
    kDistantFromLeafUtil(node, path, visited, 0, k);
}
```



```

/* Driver program to test above functions*/
int main()
{
    // Let us create binary tree given in the above example
    Node * root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);

    cout << "Nodes at distance 2 are: ";
    printKDistantfromLeaf(root, 2);

    return 0;
}

```

Output:

```
Nodes at distance 2 are: 3 1
```

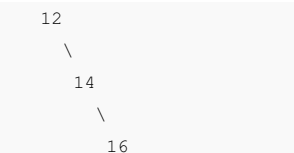
Time Complexity: Time Complexity of above code is $O(n)$ as the code does a simple tree traversal.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

113. Check if a given Binary Tree is height balanced like a 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.

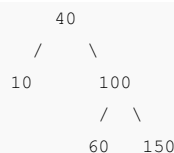
For every node, length of the longest leaf to node path has not more than twice the nodes on shortest path from node to leaf.



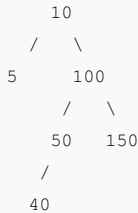
Cannot be a Red-Black Tree
with any color assignment

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.

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

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.

```

/* Program to check if a given Binary Tree is balanced like a Red-Black Tree
#include <iostream>
using namespace std;

```

```

struct Node

```

```

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

```

```

/* utility that allocates a new Node with the given key */
Node* newNode(int key)

```

```

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

```

```

// Returns true 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
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 rmhx, 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 rmhx and rmnh
    if (isBalancedUtil(root->right, rmhx, rmnh) == false)
        return false;

    // Set the max and min heights of this node for the parent call
    maxh = max(lmxh, rmhx) + 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);
}

/* Driver program to test above functions*/
int main()
{
    Node * root = newNode(10);
    root->left = newNode(5);
    root->right = newNode(100);
    root->right->left = newNode(50);
    root->right->right = newNode(150);
    root->right->left->left = newNode(40);
    isBalanced(root)? cout << "Balanced" : cout << "Not Balanced";

    return 0;
}

```

Output:

```
Balanced
```

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

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

114. Interval Tree

Consider a situation where we have a set of intervals and we need following operations to be implemented efficiently.

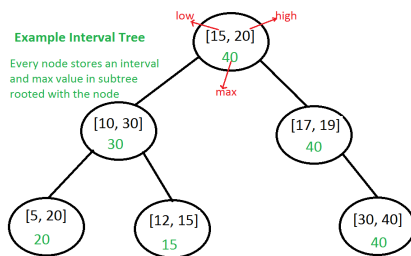
- 1) Add an interval
- 2) Remove an interval
- 3) Given an interval x , find if x overlaps with any of the existing intervals.

Interval Tree: The idea is to augment a self-balancing Binary Search Tree (BST) like **Red Black Tree**, **AVL Tree**, etc to maintain set of intervals so that all operations can be done in $O(\text{Log}n)$ time.

Every node of Interval Tree stores following information.

- a) **i**: An interval which is represented as a pair $[low, high]$
- b) **max**: Maximum *high* value in subtree rooted with this node.

The low value of an interval is used as key to maintain order in BST. The insert and delete operations are same as insert and delete in self-balancing BST used.



The main operation is to search for an overlapping interval. Following is algorithm for searching an overlapping interval x in an Interval tree rooted with *root*.

```
Interval overlappingIntervalSearch(root, x)
```

- 1) If x overlaps with *root*'s interval, return the *root*'s interval.
- 2) If left child of *root* is not empty and the *max* in left child is greater than x 's low value, recur for left child
- 3) Else recur for right child.

How does the above algorithm work?

Let the interval to be searched be x . We need to prove this in for following two cases.

Case 1: When we go to right subtree, one of the following must be true.

- a) There is an overlap in right subtree: This is fine as we need to return one overlapping interval.
- b) There is no overlap in either subtree: We go to right subtree only when either left is NULL or maximum value in left is smaller than $x.low$. So the interval cannot be present in left subtree.

Case 2: When we go to left subtree, one of the following must be true.

a) There is an overlap in left subtree: This is fine as we need to return one overlapping interval.

b) There is no overlap in either subtree: This is the most important part. We need to consider following facts.

... We went to left subtree because $x.\text{low} \leq \text{max}$ in left subtree

.... max in left subtree is a high of one of the intervals let us say $[a, \text{max}]$ in left subtree.

.... Since x doesn't overlap with any node in left subtree $x.\text{low}$ must be smaller than 'a'.

.... All nodes in BST are ordered by low value, so all nodes in right subtree must have low value greater than 'a'.

.... From above two facts, we can say all intervals in right subtree have low value greater than $x.\text{low}$. So x cannot overlap with any interval in right subtree.

Implementation of Interval Tree:

Following is C++ implementation of Interval Tree. The implementation uses basic **insert operation of BST** to keep things simple. Ideally it should be **insertion of AVL Tree** or **insertion of Red-Black Tree**. **Deletion from BST** is left as an exercise.

```
#include <iostream>
using namespace std;

// Structure to represent an interval
struct Interval
{
    int low, high;
};

// Structure to represent a node in Interval Search Tree
struct ITNode
{
    Interval *i; // 'i' could also be a normal variable
    int max;
    ITNode *left, *right;
};

// A utility function to create a new Interval Search Tree Node
ITNode * newNode(Interval i)
{
    ITNode *temp = new ITNode;
    temp->i = new Interval(i);
    temp->max = i.high;
    temp->left = temp->right = NULL;
};

// A utility function to insert a new Interval Search Tree Node
// This is similar to BST Insert. Here the low value of interval
// is used to maintain BST property
ITNode *insert(ITNode *root, Interval i)
{
    // Base case: Tree is empty, new node becomes root
    if (root == NULL)
        return newNode(i);

    // Get low value of interval at root
    int l = root->i->low;

    // If root's low value is smaller, then new interval goes to
    // left subtree
```

```

// Left subtree
if (i.low < l)
    root->left = insert(root->left, i);

// Else, new node goes to right subtree.
else
    root->right = insert(root->right, i);

// Update the max value of this ancestor if needed
if (root->max < i.high)
    root->max = i.high;

return root;
}

// A utility function to check if given two intervals overlap
bool doOverlap(Interval i1, Interval i2)
{
    if (i1.low <= i2.high && i2.low <= i1.high)
        return true;
    return false;
}

// The main function that searches a given interval i in a given
// Interval Tree.
Interval *overlapSearch(ITNode *root, Interval i)
{
    // Base Case, tree is empty
    if (root == NULL) return NULL;

    // If given interval overlaps with root
    if (doOverlap(*(root->i), i))
        return root->i;

    // If left child of root is present and max of left child is
    // greater than or equal to given interval, then i may
    // overlap with an interval in left subtree
    if (root->left != NULL && root->left->max >= i.low)
        return overlapSearch(root->left, i);

    // Else interval can only overlap with right subtree
    return overlapSearch(root->right, i);
}

```

```

void inorder(ITNode *root)
{
    if (root == NULL) return;

    inorder(root->left);

    cout << "[" << root->i->low << ", " << root->i->high << "]"
        << " max = " << root->max << endl;

    inorder(root->right);
}

// Driver program to test above functions
int main()
{
    // Let us create interval tree shown in above figure
    Interval ints[] = {{15, 20}, {10, 30}, {17, 19},
        {5, 20}, {12, 15}, {30, 40}};
};

```

```

int n = sizeof(ints)/sizeof(ints[0]);
ITNode *root = NULL;
for (int i = 0; i < n; i++)
    root = insert(root, ints[i]);

cout << "Inorder traversal of constructed Interval Tree is\n";
inorder(root);

Interval x = {6, 7};

cout << "\nSearching for interval [" << x.low << ", " << x.high <<
Interval *res = overlapSearch(root, x);
if (res == NULL)
    cout << "\nNo Overlapping Interval";
else
    cout << "\nOverlaps with [" << res->low << ", " << res->high <<
return 0;
}

```

Output:

```

Inorder traversal of constructed Interval Tree is
[5, 20] max = 20
[10, 30] max = 30
[12, 15] max = 15
[15, 20] max = 40
[17, 19] max = 40
[30, 40] max = 40

Searching for interval [6,7]
Overlaps with [5, 20]

```

Applications of Interval Tree:

Interval tree is mainly a geometric data structure and often used for windowing queries, for instance, to find all roads on a computerized map inside a rectangular viewport, or to find all visible elements inside a three-dimensional scene (Source [Wiki](#)).

Interval Tree vs Segment Tree

Both segment and interval trees store intervals. Segment tree is mainly optimized for queries for a given point, and interval trees are mainly optimized for overlapping queries for a given interval.

Exercise:

- 1) Implement delete operation for interval tree.
- 2) Extend the intervalSearch() to print all overlapping intervals instead of just one.

http://en.wikipedia.org/wiki/Interval_tree

<http://www.cse.unr.edu/~mgunes/cs302/IntervalTrees.pptx>

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

<https://www.youtube.com/watch?v=dQF0zyaym8A>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

115. Print a Binary Tree in Vertical Order | Set 1

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
```

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

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.

Algorithm:

```
// min --> Minimum horizontal distance from root
// max --> Maximum horizontal distance from root
// hd --> Horizontal distance of current node from root
findMinMax(tree, min, max, hd)
    if tree is NULL then return;
```



```

    if hd is less than min then
        min = hd;
    else if hd is greater than max then
        *max = hd;

    findMinMax(tree->left, min, max, hd-1);
    findMinMax(tree->right, min, max, hd+1);

```

printVerticalLine(tree, line_no, hd)

```

    if tree is NULL then return;

    if hd is equal to line_no, then
        print(tree->data);
    printVerticalLine(tree->left, line_no, hd-1);
    printVerticalLine(tree->right, line_no, hd+1);

```

Implementation:

Following is C++ implementation of above algorithm.

```

#include <iostream>
using namespace std;

// A node of binary tree
struct Node
{
    int data;
    struct Node *left, *right;
};

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

// A utility function to find min and max distances with respect
// to root.
void findMinMax(Node *node, int *min, int *max, int hd)
{
    // Base case
    if (node == NULL) return;

    // Update min and max
    if (hd < *min) *min = hd;
    else if (hd > *max) *max = hd;

    // Recur for left and right subtrees
    findMinMax(node->left, min, max, hd-1);
    findMinMax(node->right, min, max, hd+1);
}

// A utility function to print all nodes on a given line no.

```

```

// A utility function to print all nodes on a given line_no.
// hd is horizontal distance of current node with respect to root.
void printVerticalLine(Node *node, int line_no, int hd)
{
    // Base case
    if (node == NULL) return;

    // If this node is on the given line number
    if (hd == line_no)
        cout << node->data << " ";

    // Recur for left and right subtrees
    printVerticalLine(node->left, line_no, hd-1);
    printVerticalLine(node->right, line_no, hd+1);
}

```

```

// The main function that prints a given binary tree in
// vertical order
void verticalOrder(Node *root)
{
    // Find min and max distances with respect to root
    int min = 0, max = 0;
    findMinMax(root, &min, &max, 0);

    // Iterate through all possible vertical lines starting
    // from the leftmost line and print nodes line by line
    for (int line_no = min; line_no <= max; line_no++)
    {
        printVerticalLine(root, line_no, 0);
        cout << endl;
    }
}

```

```

// Driver program to test above functions
int main()
{
    // Create binary tree shown in above figure
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);
    root->right->right->right = newNode(9);

    cout << "Vertical order traversal is \n";
    verticalOrder(root);

    return 0;
}

```

Output:

```

Vertical order traversal is
4
2
1 5 6
3 8

```

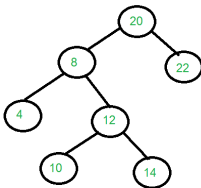
Time Complexity: Time complexity of above algorithm is $O(w \cdot n)$ where w is width of Binary Tree and n is number of nodes in Binary Tree. In worst case, the value of w can be $O(n)$ (consider a complete tree for example) and time complexity can become $O(n^2)$.

This problem can be solved more efficiently using the technique discussed in [this](#) post. We will soon be discussing complete algorithm and implementation of more efficient method.

This article is contributed by **Shalki Agarwal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

116. Print all nodes at distance k from a given node

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"

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

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.

Following is C++ implementation of the above approach.

```
#include <iostream>
using namespace std;

// A binary Tree node
struct node
{
    int data;
    struct node *left, *right;
};

/* 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 d1 = printkdistanceNode(root->left, target, k);
```

```

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 node
    // 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;
}

// A utility function to create a new binary tree node
node *newnode(int data)
{
    node *temp = new node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    /* Let us construct the tree shown in above diagram */
    node * root = newnode(20);
    root->left = newnode(8);
    root->right = newnode(22);
    root->left->left = newnode(4);
    root->left->right = newnode(12);
    root->left->right->left = newnode(10);
    root->left->right->right = newnode(14);
    node * target = root->left->right;
    printkdistanceNode(root, target, 2);
    return 0;
}

```

Output:

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 $O(n)$.

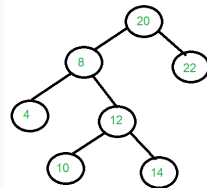
This article is contributed by **Prasant Kumar**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

117. Construct a tree from Inorder and 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

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

The following post can be considered as a prerequisite for this.

Construct Tree from given Inorder and Preorder traversals

Let us consider the above example.

```
in[] = {4, 8, 10, 12, 14, 20, 22};  
level[] = {20, 8, 22, 4, 12, 10, 14};
```

In a Levelorder 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.

```

      /   \
     /     \
    /       \
{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}

```

Following is C++ implementation of the above approach.

```

/* program to construct tree using inorder and levelorder traversals */
#include <iostream>
using namespace std;

/* A binary tree node */
struct Node
{
    int key;
    struct Node* left, *right;
};

/* Function to find index of value in arr[start...end] */
int search(int arr[], int strt, int end, int value)
{
    for (int i = strt; i <= end; i++)
        if (arr[i] == value)
            return i;
    return -1;
}

// n is size of level[], m is size of in[] and m < n. This
// function extracts keys from level[] which are present in
// in[]. The order of extracted keys must be maintained
int *extractKeys(int in[], int level[], int m, int n)
{
    int *newlevel = new int[m], j = 0;
    for (int i = 0; i < n; i++)
        if (search(in, 0, m-1, level[i]) != -1)
            newlevel[j] = level[i], j++;
    return newlevel;
}

```

```

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

/* 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 inSrt 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 inSrt, int inEnd, int n)
{
    // If start index is more than the end index
    if (inSrt > 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 (inSrt == inEnd)
        return root;

    /* Else find the index of this node in Inorder traversal */
    int inIndex = search(in, inSrt, 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, inSrt, inIndex - 1, n);
    root->right = buildTree(in, rlevel, inIndex + 1, inEnd, n);

    // Free memory to avoid memory leak
    delete [] llevel;
    delete [] rlevel;

    return root;
}

/* Utility function to print inorder traversal of binary tree */
void printInorder(Node* node)
{
    if (node == NULL)
        return;
    printInorder(node->left);
    cout << node->key << " ";
    printInorder(node->right);
}

/* Driver program to test above functions */
int main()
{

```



```

int in[] = {4, 8, 10, 12, 14, 20, 22};
int level[] = {20, 8, 22, 4, 12, 10, 14};
int n = sizeof(in)/sizeof(in[0]);
Node *root = buildTree(in, level, 0, n - 1, n);

/* Let us test the built tree by printing Inorder traversal */
cout << "Inorder traversal of the constructed tree is \n";
printInorder(root);

return 0;
}

```

Output:

```

Inorder traversal of the constructed tree is
4 8 10 12 14 20 22

```

An upper bound on time complexity of above method is $O(n^3)$. In the main recursive function, `extractNodes()` is called which takes $O(n^2)$ 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.

This article is contributed by **Abhay Rathi**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

118. Red-Black Tree | Set 3 (Delete)

We have discussed following topics on Red-Black tree in previous posts. We strongly recommend to refer following post as prerequisite of this post.

[Red-Black Tree Introduction](#)

[Red Black Tree Insert](#)

Insertion Vs Deletion:

Like Insertion, recoloring and rotations are used to maintain the Red-Black properties.

In insert operation, we check color of uncle to decide the appropriate case. In delete operation, ***we check color of sibling*** to decide the appropriate case.

The main property that violates after insertion is two consecutive reds. In delete, the main violated property is, change of black height in subtrees as deletion of a black node may cause reduced black height in one root to leaf path.

Deletion is fairly complex process. To understand deletion, notion of double black is used. When a black node is deleted and replaced by a black child, the child is marked as

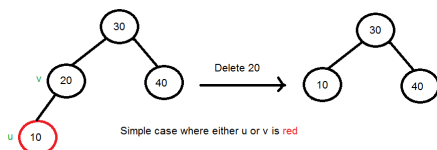
double black. The main task now becomes to convert this double black to single black.

Deletion Steps

Following are detailed steps for deletion.

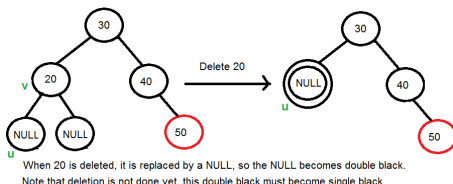
1) Perform **standard BST delete**. When we perform standard delete operation in BST, we always end up deleting a node which is either leaf or has only one child (For an internal node, we copy the successor and then recursively call delete for successor, successor is always a leaf node or a node with one child). So we only need to handle cases where a node is leaf or has one child. Let v be the node deleted and u be the child that replaces v .

2) Simple Case: If either u or v is red, we mark the replaced child as black (No change in black height). Note that both u and v cannot be red as v is parent of u and two consecutive reds are not allowed in red-black tree.



3) If Both u and v are Black.

3.1) Color u as double black. Now our task reduces to convert this double black to single black. Note that if v is leaf, then u is NULL and color of NULL is considered as black. So the deletion of a black leaf also causes a double black.



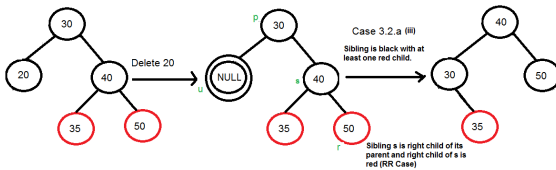
3.2) Do following while the current node u is double black or it is not root. Let sibling of node be s .

....**(a): If sibling s is black and at least one of sibling's children is red**, perform rotation(s). Let the red child of s be r . This case can be divided in four subcases depending upon positions of s and r .

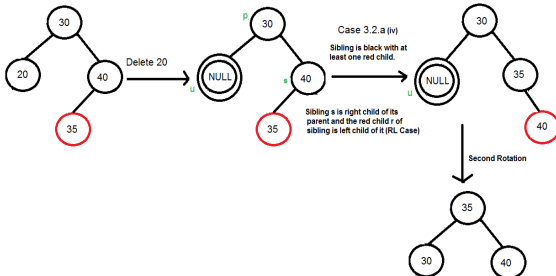
.....**(i) Left Left Case** (s is left child of its parent and r is left child of s or both children of s are red). This is mirror of right right case shown in below diagram.

.....**(ii) Left Right Case** (s is left child of its parent and r is right child). This is mirror of right left case shown in below diagram.

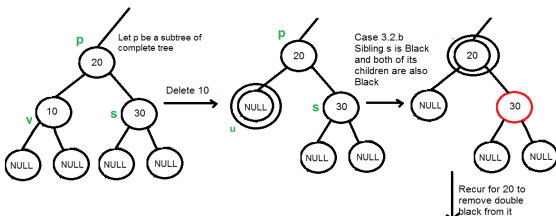
.....**(iii) Right Right Case** (s is right child of its parent and r is right child of s or both children of s are red)



.....(iv) Right Left Case (s is right child of its parent and r is left child of s)



.....(b): If sibling is black and its both children are black, perform recoloring, and recur for the parent if parent is black.

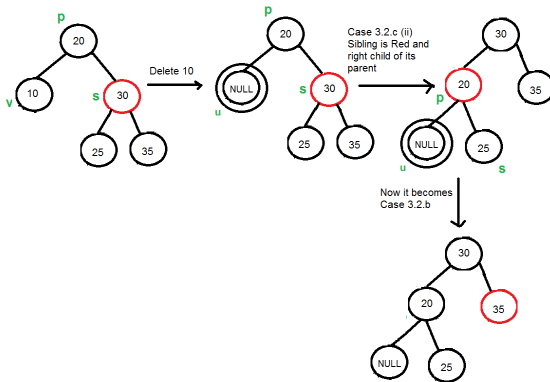


In this case, if parent was red, then we didn't need to recur for parent, we can simply make it black (red + double black = single black)

.....(c): If sibling is red, perform a rotation to move old sibling up, recolor the old sibling and parent. The new sibling is always black (See the below diagram). This mainly converts the tree to black sibling case (by rotation) and leads to case (a) or (b). This case can be divided in two subcases.

.....(i) Left Case (s is left child of its parent). This is mirror of right right case shown in below diagram. We right rotate the parent p.

.....(iii) Right Case (s is right child of its parent). We left rotate the parent p.



3.3) If u is root, make it single black and return (Black height of complete tree reduces by 1).

References:

<https://www.cs.purdue.edu/homes/ayg/CS251/slides/chap13c.pdf>

Introduction to Algorithms 3rd Edition by Clifford Stein, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

119. Print Right View of a Binary Tree

Given a Binary Tree, print Right view of it. Right view of a Binary Tree is set of nodes visible when tree is visited from Right side.

Right view of following tree is 1 3 7 8



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

The Right view contains all nodes that are last nodes in their levels. A simple solution is

to do **level order traversal** and print the last 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. And traverse the tree in a manner that right subtree is visited before left subtree. Whenever we see a node whose level is more than maximum level so far, we print the node because this is the last node in its level (Note that we traverse the right subtree before left subtree). Following is C implementation of this approach.

```

// C program to print right view of Binary Tree
#include<stdio.h>
#include<stdlib.h>

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

// A utility function to create a new Binary Tree Node
struct Node *newNode(int item)
{
    struct Node *temp = (struct Node *)malloc(sizeof(struct Node));
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

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

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

    // Recur for right subtree first, then left subtree
    rightViewUtil(root->right, level+1, max_level);
    rightViewUtil(root->left, level+1, max_level);
}

// A wrapper over rightViewUtil()
void rightView(struct Node *root)
{
    int max_level = 0;
    rightViewUtil(root, 1, &max_level);
}

// Driver Program to test above functions
int main()
{
    struct Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);

    rightView(root);

    return 0;
}

```

Output:

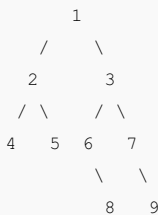
1 3 7 8

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

This article is contributed by **Shalki Agarwal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

120. Print a Binary Tree in Vertical Order | Set 2 (Hashmap based Method)

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
```

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

We have discussed a $O(n^2)$ solution in the [previous post](#). In this post, an efficient solution based on hash map is discussed. 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. For every HD value, we maintain a list of nodes in a hash map. Whenever we see a node in traversal, we go to the hash map entry and add the node to the hash map using HD as a key in map.

Following is C++ implementation of the above method. Thanks to Chirag for providing the below C++ implementation.

```
// C++ program for printing vertical order of a given binary tree
#include <iostream>
#include <vector>
#include <map>
using namespace std;

// Structure for a binary tree node
struct Node
{
    int key;
    Node *left, *right;
};

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

// Utility function to store vertical order in map 'm'
// 'hd' is horizontal distance of current node from root.
// 'hd' is initially passed as 0
void getVerticalOrder(Node* root, int hd, map<int, vector<int>> &m)
{
    // Base case
    if (root == NULL)
        return;

    // Store current node in map 'm'
    m[hd].push_back(root->key);

    // Store nodes in left subtree
    getVerticalOrder(root->left, hd-1, m);

    // Store nodes in right subtree
    getVerticalOrder(root->right, hd+1, m);
}

// The main function to print vertical order of a binary tree
// with given root
void printVerticalOrder(Node* root)
{
    // Create a map and store vertical order in map using
    // function getVerticalOrder()
    map < int, vector<int> > m;
    int hd = 0;
    getVerticalOrder(root, hd, m);
}
```



```

getVerticalOrder(root, hd, m);

// Traverse the map and print nodes at every horizontal
// distance (hd)
map< int,vector<int> > :: iterator it;
for (it=m.begin(); it!=m.end(); it++)
{
    for (int i=0; i<it->second.size(); ++i)
        cout << it->second[i] << " ";
    cout << endl;
}
}

// Driver program to test above functions
int main()
{
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);
    root->right->right->right = newNode(9);
    cout << "Vertical order traversal is \n";
    printVerticalOrder(root);
    return 0;
}

```

Output:

```

Vertical order traversal is
4
2
1 5 6
3 8
7
9

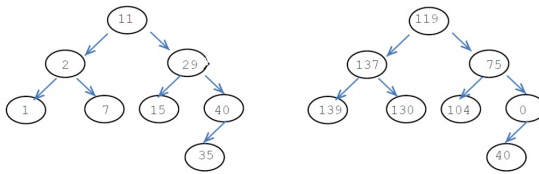
```

Time Complexity of hashing based solution can be considered as $O(n)$ under the assumption that we have good hashing function that allows insertion and retrieval operations in $O(1)$ time. In the above C++ implementation, **map of STL** is used. map in STL is typically implemented using a Self-Balancing Binary Search Tree where all operations take $O(\text{Log}n)$ time. Therefore time complexity of above implementation is $O(n\text{Log}n)$.

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

121. Transform a BST to greater sum tree

Given a BST, transform it into greater sum tree where each node contains sum of all nodes greater than that node.



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

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.

```

// C++ program to transform a BST to sum tree
#include<iostream>
using namespace std;

// A BST node
struct Node
{
    int data;
    struct Node *left, *right;
};

// A utility function to create a new Binary Tree Node
struct Node *newNode(int item)
{
    struct Node *temp = new Node;
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Recursive function to transform a BST to sum tree.
// This function traverses the tree in reverse inorder so
// that we have visited all greater key nodes of the currently
// visited node
void transformTreeUtil(struct Node *root, int *sum)
{

```

```

{
    // Base case
    if (root == NULL) return;

    // Recur for right subtree
    transformTreeUtil(root->right, sum);

    // Update sum
    *sum = *sum + root->data;

    // Store old sum in current node
    root->data = *sum - root->data;

    // Recur for left subtree
    transformTreeUtil(root->left, sum);
}

// A wrapper over transformTreeUtil()
void transformTree(struct Node *root)
{
    int sum = 0; // Initialize sum
    transformTreeUtil(root, &sum);
}

// A utility function to print indorder traversal of a
// binary tree
void printInorder(struct Node *root)
{
    if (root == NULL) return;

    printInorder(root->left);
    cout << root->data << " ";
    printInorder(root->right);
}

// Driver Program to test above functions
int main()
{
    struct Node *root = newNode(11);
    root->left = newNode(2);
    root->right = newNode(29);
    root->left->left = newNode(1);
    root->left->right = newNode(7);
    root->right->left = newNode(15);
    root->right->right = newNode(40);
    root->right->right->left = newNode(35);

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

    transformTree(root);

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

    return 0;
}

```

Output:

```
Inorder Traversal of given tree
```

```
1 2 7 11 15 29 35 40
```

Inorder Traversal of transformed tree

```
139 137 130 119 104 75 40 0
```

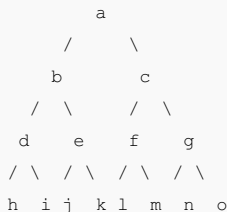
Time complexity of this method is $O(n)$ as it does a simple traversal of tree.

This article is contributed by **Bhavana**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

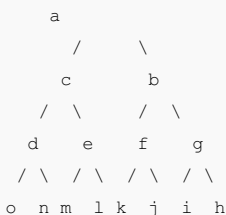
122. Reverse alternate levels of a perfect binary tree

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

Given tree:



Modified tree:



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

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.

A **tricky solution** is to do two inorder traversals. Following are steps to be followed.

- 1) Traverse the given tree in inorder fashion and store all odd level nodes in an auxiliary array. For the above example given tree, contents of array become {h, i, b, j, k, l, m, c, n,

```
o}
```

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

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

Following is C++ implementation of the above algorithm.

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

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

// A utility function to create a new Binary Tree Node
struct Node *newNode(char item)
{
    struct Node *temp = new Node;
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

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

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

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

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

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

    // Update nodes in left subtree
```

```

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

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

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

```

```

// A utility function to reverse an array from index
// 0 to n-1

```

```

void reverse(char arr[], int n)
{
    int l = 0, r = n-1;
    while (l < r)
    {
        int temp = arr[l];
        arr[l] = arr[r];
        arr[r] = temp;
        l++; r--;
    }
}

```

```

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

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

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

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

```

```

// A utility function to print inorder traversal of a
// binary tree

```

```

void printInorder(struct Node *root)
{
    if (root == NULL) return;
    printInorder(root->left);
    cout << root->data << " ";
    printInorder(root->right);
}

```

```

// Driver Program to test above functions

```

```

int main()
{
    struct Node *root = newNode('a');
    root->left = newNode('b');
    root->right = newNode('c');
    root->left->left = newNode('d');
}

```

```

root->left->right = newNode('e');
root->right->left = newNode('f');
root->right->right = newNode('g');
root->left->left->left = newNode('h');
root->left->left->right = newNode('i');
root->left->right->left = newNode('j');
root->left->right->right = newNode('k');
root->right->left->left = newNode('l');
root->right->left->right = newNode('m');
root->right->right->left = newNode('n');
root->right->right->right = newNode('o');

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

reverseAlternate(root);

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

return 0;
}

```

Output:

```

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

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

```

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

This article is contributed by **Kripal Gaurav**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

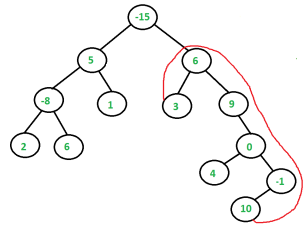
123. Find the maximum path sum between two leaves of a binary tree

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 $O(n)$.

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 $X \rightarrow \text{data}$ 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 $O(n^2)$

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 $X \rightarrow \text{data}$, and compare the sum with maximum path sum found so far.

Following is C++ implementation of the above $O(n)$ solution.

```

// C++ program to find maximum path sum between two leaves of
// a binary tree
#include <iostream>
using namespace std;

// A binary tree node
struct Node
{
    int data;
    struct Node* left, *right;
};

// Utility function to allocate memory for a new node
struct Node* newNode(int data)
{
    struct Node* node = new(struct Node);
    node->data = data;
    node->left = node->right = NULL;
    return (node);
}

// Utility function to find maximum of two integers
int max(int a, int b)
{
    return (a >= b)? a: b;
}

// 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.
int maxPathSumUtil(struct Node *root, int &res)
{

```



```

// Base case
if (root==NULL) return 0;

// Find maximum sum in left and right subtree. Also find
// maximum root to leaf sums in left and right subtrees
// and store them in lLPSum and rLPSum
int lLPSum = maxPathSumUtil(root->left, res);
int rLPSum = maxPathSumUtil(root->right, res);

// Find the maximum path sum passing through root
int curr_sum = max((lLPSum+rLPSum+root->data), max(lLPSum, rLPSum))

// Update res (or result) if needed
if (res < curr_sum)
    res = curr_sum;

// Return the maximum root to leaf path sum
return max(lLPSum, rLPSum)+root->data;
}

// The main function which returns sum of the maximum
// sum path between two leaves. This function mainly uses
// maxPathSumUtil()
int maxPathSum(struct Node *root)
{
    int res = 0;
    maxPathSumUtil(root, res);
    return res;
}

// driver program to test above function
int main()
{
    struct Node *root = newNode(-15);
    root->left = newNode(5);
    root->right = newNode(6);
    root->left->left = newNode(-8);
    root->left->right = newNode(1);
    root->left->left->left = newNode(2);
    root->left->left->right = newNode(6);
    root->right->left = newNode(3);
    root->right->right = newNode(9);
    root->right->right->right= newNode(0);
    root->right->right->right->left= newNode(4);
    root->right->right->right->right= newNode(-1);
    root->right->right->right->right->left= newNode(10);
    cout << "Max pathSum of the given binary tree is " << maxPathSum(r
    return 0;
}

```

Output:

```
Max pathSum of the given binary tree is 27.
```

This article is contributed by **Kripal Gaurav**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

124. Inorder predecessor and successor for a given key in BST

I recently encountered with a question in an interview at e-commerce company. The interviewer asked the following question:

There is BST given with root node with key part as integer only. The structure of each node is as follows:

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

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
```

Following is C++ implementation of the above algorithm:

```
// C++ program to find predecessor and successor in a BST
#include <iostream>
using namespace std;

// BST Node
struct Node
{
    int key;
```

```

    struct Node *left, *right;
};

// 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) ;
    }
}

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

/* A utility function to insert a new node with given key in BST */
Node* insert(Node* node, int key)
{
    if (node == NULL) return newNode(key);
    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);
}

```

```

        node->right = insert(node->right, key);
    return node;
}

// Driver program to test above function
int main()
{
    int key = 65;    //Key to be searched in BST

    /* Let us create following BST
            50
          /  \
        30    70
       /  \  /  \
      20  40 60  80 */
    Node *root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    Node* pre = NULL, *suc = NULL;

    findPreSuc(root, pre, suc, key);
    if (pre != NULL)
        cout << "Predecessor is " << pre->key << endl;
    else
        cout << "No Predecessor";

    if (suc != NULL)
        cout << "Successor is " << suc->key;
    else
        cout << "No Successor";
    return 0;
}

```

Output:

```

Predecessor is 60
Successor is 70

```

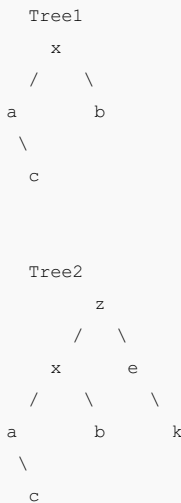
This article is contributed by **algoLover**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

125. Check if a binary tree is subtree of another binary tree | Set 2

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, Tree1 is a subtree of Tree2.



We have discussed a $O(n^2)$ solution for this problem. In this post a $O(n)$ solution is discussed. 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 are 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 of `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}
```

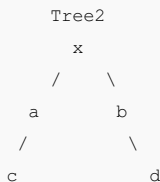
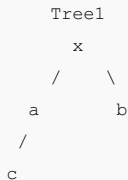
Inorder and Preorder traversals of small tree are

```
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.



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. Thanks to Shivam Goel for suggesting this extension.

Following is C++ implementation of above algorithm.

```
#include <iostream>
#include <cstring>
using namespace std;
#define MAX 100

// Structure of a tree node
struct Node
{
    char key;
    struct Node *left, *right;
};
```

```

};

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

// A utility function to store inorder traversal of tree rooted
// with root in an array arr[]. Note that i is passed as reference
void storeInorder(Node *root, char arr[], int &i)
{
    if (root == NULL)
    {
        arr[i++] = '$';
        return;
    }
    storeInorder(root->left, arr, i);
    arr[i++] = root->key;
    storeInorder(root->right, arr, i);
}

// A utility function to store preorder traversal of tree rooted
// with root in an array arr[]. Note that i is passed as reference
void storePreOrder(Node *root, char arr[], int &i)
{
    if (root == NULL)
    {
        arr[i++] = '$';
        return;
    }
    arr[i++] = root->key;
    storePreOrder(root->left, arr, i);
    storePreOrder(root->right, arr, i);
}

/* 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 preS[] is not a substring of preT[], return false
    if (strstr(preT, preS) == NULL)
        return false;

    return true;
}

```

```

        storePreOrder(i, preS, 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);
    }

// Driver program to test above function
int main()
{
    Node *T = newNode('a');
    T->left = newNode('b');
    T->right = newNode('d');
    T->left->left = newNode('c');
    T->right->right = newNode('e');

    Node *S = newNode('a');
    S->left = newNode('b');
    S->left->left = newNode('c');
    S->right = newNode('d');

    if (isSubtree(T, S))
        cout << "Yes: S is a subtree of T";
    else
        cout << "No: S is NOT a subtree of T";

    return 0;
}

```

Output:

```
No: S is NOT a subtree of T
```

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: $O(n)$

Thanks to **Ashwini Singh** for suggesting this method. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

126. Check if two nodes are cousins in a 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



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.

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

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.

Following is C implementation of the above approach.

```
// C program to check if two Nodes in a binary tree are cousins
#include <stdio.h>
#include <stdlib.h>

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

// A utility function to create a new Binary Tree Node
struct Node *newNode(int item)
{
    struct Node *temp = (struct Node *)malloc(sizeof(struct Node));
    temp->data = item;
    temp->left = temp->right = NULL;
    return temp;
}

// Recursive function to check if two Nodes are siblings
int isSibling(struct Node *root, struct Node *a, struct Node *b)
{
    // Base case
    if (root==NULL) return 0;

    return ((root->left==a && root->right==b)||
            (root->left==b && root->right==a)||
            isSibling(root->left, a, b)||
            isSibling(root->right, a, b));
}

// Recursive function to find level of Node 'ptr' in a binary tree
int level(struct Node *root, struct Node *ptr, int lev)
{
    // base cases
```

```

    if (root == NULL) return 0;
    if (root == ptr) return lev;

    // Return level if Node is present in left subtree
    int l = level(root->left, ptr, lev+1);
    if (l != 0) return l;

    // Else search in right subtree
    return level(root->right, ptr, lev+1);
}

// 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;
}

// Driver Program to test above functions
int main()
{
    struct Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->left->right->right = newNode(15);
    root->right->left = newNode(6);
    root->right->right = newNode(7);
    root->right->left->right = newNode(8);

    struct Node *Node1,*Node2;
    Node1 = root->left->left;
    Node2 = root->right->right;

    isCousin(root,Node1,Node2)? puts("Yes"): puts("No");

    return 0;
}

```

Output:

```
Yes
```

Time Complexity of the above solution is $O(n)$ as it does at most three traversals of binary tree.

This article is contributed by **Ayush Srivastava**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

127. Clone a Binary Tree with 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 tree 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.

```
// A hashmap based C++ program to clone a binary tree with random pointer
#include<iostream>
#include<map>
using namespace std;

/* A binary tree node has data, pointer to left child, a pointer to right
child and a pointer to random node*/
struct Node
{
    int key;
    struct Node* left, *right, *random;
};

/* Helper function that allocates a new Node with the
given data and NULL left, right and random pointers. */
```

```

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

/* Given a binary tree, print its Nodes in inorder*/
void printInorder(Node* node)
{
    if (node == NULL)
        return;

    /* First recur on left subtree */
    printInorder(node->left);

    /* then print data of Node and its random */
    cout << "[" << node->key << " ";
    if (node->random == NULL)
        cout << "NULL], ";
    else
        cout << node->random->key << "], ";

    /* now recur on right subtree */
    printInorder(node->right);
}

// 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;
}

```

```

/* Driver program to test above functions*/
int main()
{
    //Test No 1
    Node *tree = newNode(1);
    tree->left = newNode(2);
    tree->right = newNode(3);
    tree->left->left = newNode(4);
    tree->left->right = newNode(5);
    tree->random = tree->left->right;
    tree->left->left->random = tree;
    tree->left->right->random = tree->right;

    // Test No 2
    // tree = NULL;

    // Test No 3
    // tree = newNode(1);

    // Test No 4
    /* tree = newNode(1);
    tree->left = newNode(2);
    tree->right = newNode(3);
    tree->random = tree->right;
    tree->left->random = tree;
    */

    cout << "Inorder traversal of original binary tree is: \n";
    printInorder(tree);

    Node *clone = cloneTree(tree);

    cout << "\n\nInorder traversal of cloned binary tree is: \n";
    printInorder(clone);

    return 0;
}

```

Output:

```

Inorder traversal of original binary tree is:
[4 1], [2 NULL], [5 3], [1 5], [3 NULL],

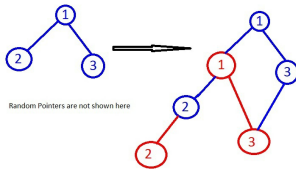
Inorder traversal of cloned binary tree is:
[4 1], [2 NULL], [5 3], [1 5], [3 NULL],

```

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 will 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

Following is C++ implementation of above algorithm.

```
#include <iostream>
using namespace std;

/* A binary tree node has data, pointer to left child, a pointer to right
child and a pointer to random node*/
struct Node
{
    int key;
    struct Node* left, *right, *random;
};

/* Helper function that allocates a new Node with the
given data and NULL left, right and random pointers. */
Node* newNode(int key)
{
    Node* temp = new Node;
    temp->key = key;
    temp->random = temp->right = temp->left = NULL;
    return (temp);
}

/* Given a binary tree, print its Nodes in inorder*/
void printInorder(Node* node)
{
    if (node == NULL)
        return;

    /* First recur on left subtree */
    printInorder(node->left);

    /* then print data of Node and its random */
    cout << "[" << node->key << " ";
    if (node->random == NULL)
        cout << "NULL], ";
    else
        cout << node->random->key << "], ";

    /* now recur on right subtree */
    printInorder(node->right);
}

// This function creates new nodes cloned tree and puts new cloned node
```

```

// 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 will 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 will 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;
}

```

```

/* Driver program to test above functions*/
int main()
{
    /* //Test No 1
    Node *tree = newNode(1);
    tree->left = newNode(2);
    tree->right = newNode(3);
    tree->left->left = newNode(4);
    tree->left->right = newNode(5);
    tree->random = tree->left->right;
    tree->left->left->random = tree;
    tree->left->right->random = tree->right;

    // Test No 2
    // Node *tree = NULL;
    /*
    // Test No 3
    Node *tree = newNode(1);

    // Test No 4
    Node *tree = newNode(1);
    tree->left = newNode(2);
    tree->right = newNode(3);
    tree->random = tree->right;
    tree->left->random = tree;

    Test No 5
    Node *tree = newNode(1);
    tree->left = newNode(2);
    tree->right = newNode(3);
    tree->left->left = newNode(4);
    tree->left->right = newNode(5);
    tree->right->left = newNode(6);
    tree->right->right = newNode(7);
    tree->random = tree->left;
    */
    // Test No 6
    Node *tree = newNode(10);
    Node *n2 = newNode(6);
    Node *n3 = newNode(12);
    Node *n4 = newNode(5);
    Node *n5 = newNode(8);
    Node *n6 = newNode(11);
    Node *n7 = newNode(13);
    Node *n8 = newNode(7);
    Node *n9 = newNode(9);
    tree->left = n2;
    tree->right = n3;
    tree->random = n2;
    n2->left = n4;
    n2->right = n5;
    n2->random = n6;
    n3->left = n7;
    n3->right = n8;
    n3->random = n9;
    n4->left = n5;
    n4->right = n6;
    n4->random = n7;
    n5->left = n6;
    n5->right = n7;
    n5->random = n8;
    n6->left = n7;
    n6->right = n8;
    n6->random = n9;
    n7->left = n8;
    n7->right = n9;
    n7->random = tree;
    n8->left = tree;
    n8->right = n9;
    n8->random = n2;
    n9->left = tree;
    n9->right = n2;
    n9->random = n3;
    */
}

```



```

n2->right = n5;
n2->random = n8;
n3->left = n6;
n3->right = n7;
n3->random = n5;
n4->random = n9;
n5->left = n8;
n5->right = n9;
n5->random = tree;
n6->random = n9;
n9->random = n8;

/* Test No 7
Node *tree = newNode(1);
tree->left = newNode(2);
tree->right = newNode(3);
tree->left->random = tree;
tree->right->random = tree->left;
*/
cout << "Inorder traversal of original binary tree is: \n";
printInorder(tree);

Node *clone = cloneTree(tree);

cout << "\n\nInorder traversal of cloned binary tree is: \n";
printInorder(clone);

return 0;
}

```

Output:

```

Inorder traversal of original binary tree is:
[5 9], [6 7], [7 NULL], [8 10], [9 7], [10 6], [11 9], [12 8], [13 NULL],

Inorder traversal of cloned binary tree is:
[5 9], [6 7], [7 NULL], [8 10], [9 7], [10 6], [11 9], [12 8], [13 NULL],

```

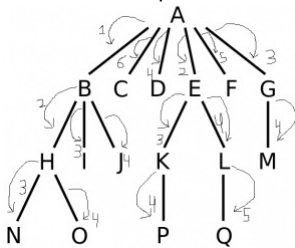
This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

128. Minimum no. of iterations to pass information to all nodes in the 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

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 parent has more children, it will pass info to them in subsequent iterations. Let's say children of a parent takes $c_1, c_2, c_3, c_4, \dots, c_n$ 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 i th iteration, then parent will take $(i + c_i)$ iterations to pass info to child i and all it's subtree.

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)

Nodes with height = 0: (Trivial case) Leaf node has no children (no information passing needed), so no of iterations would be ZERO.

Take a counter initialized with ZERO, loop through all children and keep incrementing counter.

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 c_i and c_j where $c_i > c_j$, 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: $\max(1 + c_j, 2 + c_i) = 2 + c_i$

If parent picks child i 1st then no of iterations needed by parent to pass info to both children and their subtree would be: $\max(1 + c_i, 2 + c_j) = 1 + c_i$ (So picking c_i gives better result than picking c_j)

This tells that parent should always choose child i with max c_i value in any iteration.

SO here greedy approach is:

sort all c_i values decreasing order,

let's say after sorting, values are $c_1 > c_2 > c_3 > \dots > c_n$

take a counter c, set $c = 1 + c_1$ (for child with maximum no of iterations)

for all children i from 2 to n, $c = c + 1 + c_i$

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

Following is C++ implementation of above idea.

```
// C++ program to find minimum number of iterations to pass
// information from root to all nodes in an n-ary tree
#include<iostream>
#include<list>
#include<cmath>
#include <stdlib.h>
using namespace std;

// A class to represent n-ary tree (Note that the implementation
// is similar to graph for simplicity of implementation)
class NaryTree
```

```

class NAryTree
{
    int N;    // No. of nodes in Tree

    // Pointer to an array containing list of children
    list<int> *adj;

    // A function used by getMinIter(), it basically does postorder
    void getMinIterUtil(int v, int minItr[]);
public:
    NAryTree(int N);    // Constructor

    // function to add a child w to v
    void addChild(int v, int w);

    // The main function to find minimum iterations
    int getMinIter();

    static int compare(const void * a, const void * b);
};

NAryTree::NAryTree(int N)
{
    this->N = N;
    adj = new list<int>[N];
}

// To add a child w to v
void NAryTree::addChild(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
}

/* A recursive function to used by getMinIter(). This function
// mainly does postorder traversal and get minimum iteration of all ch
// of node u, sort them in decreasing order and then get minimum itera
// of node u

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) -->> child(A) is children count of node 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
*/

void NAryTree::getMinIterUtil(int u, int minItr[])
{
    minItr[u] = adj[u].size();
    int *minItrTemp = new int[minItr[u]];
    int k = 0, tmp = 0;
    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[u].begin(); i != adj[u].end(); ++i)
    {
        getMinIterUtil(*i, minItr);
        minItrTemp[k++] = minItr[*i];
    }
    qsort(minItrTemp, minItr[u], sizeof (int), compare);
}

```

```

        for (k = 0; k < adj[u].size(); k++)
        {
            tmp = minItrTemp[k] + k + 1;
            minItr[u] = max(minItr[u], tmp);
        }
        delete[] minItrTemp;
    }

// The function to do PostOrder traversal. It uses
// recursive getMinIterUtil()
int NArYTree::getMinIter()
{
    // Set minimum iteration all the vertices as zero
    int *minItr = new int[N];
    int res = -1;
    for (int i = 0; i < N; i++)
        minItr[i] = 0;

    // Start Post Order Traversal from Root
    getMinIterUtil(0, minItr);
    res = minItr[0];
    delete[] minItr;
    return res;
}

```

```

int NArYTree::compare(const void * a, const void * b)
{
    return ( *(int*)b - *(int*)a );
}

```

```

// Driver function to test above functions
int main()
{

```

```

    // TestCase 1
    NArYTree tree1(17);
    tree1.addChild(0, 1);
    tree1.addChild(0, 2);
    tree1.addChild(0, 3);
    tree1.addChild(0, 4);
    tree1.addChild(0, 5);
    tree1.addChild(0, 6);

    tree1.addChild(1, 7);
    tree1.addChild(1, 8);
    tree1.addChild(1, 9);

    tree1.addChild(4, 10);
    tree1.addChild(4, 11);

    tree1.addChild(6, 12);

    tree1.addChild(7, 13);
    tree1.addChild(7, 14);
    tree1.addChild(10, 15);
    tree1.addChild(11, 16);

```

```

    cout << "TestCase 1 - Minimum Iteration: "
         << tree1.getMinIter() << endl;

```

```

// TestCase 2
NArYTree tree2(3);
tree2.addChild(0, 1);

```

```

tree2.addChild(0, 2);
cout << "TestCase 2 - Minimum Iteration: "
      << tree2.getMinIter() << endl;

// TestCase 3
NAryTree tree3(1);
cout << "TestCase 3 - Minimum Iteration: "
      << tree3.getMinIter() << endl;

// TestCase 4
NAryTree tree4(6);
tree4.addChild(0, 1);
tree4.addChild(1, 2);
tree4.addChild(2, 3);
tree4.addChild(3, 4);
tree4.addChild(4, 5);
cout << "TestCase 4 - Minimum Iteration: "
      << tree4.getMinIter() << endl;

// TestCase 5
NAryTree tree5(6);
tree5.addChild(0, 1);
tree5.addChild(0, 2);
tree5.addChild(2, 3);
tree5.addChild(2, 4);
tree5.addChild(2, 5);
cout << "TestCase 5 - Minimum Iteration: "
      << tree5.getMinIter() << endl;

// TestCase 6
NAryTree tree6(6);
tree6.addChild(0, 1);
tree6.addChild(0, 2);
tree6.addChild(2, 3);
tree6.addChild(2, 4);
tree6.addChild(3, 5);
cout << "TestCase 6 - Minimum Iteration: "
      << tree6.getMinIter() << endl;

// TestCase 7
NAryTree tree7(14);
tree7.addChild(0, 1);
tree7.addChild(0, 2);
tree7.addChild(0, 3);
tree7.addChild(1, 4);
tree7.addChild(2, 5);
tree7.addChild(2, 6);
tree7.addChild(4, 7);
tree7.addChild(5, 8);
tree7.addChild(5, 9);
tree7.addChild(7, 10);
tree7.addChild(8, 11);
tree7.addChild(8, 12);
tree7.addChild(10, 13);
cout << "TestCase 7 - Minimum Iteration: "
      << tree7.getMinIter() << endl;

// TestCase 8
NAryTree tree8(14);
tree8.addChild(0, 1);
tree8.addChild(0, 2);
tree8.addChild(0, 3);
tree8.addChild(0, 4);

```

```

tree8.addChild(0, 4);
tree8.addChild(0, 5);
tree8.addChild(1, 6);
tree8.addChild(2, 7);
tree8.addChild(3, 8);
tree8.addChild(4, 9);
tree8.addChild(6, 10);
tree8.addChild(7, 11);
tree8.addChild(8, 12);
tree8.addChild(9, 13);
cout << "TestCase 8 - Minimum Iteration: "
    << tree8.getMinIter() << endl;

```

// TestCase 9

```

NAryTree tree9(25);
tree9.addChild(0, 1);
tree9.addChild(0, 2);
tree9.addChild(0, 3);
tree9.addChild(0, 4);
tree9.addChild(0, 5);
tree9.addChild(0, 6);

tree9.addChild(1, 7);
tree9.addChild(2, 8);
tree9.addChild(3, 9);
tree9.addChild(4, 10);
tree9.addChild(5, 11);
tree9.addChild(6, 12);

tree9.addChild(7, 13);
tree9.addChild(8, 14);
tree9.addChild(9, 15);
tree9.addChild(10, 16);
tree9.addChild(11, 17);
tree9.addChild(12, 18);

tree9.addChild(13, 19);
tree9.addChild(14, 20);
tree9.addChild(15, 21);
tree9.addChild(16, 22);
tree9.addChild(17, 23);
tree9.addChild(19, 24);

```

```

cout << "TestCase 9 - Minimum Iteration: "
    << tree9.getMinIter() << endl;

```

return 0;

}

Output:

```

TestCase 1 - Minimum Iteration: 6
TestCase 2 - Minimum Iteration: 2
TestCase 3 - Minimum Iteration: 0
TestCase 4 - Minimum Iteration: 5
TestCase 5 - Minimum Iteration: 4
TestCase 6 - Minimum Iteration: 3
TestCase 7 - Minimum Iteration: 6
TestCase 8 - Minimum Iteration: 6

```

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

129. Find Height of Binary Tree represented by Parent array

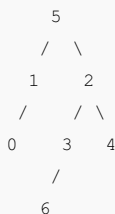
A given array represents a tree in such a way that the array value gives the parent node of that particular index. The value of the root node index would always be -1. Find the height of the tree.

Height of a Binary Tree is number of nodes on the path from root to the deepest leaf node, the number includes both root and leaf.

Input: parent[] = {1 5 5 2 2 -1 3}

Output: 4

The given array represents following Binary Tree



Input: parent[] = {-1, 0, 0, 1, 1, 3, 5};

Output: 5

The given array represents following Binary Tree



Source: [Amazon Interview experience | Set 128 \(For SDET\)](#)

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

A **simple solution** is to first construct the tree and then find height of the constructed binary tree. The tree can be constructed recursively by first searching the current root, then recurring for the found indexes and making them left and right subtrees of root. This solution takes $O(n^2)$ 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 first calculate depth of every node and store in an array `depth[]`. Once we have depths of all nodes, we return maximum of all depths.

- 1) Find depth of all nodes and fill in an auxiliary array `depth[]`.
- 2) Return maximum value in `depth[]`.

Following are steps to find depth of a node at index `i`.

- 1) If it is root, `depth[i]` is 1.
- 2) If depth of `parent[i]` is evaluated, `depth[i]` is `depth[parent[i]] + 1`.
- 3) If depth of `parent[i]` is not evaluated, recur for parent and assign `depth[i]` as `depth[parent[i]] + 1` (same as above).

Following is C++ implementation of above idea.

```
// C++ program to find height using parent array
#include <iostream>
using namespace std;

// This function fills depth of i'th element in parent[]. The depth is
// filled in depth[i].
void fillDepth(int parent[], int i, int depth[])
{
    // If depth[i] is already filled
    if (depth[i])
        return;

    // If node at index i is root
    if (parent[i] == -1)
    {
        depth[i] = 1;
        return;
    }

    // If depth of parent is not evaluated before, then evaluate
    // depth of parent first
    if (depth[parent[i]] == 0)
        fillDepth(parent, parent[i], depth);

    // Depth of this node is depth of parent plus 1
    depth[i] = depth[parent[i]] + 1;
}

// This function returns height of binary tree represented by
// parent array
int findHeight(int parent[], int n)
{
    // Create an array to store depth of all nodes/ and
    // initialize depth of every node as 0 (an invalid
    // value). Depth of root is 1
    int depth[n];
    for (int i = 0; i < n; i++)
        depth[i] = 0;

    // Find depth of all nodes
    for (int i = 0; i < n; i++)
        fillDepth(parent, i, depth);

    // Return maximum depth
    int maxHeight = 0;
    for (int i = 0; i < n; i++)
        if (depth[i] > maxHeight)
            maxHeight = depth[i];

    return maxHeight;
}
```

```

        depth[i] = 0;

    // fill depth of all nodes
    for (int i = 0; i < n; i++)
        fillDepth(parent, i, depth);

    // The height of binary tree is maximum of all depths.
    // Find the maximum value in depth[] and assign it to ht.
    int ht = depth[0];
    for (int i=1; i<n; i++)
        if (ht < depth[i])
            ht = depth[i];
    return ht;
}

// Driver program to test above functions
int main()
{
    // int parent[] = {1, 5, 5, 2, 2, -1, 3};
    int parent[] = {-1, 0, 0, 1, 1, 3, 5};

    int n = sizeof(parent)/sizeof(parent[0]);
    cout << "Height is " << findHeight(parent, n);
    return 0;
}

```

Output:

```
Height is 5
```

Note that the time complexity of this programs seems more than $O(n)$. If we take a closer look, we can observe that depth of every node is evaluated only once.

This article is contributed by **Siddharth**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

130. Print nodes between two given level numbers of a binary tree

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.

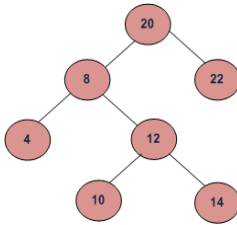
Input: Root of below tree, low = 2, high = 4

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 $O(n^2)$

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.

The following is C++ implementation of above idea.

```
// A C++ program to print Nodes level by level between given two level:
#include <iostream>
#include <queue>
using namespace std;
```

```
/* A binary tree Node has key, pointer to left and right children */
struct Node
{
    int key;
    struct Node* left, *right;
};
```

```
/* Given a binary tree, print nodes from level number 'low' to level
   number 'high'*/
void printLevels(Node* root, int low, int high)
{
    queue <Node *> Q;

    Node *marker = new Node; // Marker node to indicate end of level

    int level = 1; // Initialize level number

    // Enqueue the only first level node and marker node for end of level
    Q.push(root);
    Q.push(marker);

    // Simple level order traversal loop
    while (Q.empty() == false)
    {
        // Remove the front item from queue
        Node *n = Q.front();
        Q.pop();

        // Check if end of level is reached
        if (n == marker)
        {
            // print a new line and increment level number
            cout << endl;
            level++;
        }
    }
}
```

```

        level++;

        // Check if marker node was last node in queue or
        // level number is beyond the given upper limit
        if (Q.empty() == true || level > high) break;

        // Enqueue the marker for end of next level
        Q.push(marker);

        // If this is marker, then we don't need print it
        // and enqueue its children
        continue;
    }

    // If level is equal to or greater than given lower level,
    // print it
    if (level >= low)
        cout << n->key << " ";

    // Enqueue children of non-marker node
    if (n->left != NULL) Q.push(n->left);
    if (n->right != NULL) Q.push(n->right);
}
}

```

/* Helper function that allocates a new Node with the given key and NULL left and right pointers. */

```

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

```

/* Driver program to test above functions*/

```

int main()
{
    // Let us construct the BST shown in the above figure
    struct Node *root = newNode(20);
    root->left = newNode(8);
    root->right = newNode(22);
    root->left->left = newNode(4);
    root->left->right = newNode(12);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(14);

    cout << "Level Order traversal between given two levels is";
    printLevels(root, 2, 3);

    return 0;
}

```

Level Order traversal between given two levels is

8 22

4 12

Time complexity of above method is $O(n)$ as it does a simple level order traversal.

This article is contributed by **Frank**. Please write comments if you find anything

incorrect, or you want to share more information about the topic discussed above

131. Serialize and Deserialize a Binary Tree

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

Following are some simpler versions of the problem:

If given 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

Input:

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

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

Input:

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

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

Input:

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

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

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

Following is C++ implementation of the above idea.

```
// A C++ program to demonstrate serialization and deserialiation of
// Binary Tree
#include <stdio.h>
#define MARKER -1

/* A binary tree Node has key, pointer to left and right children */
struct Node
{
    int key;
    struct Node* left, *right;
};

/* Helper function that allocates a new Node with the
given kev and NULL left and right pointers. */
```

```

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

// 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 there 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);
}

// A simple inorder traversal used for testing the constructed tree
void inorder(Node *root)
{
    if (root)
    {
        inorder(root->left);
        printf("%d ", root->key);
        inorder(root->right);
    }
}

/* Driver program to test above functions*/
int main()
{
    // Let us construct a tree shown in the above figure
    struct Node *root = newNode(20);
    root->left = newNode(8);
    root->right = newNode(22);
    root->left->left = newNode(4);
    root->left->right = newNode(12);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(14);
}

```

```

// Let us open a file and serialize the tree into the file
FILE *fp = fopen("tree.txt", "w");
if (fp == NULL)
{
    puts("Could not open file");
    return 0;
}
serialize(root, fp);
fclose(fp);

// Let us deserialize the stored tree into root1
Node *root1 = NULL;
fp = fopen("tree.txt", "r");
deserialize(root1, fp);

printf("Inorder Traversal of the tree constructed from file:\n");
inorder(root1);

return 0;
}

```

Output:

```

Inorder Traversal of the tree constructed from file:
4 8 10 12 14 20 22

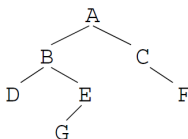
```

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 observe 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](#).



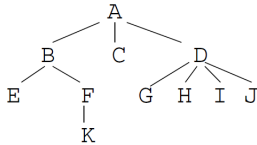
A ' B ' D E ' G / C ' / F

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](#).



ABE) FK))) C) DG) H) I) J)))

References:

<http://www.cs.usfca.edu/~brooks/S04classes/cs245/lectures/lecture11.pdf>

This article is contributed by **Shivam Gupta**, Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

132. Serialize and Deserialize an N-ary Tree

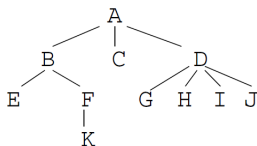
Given an N-ary tree where every node has at-most N children. How to serialize and deserialize it? Serialization is to store tree in a file so that it can be later restored. The structure of tree must be maintained. Deserialization is reading tree back from file.

This post is mainly an extension of below post.

[Serialize and Deserialize a Binary Tree](#)

In an N-ary tree, there are no designated left and right children. An N-ary tree is represented by storing an array or list of child pointers with every node.

The idea is to store an 'end of children' marker with every node. The following diagram shows serialization where ')' is used as end of children marker. The diagram is taken from [here](#).



ABE) FK))) C) DG) H) I) J)))

Following is C++ implementation of above idea.

```

// A C++ Program serialize and deserialize an N-ary tree
#include<cstdio>
#define MARKER ')'
#define N 5
using namespace std;

// A node of N-ary tree
  
```

```

struct Node {
    char key;
    Node *child[N]; // An array of pointers for N children
};

// A utility function to create a new N-ary tree node
Node *newNode(char key)
{
    Node *temp = new Node;
    temp->key = key;
    for (int i = 0; i < N; i++)
        temp->child[i] = NULL;
    return temp;
}

// This function stores the given N-ary tree in a file pointed by fp
void serialize(Node *root, FILE *fp)
{
    // Base case
    if (root == NULL) return;

    // Else, store current node and recur for its children
    fprintf(fp, "%c ", root->key);
    for (int i = 0; i < N && root->child[i]; i++)
        serialize(root->child[i], fp);

    // Store marker at the end of children
    fprintf(fp, "%c ", MARKER);
}

// This function constructs N-ary tree from a file pointed by 'fp'.
// This function returns 0 to indicate that the next item is a valid
// tree key. Else returns 0
int deSerialize(Node *&root, FILE *fp)
{
    // Read next item from file. If there are no more items or next
    // item is marker, then return 1 to indicate same
    char val;
    if ( !fscanf(fp, "%c ", &val) || val == MARKER )
        return 1;

    // Else create node with this item and recur for children
    root = newNode(val);
    for (int i = 0; i < N; i++)
        if (deSerialize(root->child[i], fp))
            break;

    // Finally return 0 for successful finish
    return 0;
}

// A utility function to create a dummy tree shown in above diagram
Node *createDummyTree()
{
    Node *root = newNode('A');
    root->child[0] = newNode('B');
    root->child[1] = newNode('C');
    root->child[2] = newNode('D');
    root->child[0]->child[0] = newNode('E');
    root->child[0]->child[1] = newNode('F');
    root->child[2]->child[0] = newNode('G');
    root->child[2]->child[1] = newNode('H');
}

```

```

    root->child[2]->child[2] = newNode('I');
    root->child[2]->child[3] = newNode('J');
    root->child[0]->child[1]->child[0] = newNode('K');
    return root;
}

// A utility function to traverse the constructed N-ary tree
void traverse(Node *root)
{
    if (root)
    {
        printf("%c ", root->key);
        for (int i = 0; i < N; i++)
            traverse(root->child[i]);
    }
}

// Driver program to test above functions
int main()
{
    // Let us create an N-ary tree shown in above diagram
    Node *root = createDummyTree();

    // Let us open a file and serialize the tree into the file
    FILE *fp = fopen("tree.txt", "w");
    if (fp == NULL)
    {
        puts("Could not open file");
        return 0;
    }
    serialize(root, fp);
    fclose(fp);

    // Let us deserialize the stored tree into root1
    Node *root1 = NULL;
    fp = fopen("tree.txt", "r");
    deserialize(root1, fp);

    printf("Constructed N-Ary Tree from file is \n");
    traverse(root1);

    return 0;
}

```

Output:

```

Constructed N-Ary Tree from file is
A B E F K C D G H I J

```

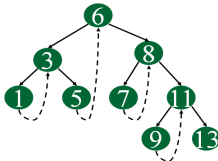
The above implementation can be optimized in many ways for example by using a vector in place of array of pointers. We have kept it this way to keep it simple to read and understand.

This article is contributed by **varun**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

133. Convert a Binary Tree to 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 queue and make it the right of current node. We also set isThreaded to true to indicate that the right pointer is a threaded link.

Following is C++ implementation of the above idea.

```
/* C++ program to convert a Binary Tree to Threaded Tree */
#include <iostream>
#include <queue>
using namespace std;

/* Structure of a node in 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;
};
```

```
};
```

```
// Helper function to put the Nodes in inorder into queue
void populateQueue(Node *root, std::queue <Node *> *q)
{
    if (root == NULL) return;
    if (root->left)
        populateQueue(root->left, q);
    q->push(root);
    if (root->right)
        populateQueue(root->right, q);
}

// Function to traverse queue, and make tree threaded
void createThreadedUtil(Node *root, std::queue <Node *> *q)
{
    if (root == NULL) return;

    if (root->left)
        createThreadedUtil(root->left, q);
    q->pop();

    if (root->right)
        createThreadedUtil(root->right, q);

    // If right pointer is NULL, link it to the
    // inorder successor and set 'isThreaded' bit.
    else
    {
        root->right = q->front();
        root->isThreaded = true;
    }
}

// This function uses populateQueue() and
// createThreadedUtil() to convert a given binary tree
// to threaded tree.
void createThreaded(Node *root)
{
    // Create a queue to store inorder traversal
    std::queue <Node *> q;

    // Store inorder traversal in queue
    populateQueue(root, &q);

    // Link NULL right pointers to inorder successor
    createThreadedUtil(root, &q);
}
```

```
// A utility function to find leftmost node in a binary
// tree rooted with 'root'. This function is used in inOrder()
Node *leftMost(Node *root)
{
    while (root != NULL && root->left != NULL)
        root = root->left;
    return root;
}
```

```
// Function to do inorder traversal of a threaded binary tree
void inOrder(Node *root)
{
    if (root == NULL) return;
```

```

// Find the leftmost node in Binary Tree
Node *cur = leftMost(root);

while (cur != NULL)
{
    cout << cur->key << " ";

    // If this Node is a thread Node, then go to
    // inorder successor
    if (cur->isThreaded)
        cur = cur->right;

    else // Else go to the leftmost child in right subtree
        cur = leftMost(cur->right);
}

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

// Driver program to test above functions
int main()
{
    /*
        1
       /\
      2  3
     /\ /\
    4 5 6 7  */
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->left = newNode(6);
    root->right->right = newNode(7);

    createThreaded(root);

    cout << "Inorder traversal of creeated threaded tree is\n";
    inOrder(root);
    return 0;
}

```

Output:

```

Inorder traversal of creeated threaded tree is
4 2 5 1 6 3 7

```

This article is contributed by **Minhaz**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

134. K Dimensional Tree

A K-D Tree(also called as K-Dimensional Tree) is a binary search tree where data in each node is a K-Dimensional point in space. In short, it is a space partitioning(details below) data structure for organizing points in a K-Dimensional space.

A non-leaf node in K-D tree divides the space into two parts, called as half-spaces.

Points to the left of this space are represented by the left subtree of that node and points to the right of the space are represented by the right subtree. We will soon be explaining the concept on how the space is divided and tree is formed.

For the sake of simplicity, let us understand a 2-D Tree with an example.

The root would have an x-aligned plane, the root's children would both have y-aligned planes, the root's grandchildren would all have x-aligned planes, and the root's great-grandchildren would all have y-aligned planes and so on.

Generalization:

Let us number the planes as 0, 1, 2, ...($K - 1$). From the above example, it is quite clear that a point (node) at depth D will have A aligned plane where A is calculated as:

$$A = D \bmod K$$

How to determine if a point will lie in the left subtree or in right subtree?

If the root node is aligned in planeA, then the left subtree will contain all points whose coordinates in that plane are smaller than that of root node. Similarly, the right subtree will contain all points whose coordinates in that plane are greater-equal to that of root node.

Creation of a 2-D Tree:

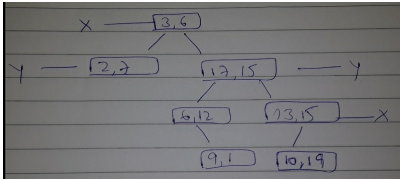
Consider following points in a 2-D plane:

(3, 6), (17, 15), (13, 15), (6, 12), (9, 1), (2, 7), (10, 19)

1. Insert (3, 6): Since tree is empty, make it the root node.
2. Insert (17, 15): Compare it with root node point. Since root node is X-aligned, the X-coordinate value will be compared to determine if it lies in the rightsubtree or in the left subtree. This point will be Y-aligned.
3. Insert (13, 15): X-value of this point is greater than X-value of point in root node. So, this will lie in the right subtree of (3, 6). Again Compare Y-value of this point with the Y-value of point (17, 15) (Why?). Since, they are equal, this point will lie in the right subtree of (17, 15). This point will be X-aligned.
4. Insert (6, 12): X-value of this point is greater than X-value of point in root node. So, this will lie in the right subtree of (3, 6). Again Compare Y-value of this point with the Y-value of point (17, 15) (Why?). Since, $12 < 15$, this point will lie in the left subtree

of (17, 15). This point will be X-aligned.

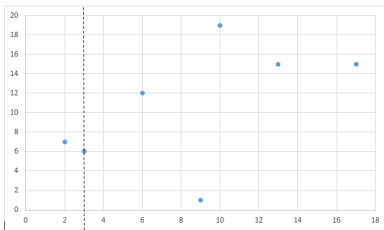
5. Insert (9, 1): Similarly, this point will lie in the right of (6, 12).
6. Insert (2, 7): Similarly, this point will lie in the left of (3, 6).
7. Insert (10, 19): Similarly, this point will lie in the left of (13, 15).



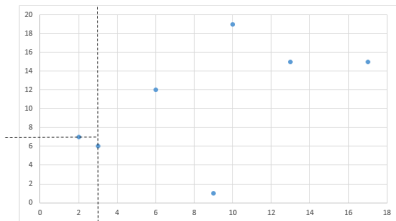
How is space partitioned?

All 7 points will be plotted in the X-Y plane as follows:

1. Point (3, 6) will divide the space into two parts: Draw line $X = 3$.

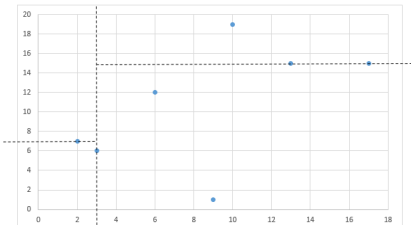


2. Point (2, 7) will divide the space to the left of line $X = 3$ into two parts horizontally. Draw line $Y = 7$ to the left of line $X = 3$.



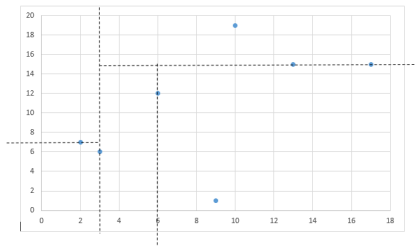
3. Point (17, 15) will divide the space to the right of line $X = 3$ into two parts horizontally.

Draw line $Y = 15$ to the right of line $X = 3$.



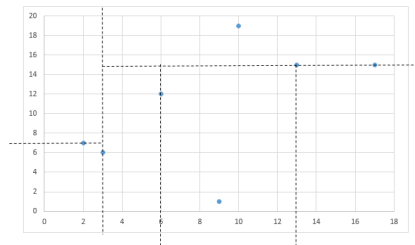
- Point (6, 12) will divide the space below line $Y = 15$ and to the right of line $X = 3$ into two parts.

Draw line $X = 6$ to the right of line $X = 3$ and below line $Y = 15$.



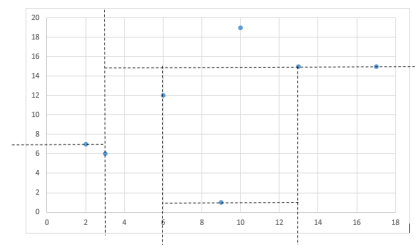
- Point (13, 15) will divide the space below line $Y = 15$ and to the right of line $X = 6$ into two parts.

Draw line $X = 13$ to the right of line $X = 6$ and below line $Y = 15$.



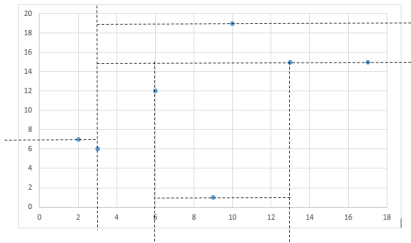
- Point (9, 1) will divide the space between lines $X = 3$, $X = 6$ and $Y = 15$ into two parts.

Draw line $Y = 1$ between lines $X = 3$ and $X = 6$.



- Point (10, 19) will divide the space to the right of line $X = 3$ and above line $Y = 15$ into two parts.

Draw line $Y = 19$ to the right of line $X = 3$ and above line $Y = 15$.



Following is C++ implementation of KD Tree basic operations like search, insert and delete.

```
// A C++ program to demonstrate operations of KD tree
#include <iostream>
#include <cstdio>
#include <cassert>
#include <cstdlib>
using namespace std;

// A structure to represent a point in K dimensional space
// k: K dimensional space
// coord: An array to represent position of point
struct Point
{
    unsigned k;
    int* coord; // Coordinate (A pointer to array of size k)
};

// A structure to represent the Input
// n: Number of points in space
// pointArray: An array to keep information of each point
struct Input
{
    // n --> NUMBER OF POINTS
    unsigned n;
    Point* *pointArray;
};

// A structure to represent node of 2 dimensional tree
struct Node
{
    Point point;
    Node *left, *right;
};

// Creates and return a Point structure
Point* CreatePoint(unsigned k)
{
    Point* point = new Point;

    // Memory allocation failure
    assert(NULL != point);

    point->k = k;
    point->coord = new int[k];

    // Memory allocation failure
    assert(NULL != point->coord);
}
```

```

    return point;
}

// Creates and returns an Input structure
struct Input* CreateInput(unsigned k, unsigned n)
{
    struct Input* input = new Input;

    // Memory allocation failure
    assert(NULL != input);

    input->n = n;
    input->pointArray = new Point*[n];

    // Memory allocation failure
    assert(NULL != input->pointArray);

    return input;
}

// A method to create a node of K D tree
struct Node* CreateNode(struct Point* point)
{
    struct Node* tempNode = new Node;

    // Memory allocation failure
    assert(NULL != tempNode);

    // Avoid shallow copy [We could have directly use
    // the below assignment, But didn't, why?]
    /*tempNode->point = point;*/
    (tempNode->point).k = point->k;
    (tempNode->point).coord = new int[point->k];

    // Copy coordinate values
    for (int i=0; i<(tempNode->point).k; ++i)
        (tempNode->point).coord[i] = point->coord[i];

    tempNode->left = tempNode->right = NULL;
    return tempNode;
}

// Root is passed as pointer to pointer so that
// The parameter depth is used to decide axis of comparison
void InsertKDTreeUtil(Node * * root, Node* newNode, unsigned depth)
{
    // Tree is empty?
    if (!*root)
    {
        *root = newNode;
        return;
    }

    // Calculate axis of comparison to determine left/right
    unsigned axisOfComparison = depth % (newNode->point).k;

    // Compare the new point with root and decide the left or
    // right subtree
    if ((newNode->point).coord[axisOfComparison] <
        ((*root)->point).coord[axisOfComparison])
        InsertKDTreeUtil(&((*root)->left), newNode, depth + 1);
    else
        InsertKDTreeUtil(&((*root)->right), newNode, depth + 1);
}

```

```

        else
            InsertKDTreeUtil(&((*root)->right), newNode, depth + 1);
    }

    // Function to insert a new point in KD Tree. It mainly uses
    // above recursive function "InsertKDTreeUtil()"
    void InsertKDTree(Node* *root, Point* point)
    {
        Node* newNode = CreateNode(point);
        unsigned zeroDepth = 0;
        InsertKDTreeUtil(root, newNode, zeroDepth);
    }

    // A utility method to determine if two Points are same
    // in K Dimensional space
    int ArePointsSame(Point firstPoint, Point secondPoint)
    {
        if (firstPoint.k != secondPoint.k)
            return 0;

        // Compare individual coordinate values
        for (int i = 0; i < firstPoint.k; ++i)
            if (firstPoint.coord[i] != secondPoint.coord[i])
                return 0;

        return 1;
    }

    // Searches a Point in the K D tree. The parameter depth is used
    // to determine current axis.
    int SearchKDTreeUtil(Node* root, Point point, unsigned depth)
    {
        if (!root)
            return 0;

        if (ArePointsSame(root->point, point))
            return 1;

        unsigned axisOfComparison = depth % point.k;

        if (point.coord[axisOfComparison] <
            (root->point).coord[axisOfComparison])
            return SearchKDTreeUtil(root->left, point, depth + 1);

        return SearchKDTreeUtil(root->right, point, depth + 1);
    }

    // Searches a Point in the K D tree. It mainly uses
    // SearchKDTreeUtil()
    int SearchKDTree(Node* root, Point point)
    {
        unsigned zeroDepth = 0;
        return SearchKDTreeUtil(root, point, zeroDepth);
    }

    // Creates a KD tree from given input points. It mainly
    // uses InsertKDTree
    Node* CreateKDTree(Input* input)
    {
        Node* root = NULL;
        for (int i = 0; i < input->n; ++i)
            InsertKDTree(&root, input->pointArray[i]);
        return root;
    }

```

```

}

// A utility function to print an array
void PrintArray(int* array, unsigned size)
{
    for (unsigned i = 0; i < size; ++i)
        cout << array[i];
    cout << endl;
}

// A utility function to do inorder tree traversal
void inorderKD(Node* root)
{
    if (root)
    {
        inorderKD(root->left);
        PrintArray((root->point).coord, (root->point).k);
        inorderKD(root->right);
    }
}

// Driver program to test above functions
int main()
{
    // 2 Dimensional tree [For the sake of simplicity]
    unsigned k = 2;

    // Total number of Points is 7
    unsigned n = 7;
    Input* input = CreateInput(k, n);

    // itc --> ITERATOR for coord
    // itp --> ITERATOR for POINTS
    for (int itp = 0; itp < n; ++itp)
    {
        input->pointArray[itp] = CreatePoint(k);

        for (int itc = 0; itc < k; ++itc)
            input->pointArray[itp]->coord[itc] = rand() % 20;

        PrintArray(input->pointArray[itp]->coord, k);
    }

    Node* root = CreateKDTree(input);

    cout << "Inorder traversal of K-D Tree created is:\n";
    inorderKD(root);

    return 0;
}

```

Output:

```

17
140
94
1818
24
55

```

```

17
Inorder traversal of K-D Tree created is:
17
140
24
17
55
94
1818

```

This article is compiled by **Aashish Barnwal**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

135. Print Nodes in Top View of Binary Tree

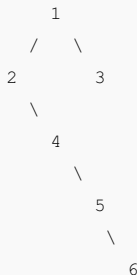
Top view of a binary tree is the set of nodes visible when the tree is viewed from the top. Given a binary tree, print the top view of it. The output nodes can be printed in any order. Expected time complexity is $O(n)$

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

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

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.

```
// Java program to print top view of Binary tree
import java.util.*;

// Class for a tree node
class TreeNode
{
    // Members
    int key;
    TreeNode left, right;

    // Constructor
    public TreeNode(int key)
    {
        this.key = key;
        left = right = null;
    }
}

// A class to represent a queue item. The queue is used to do Level
// order traversal. Every Queue item contains node and horizontal
// distance of node from root
class QItem
{
    TreeNode node;
    int hd;
    public QItem(TreeNode n, int h)
    {
        node = n;
        hd = h;
    }
}

// Class for a Binary Tree
class Tree
{
    TreeNode root;
```

```

// Constructors
public Tree() { root = null; }
public Tree(TreeNode n) { root = n; }

// This method prints nodes in top view of binary tree
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));
    }
}

// Driver class to test above methods

```



```

public class Main
{
    public static void main(String[] args)
    {
        /* Create following Binary Tree
            1
           / \
          2   3
           \
            4
             \
              5
               \
                6*/

        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.left.right = new TreeNode(4);
        root.left.right.right = new TreeNode(5);
        root.left.right.right.right = new TreeNode(6);
        Tree t = new Tree(root);
        System.out.println("Following are nodes in top view of Binary Tree");
        t.printTopView();
    }
}

```

Output:

```

Following are nodes in top view of Binary Tree
1 2 3 6

```

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.

This article is contributed by **Rohan**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

136. 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:

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.

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

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.

```
/* C++ program for special order traversal */
#include <iostream>
#include <queue>
using namespace std;

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct Node
{
    int data;
    Node *left;
    Node *right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
Node *newNode(int data)
{
    Node *node = new Node;
    node->data = data;
    node->right = node->left = NULL;
    return node;
}
```

```

/* 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);
        }
    }
}

/* Driver program to test above functions*/
int main()
{
    //Perfect Binary Tree of Height 4
    Node *root = newNode(1);

    root->left      = newNode(2);
    root->right     = newNode(3);

```

```

root->left->left = newNode(4);
root->left->right = newNode(5);
root->right->left = newNode(6);
root->right->right = newNode(7);

root->left->left->left = newNode(8);
root->left->left->right = newNode(9);
root->left->right->left = newNode(10);
root->left->right->right = newNode(11);
root->right->left->left = newNode(12);
root->right->left->right = newNode(13);
root->right->right->left = newNode(14);
root->right->right->right = newNode(15);

root->left->left->left->left = newNode(16);
root->left->left->left->right = newNode(17);
root->left->left->right->left = newNode(18);
root->left->left->right->right = newNode(19);
root->left->right->left->left = newNode(20);
root->left->right->left->right = newNode(21);
root->left->right->right->left = newNode(22);
root->left->right->right->right = newNode(23);
root->right->left->left->left = newNode(24);
root->right->left->left->right = newNode(25);
root->right->left->right->left = newNode(26);
root->right->left->right->right = newNode(27);
root->right->right->left->left = newNode(28);
root->right->right->left->right = newNode(29);
root->right->right->right->left = newNode(30);
root->right->right->right->right = newNode(31);

cout << "Specific Level Order traversal of binary tree is \n";
printSpecificLevelOrder(root);

return 0;
}

```

Output:

Followup Questions:

1. 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](#))
2. What if tree is not perfect, but complete.
3. What if tree is neither perfect, nor complete. It can be any general binary tree.

This article is contributed by **Anurag Singh**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Given n appointments, find all conflicting appointments.

Examples:

```
Input: appointments[] = { {1, 5} {3, 7}, {2, 6}, {10, 15}, {5, 6}, {4, 100}}
Output: Following are conflicting intervals
[3,7] Conflicts with [1,5]
[2,6] Conflicts with [1,5]
[5,6] Conflicts with [3,7]
[4,100] Conflicts with [1,5]
```

An appointment is conflicting, if it conflicts with any of the previous appointments in array.

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

A **Simple Solution** is to one by one process all appointments from second appointment to last. For every appointment i, check if it conflicts with i-1, i-2, ... 0. The time complexity of this method is $O(n^2)$.

We can use **Interval Tree** to solve this problem in $O(n \log n)$ time. Following is detailed algorithm.

- 1) Create an Interval Tree, initially with the first appointment.
- 2) Do following for all other appointments starting from the second one.
 - a) Check if the current appointment conflicts with any of the existing appointments in Interval Tree. If conflicts, then print the current appointment. This step can be done $O(\log n)$ time.
 - b) Insert the current appointment in Interval Tree. This step also can be done $O(\log n)$ time.

Following is C++ implementation of above idea.

```
// C++ program to print all conflicting appointments in a
// given set of appointments
#include <iostream>
using namespace std;

// Structure to represent an interval
struct Interval
{
    int low, high;
};

// Structure to represent a node in Interval Search Tree
struct ITNode
{
    Interval *i; // 'i' could also be a normal variable
    int max;
    ITNode *left, *right;
};

// A utility function to create a new Interval Search Tree Node
```

```

// A utility function to create a new Interval Search Tree Node
ITNode * newNode(Interval i)
{
    ITNode *temp = new ITNode;
    temp->i = new Interval(i);
    temp->max = i.high;
    temp->left = temp->right = NULL;
};

// A utility function to insert a new Interval Search Tree
// Node. This is similar to BST Insert. Here the low value
// of interval is used to maintain BST property
ITNode *insert(ITNode *root, Interval i)
{
    // Base case: Tree is empty, new node becomes root
    if (root == NULL)
        return newNode(i);

    // Get low value of interval at root
    int l = root->i->low;

    // If root's low value is smaller, then new interval
    // goes to left subtree
    if (i.low < l)
        root->left = insert(root->left, i);

    // Else, new node goes to right subtree.
    else
        root->right = insert(root->right, i);

    // Update the max value of this ancestor if needed
    if (root->max < i.high)
        root->max = i.high;

    return root;
}

// A utility function to check if given two intervals overlap
bool doOverlap(Interval i1, Interval i2)
{
    if (i1.low < i2.high && i2.low < i1.high)
        return true;
    return false;
}

// The main function that searches a given interval i
// in a given Interval Tree.
Interval *overlapSearch(ITNode *root, Interval i)
{
    // Base Case, tree is empty
    if (root == NULL) return NULL;

    // If given interval overlaps with root
    if (doOverlap(*(root->i), i))
        return root->i;

    // If left child of root is present and max of left child
    // is greater than or equal to given interval, then i may
    // overlap with an interval in left subtree
    if (root->left != NULL && root->left->max >= i.low)
        return overlapSearch(root->left, i);

    // Else interval can only overlap with right subtree

```

```

// These intervals can only overlap with right subtree
return overlapSearch(root->right, i);
}

// This function prints all conflicting appointments in a given
// array of appointments.
void printConflicting(Interval appt[], int n)
{
    // Create an empty Interval Search Tree, add first
    // appointment
    ITNode *root = NULL;
    root = insert(root, appt[0]);

    // Process rest of the intervals
    for (int i=1; i<n; i++)
    {
        // If current appointment conflicts with any of the
        // existing intervals, print it
        Interval *res = overlapSearch(root, appt[i]);
        if (res != NULL)
            cout << "[" << appt[i].low << "," << appt[i].high
                << "]" Conflicts with [" << res->low << ","
                << res->high << "]\n";

        // Insert this appointment
        root = insert(root, appt[i]);
    }
}

```

```

// Driver program to test above functions
int main()
{
    // Let us create interval tree shown in above figure
    Interval appt[] = { {1, 5}, {3, 7}, {2, 6}, {10, 15},
                        {5, 6}, {4, 100}};
    int n = sizeof(appt)/sizeof(appt[0]);
    cout << "Following are conflicting intervals\n";
    printConflicting(appt, n);
    return 0;
}

```

Output:

```

Following are conflicting intervals
[3,7] Conflicts with [1,5]
[2,6] Conflicts with [1,5]
[5,6] Conflicts with [3,7]
[4,100] Conflicts with [1,5]

```

Note that the above implementation uses simple Binary Search Tree insert operations. Therefore, time complexity of the above implementation is more than $O(n \log n)$. We can use [Red-Black Tree](#) or [AVL Tree](#) balancing techniques to make the above implementation $O(n \log n)$.

This article is contributed by **Anmol**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

138. How to Implement Reverse DNS Look Up Cache?

Reverse DNS look up is using an internet IP address to find a domain name. For example, if you type 74.125.200.106 in browser, it automatically redirects to google.in.

How to implement Reverse DNS Look Up cache? Following are the operations needed from cache.

- 1) Add a IP address to URL Mapping in cache.
- 2) Find URL for a given IP address.

One solution is to use [Hashing](#).

In this post, a [Trie](#) based solution is discussed. One advantage of Trie based solutions is, worst case upper bound is $O(1)$ for Trie, for hashing, the best possible average case time complexity is $O(1)$. Also, with Trie we can implement prefix search (finding all urls for a common prefix of IP addresses).

The general disadvantage of Trie is large amount of memory requirement, this is not a major problem here as the alphabet size is only 11 here. Ten characters are needed for digits from '0' to '9' and one for dot ('.').

The idea is to store IP addresses in Trie nodes and in the last node we store the corresponding domain name. Following is C style implementation in C++.

```
// C based program to implement reverse DNS lookup
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

// There are atmost 11 different chars in a valid IP address
#define CHARS 11

// Maximum length of a valid IP address
#define MAX 50

// A utility function to find index of child for a given character 'c'
int getIndex(char c) { return (c == '.')? 10: (c - '0'); }

// A utility function to find character for a given child index.
char getCharFromIndex(int i) { return (i== 10)? '.' : ('0' + i); }

// Trie Node.
struct trieNode
{
    bool isLeaf;
    char *URL;
    struct trieNode *child[CHARS];
};

// Function to create a new trie node.
struct trieNode *newTrieNode(void)
{
```



```

    struct trieNode *newNode = new trieNode;
    newNode->isLeaf = false;
    newNode->URL = NULL;
    for (int i=0; i<CHARS; i++)
        newNode->child[i] = NULL;
    return newNode;
}

// This method inserts an ip address and the corresponding
// domain name in the trie. The last node in Trie contains the URL.
void insert(struct trieNode *root, char *ipAdd, char *URL)
{
    // Length of the ip address
    int len = strlen(ipAdd);
    struct trieNode *pCrawl = root;

    // Traversing over the length of the ip address.
    for (int level=0; level<len; level++)
    {
        // Get index of child node from current character
        // in ipAdd[]. Index must be from 0 to 10 where
        // 0 to 9 is used for digits and 10 for dot
        int index = getIndex(ipAdd[level]);

        // Create a new child if not exist already
        if (!pCrawl->child[index])
            pCrawl->child[index] = newTrieNode();

        // Move to the child
        pCrawl = pCrawl->child[index];
    }

    //Below needs to be carried out for the last node.
    //Save the corresponding URL of the ip address in the
    //last node of trie.
    pCrawl->isLeaf = true;
    pCrawl->URL = new char[strlen(URL) + 1];
    strcpy(pCrawl->URL, URL);
}

// This function returns URL if given IP address is present in DNS cache
// Else returns NULL
char *searchDNSCache(struct trieNode *root, char *ipAdd)
{
    // Root node of trie.
    struct trieNode *pCrawl = root;
    int len = strlen(ipAdd);

    // Traversal over the length of ip address.
    for (int level=0; level<len; level++)
    {
        int index = getIndex(ipAdd[level]);
        if (!pCrawl->child[index])
            return NULL;
        pCrawl = pCrawl->child[index];
    }

    // If we find the last node for a given ip address, print the URL.
    if (pCrawl!=NULL && pCrawl->isLeaf)
        return pCrawl->URL;

    return NULL;
}

```

```

}

//Driver function.
int main()
{
    /* Change third ipAddress for validation */
    char ipAdd[][MAX] = {"107.108.11.123", "107.109.123.255",
                        "74.125.200.106"};
    char URL[][50] = {"www.samsung.com", "www.samsung.net",
                    "www.google.in"};
    int n = sizeof(ipAdd)/sizeof(ipAdd[0]);
    struct trieNode *root = newTrieNode();

    // Inserts all the ip address and their corresponding
    // domain name after ip address validation.
    for (int i=0; i<n; i++)
        insert(root,ipAdd[i],URL[i]);

    // If reverse DNS look up succeeds print the domain
    // name along with DNS resolved.
    char ip[] = "107.108.11.123";
    char *res_url = searchDNSCache(root, ip);
    if (res_url != NULL)
        printf("Reverse DNS look up resolved in cache:\n%s --> %s",
            ip, res_url);
    else
        printf("Reverse DNS look up not resolved in cache ");
    return 0;
}

```

Output:

```

Reverse DNS look up resolved in cache:
107.108.11.123 --> www.samsung.com

```

Note that the above implementation of Trie assumes that the given IP address does not contain characters other than {'0', '1',..... '9', '.'}. What if a user gives an invalid IP address that contains some other characters? This problem can be resolved by [validating the input IP address](#) before inserting it into Trie. We can use the approach discussed [here](#) for IP address validation.

This article is contributed by **Kumar Gautam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

139. Binary Indexed Tree or Fenwick tree

Let us consider the following problem to understand Binary Indexed Tree.

We have an array arr[0 . . . n-1]. We should be able to

1 Find the sum of first i elements.

2 Change value of a specified element of the array $arr[i] = x$ where $0 \leq i \leq n-1$.

A **simple solution** is to run a loop from 0 to $i-1$ and calculate sum of elements. To update a value, simply do $arr[i] = x$. The first operation takes $O(n)$ time and second operation takes $O(1)$ time. Another simple solution is to create another array and store sum from start to i at the i 'th index in this array. Sum of a given range can now be calculated in $O(1)$ time, but update operation takes $O(n)$ time now. This works well if the number of query operations are large and very few updates.

Can we perform both the operations in $O(\log n)$ time once given the array?

One Efficient Solution is to use **Segment Tree** that does both operations in $O(\log n)$ time.

Using Binary Indexed Tree, we can do both tasks in $O(\log n)$ time. The advantages of Binary Indexed Tree over Segment are, requires less space and very easy to implement..

Representation

Binary Indexed Tree is represented as an array. Let the array be `BITree[]`. Each node of Binary Indexed Tree stores sum of some elements of given array. Size of Binary Indexed Tree is equal to n where n is size of input array. In the below code, we have used size as $n+1$ for ease of implementation.

Construction

We construct the Binary Indexed Tree by first initializing all values in `BITree[]` as 0. Then we call `update()` operation for all indexes to store actual sums, update is discussed below.

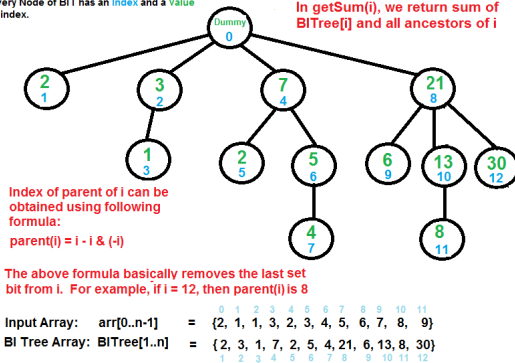
Operations

`getSum(index)`: Returns sum of `arr[0..index]`

```
// Returns sum of arr[0..index] using BITree[0..n]. It assumes that
// BITree[] is constructed for given array arr[0..n-1]
1) Initialize sum as 0 and index as index+1.
2) Do following while index is greater than 0.
...a) Add BITree[index] to sum
...b) Go to parent of BITree[index]. Parent can be obtained by removing
    the last set bit from index, i.e., index = index - (index & (-index))
3) Return sum.
```

Every Node of BIT has an Index and a Value at index.

In `getSum(i)`, we return sum of `BITree[i]` and all ancestors of `i`



View of Binary Indexed Tree to understand `getSum()` operation

The above diagram demonstrates working of `getSum()`. Following are some important observations.

Node at index 0 is a dummy node.

A node at index `y` is parent of a node at index `x`, iff `y` can be obtained by removing last set bit from binary representation of `x`.

A child `x` of a node `y` stores sum of elements from of `y`(exclusive `y`) and of `x`(inclusive `x`).

`update(index, val)`: Updates BIT for operation `arr[index] += val`

// Note that `arr[]` is not changed here. It changes

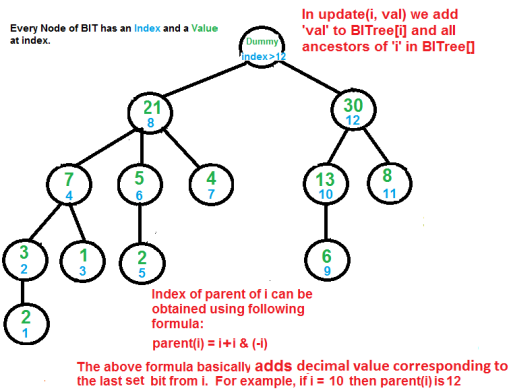
// only BI Tree for the already made change in `arr[]`.

1) Initialize index as `index+1`.

2) Do following while index is smaller than or equal to `n`.

...a) Add value to `BITree[index]`

...b) Go to parent of `BITree[index]`. Parent can be obtained by removing the last set bit from index, i.e., `index = index - (index & (-index))`



Contents of arr[] and BITree[] are same as above diagram for getSum()

View of Binary Indexed Tree to understand update() operation

The update process needs to make sure that all BITree nodes that have arr[i] as part of the section they cover must be updated. We get all such nodes of BITree by repeatedly adding the decimal number corresponding to the last set bit.

How does Binary Indexed Tree work?

The idea is based on the fact that all positive integers can be represented as sum of powers of 2. For example 19 can be represented as $16 + 2 + 1$. Every node of BI Tree stores sum of n elements where n is a power of 2. For example, in the above first diagram for getSum(), sum of first 12 elements can be obtained by sum of last 4 elements (from 9 to 12) plus sum of 8 elements (from 1 to 8). The number of set bits in binary representation of a number n is $O(\text{Log}n)$. Therefore, we traverse at-most $O(\text{Log}n)$ nodes in both getSum() and update() operations. Time complexity of construction is $O(n\text{Log}n)$ as it calls update() for all n elements.

Implementation:

Following is C++ implementation of Binary Indexed Tree.

```
// C++ code to demonstrate operations of Binary Index Tree
#include <iostream>
using namespace std;

/*      n --> No. of elements present in input array.
    BITree[0..n] --> Array that represents Binary Indexed Tree.
    arr[0..n-1] --> Input array for whic prefix sum is evaluated. */

// Returns sum of arr[0..index]. This function assumes
// that the array is preprocessed and partial sums of
// array elements are stored in BITree[].
int getSum(int BITree[], int n, int index)
{
    int sum = 0; // Initialize result

    // index in BITree[] is 1 more than the index in arr[]
    index = index + 1;
```

```

    index = index / 2;

    // Traverse ancestors of BITree[index]
    while (index>0)
    {
        // Add current element of BITree to sum
        sum += BITree[index];

        // Move index to parent node
        index -= index & (-index);
    }
    return sum;
}

// Updates a node in Binary Index Tree (BITree) at given index
// in BITree. The given value 'val' is added to BITree[i] and
// all of its ancestors in tree.
void updateBIT(int *BITree, int n, int index, int val)
{
    // index in BITree[] is 1 more than the index in arr[]
    index = index + 1;

    // Traverse all ancestors and add 'val'
    while (index <= n)
    {
        // Add 'val' to current node of BI Tree
        BITree[index] += val;

        // Update index to that of parent
        index += index & (-index);
    }
}

```

```

// Constructs and returns a Binary Indexed Tree for given
// array of size n.
int *constructBITree(int arr[], int n)
{
    // Create and initialize BITree[] as 0
    int *BITree = new int[n+1];
    for (int i=1; i<=n; i++)
        BITree[i] = 0;

    // Store the actual values in BITree[] using update()
    for (int i=0; i<n; i++)
        updateBIT(BITree, n, i, arr[i]);

    // Uncomment below lines to see contents of BITree[]
    //for (int i=1; i<=n; i++)
    //    cout << BITree[i] << " ";

    return BITree;
}

```

```

// Driver program to test above functions
int main()
{
    int freq[] = {2, 1, 1, 3, 2, 3, 4, 5, 6, 7, 8, 9};
    int n = sizeof(freq)/sizeof(freq[0]);
    int *BITree = constructBITree(freq, n);
    cout << "Sum of elements in arr[0..5] is "
         << getSum(BITree, n, 5);
}

```

```

// Let use test the update operation
freq[3] += 6;
updateBIT(BITree, n, 3, 6); //Update BIT for above change in arr[]

cout << "\nSum of elements in arr[0..5] after update is "
      << getSum(BITree, n, 5);

return 0;
}

```

Output:

```

Sum of elements in arr[0..5] is 12
Sum of elements in arr[0..5] after update is 18

```

Can we extend the Binary Indexed Tree for range Sum in Logn time?

This is simple to answer. The rangeSum(l, r) can be obtained as getSum(r) – getSum(l-1).

Applications:

Used to implement the arithmetic coding algorithm. Development of operations it supports were primarily motivated by use in that case. See [this](#) for more details.

References:

http://en.wikipedia.org/wiki/Fenwick_tree

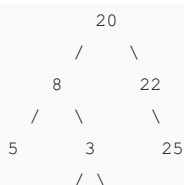
<http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=binaryIndexedTrees>

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

140. Bottom View of a 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.

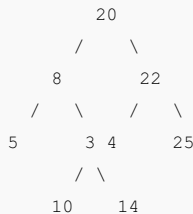
Examples:



10 14

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.

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

The following are steps to print Bottom View of Binary 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.

A Java based implementation is below :

```
// Java Program to print Bottom View of Binary Tree
import java.util.*;
import java.util.Map.Entry;

// Tree node class
class TreeNode
{
    int data; //data of the node
    int hd; //horizontal distance of the node
    TreeNode left, right; //left and right references

    // Constructor of tree node
```



```

public TreeNode(int key)
{
    data = key;
    hd = Integer.MAX_VALUE;
    left = right = null;
}
}

//Tree class
class Tree
{
    TreeNode root; //root node of tree

    // Default constructor
    public Tree() {}

    // Parameterized tree constructor
    public Tree(TreeNode node)
    {
        root = node;
    }

    // Method that prints the bottom view.
    public void bottomView()
    {
        if (root == null)
            return;

        // Initialize a variable 'hd' with 0 for the root element.
        int hd = 0;

        // TreeMap which stores key value pair sorted on key value
        Map<Integer, Integer> map = new TreeMap<>();

        // Queue to store tree nodes in level order traversal
        Queue<TreeNode> queue = new LinkedList<TreeNode>();

        // Assign initialized horizontal distance value to root
        // node and add it to the queue.
        root.hd = hd;
        queue.add(root);

        // Loop until the queue is empty (standard level order loop)
        while (!queue.isEmpty())
        {

```

```

        TreeNode temp = queue.remove();

        // Extract the horizontal distance value from the
        // dequeued tree node.
        hd = temp.hd;

        // Put the dequeued tree node to TreeMap having key
        // as horizontal distance. Every time we find a node
        // having same horizontal distance we need to replace
        // the data in the map.
        map.put(hd, temp.data);

        // If the dequeued node has a left child add it to the
        // queue with a horizontal distance hd-1.
        if (temp.left != null)
        {
            temp.left.hd = hd-1;
            queue.add(temp.left);
        }
        // If the dequeued node has a left child add it to the
        // queue with a horizontal distance hd+1.
        if (temp.right != null)
        {
            temp.right.hd = hd+1;
            queue.add(temp.right);
        }
    }

    // Extract the entries of map into a set to traverse
    // an iterator over that.
    Set<Entry<Integer, Integer>> set = map.entrySet();

    // Make an iterator
    Iterator<Entry<Integer, Integer>> iterator = set.iterator();

    // Traverse the map elements using the iterator.
    while (iterator.hasNext())
    {
        Map.Entry<Integer, Integer> me = iterator.next();
        System.out.print(me.getValue()+" ");
    }
}

```

```
// Main driver class
public class BottomView
{
    public static void main(String[] args)
    {
        TreeNode root = new TreeNode(20);
        root.left = new TreeNode(8);
        root.right = new TreeNode(22);
        root.left.left = new TreeNode(5);
        root.left.right = new TreeNode(3);
        root.right.left = new TreeNode(4);
        root.right.right = new TreeNode(25);
        root.left.right.left = new TreeNode(10);
        root.left.right.right = new TreeNode(14);
        Tree tree = new Tree(root);
        System.out.println("Bottom view of the given binary tree:");
        tree.bottomView();
    }
}
```

Output:

```
Bottom view of the given binary tree:
5 10 4 14 25
```

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.

This article is contributed by **Kumar Gautam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

141. Diagonal Sum of a Binary Tree

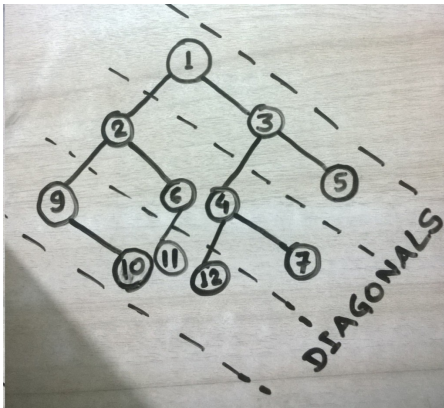
Consider lines of slope -1 passing between nodes (dotted lines in below diagram). Diagonal sum in a binary tree is sum of all node's data lying between these lines. Given a Binary Tree, print all diagonal sums.

For the following input tree, output should be 9, 19, 42.

9 is sum of 1, 3 and 5.

19 is sum of 2, 6, 4 and 7.

42 is sum of 9, 10, 11 and 12.



We strongly recommend to minimize your browser and try this yourself first

Algorithm:

The idea is to keep track of vertical distance from top diagonal passing through root.

We increment the vertical distance we go down to next diagonal.

1. Add root with vertical distance as 0 to the queue.
2. Process the sum of all right child and right of right child and so on.
3. Add left child current node into the queue for later processing. The vertical distance of left child is vertical distance of current node plus 1.
4. Keep doing 2nd, 3rd and 4th step till the queue is empty.

Following is Java implementation of above idea.

```
// Java Program to find diagonal sum in a Binary Tree
import java.util.*;
import java.util.Map.Entry;

//Tree node
class TreeNode
{
    int data; //node data
    int vd; //vertical distance diagonally
    TreeNode left, right; //left and right child's reference

    // Tree node constructor
    public TreeNode(int data)
    {
        this.data = data;
        vd = Integer.MAX_VALUE;
        left = right = null;
    }
}
```

```

}

// Tree class
class Tree
{
    TreeNode root;//Tree root

    // Tree constructor
    public Tree(TreeNode root) { this.root = root; }

    // Diagonal sum method
    public void diagonalSum()
    {
        // Queue which stores tree nodes
        Queue<TreeNode> queue = new LinkedList<TreeNode>();

        // Map to store sum of node's data lying diagonally
        Map<Integer, Integer> map = new TreeMap<>();

        // Assign the root's vertical distance as 0.
        root.vd = 0;

        // Add root node to the queue
        queue.add(root);

        // Loop while the queue is not empty
        while (!queue.isEmpty())
        {
            // Remove the front tree node from queue.
            TreeNode curr = queue.remove();

            // Get the vertical distance of the dequeued node.
            int vd = curr.vd;

            // Sum over this node's right-child, right-of-right-child
            // and so on
            while (curr != null)
            {
                int prevSum = (map.get(vd) == null)? 0: map.get(vd);
                map.put(vd, prevSum + curr.data);

                // If for any node the left child is not null add
                // it to the queue for future processing.
                if (curr.left != null)
                {

```

```

        curr.left.vd = vd+1;
        queue.add(curr.left);
    }

    // Move to the current node's right child.
    curr = curr.right;
}

}

// Make an entry set from map.
Set<Entry<Integer, Integer>> set = map.entrySet();

// Make an iterator
Iterator<Entry<Integer, Integer>> iterator = set.iterator();

// Traverse the map elements using the iterator.
while (iterator.hasNext())
{
    Map.Entry<Integer, Integer> me = iterator.next();
    System.out.print(me.getValue()+" ");
}
}

}

//Driver class
public class DiagonalSum
{
    public static void main(String[] args)
    {
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.left.left = new TreeNode(9);
        root.left.right = new TreeNode(6);
        root.right.left = new TreeNode(4);
        root.right.right = new TreeNode(5);
        root.right.left.left = new TreeNode(12);
        root.right.left.right = new TreeNode(7);
        root.left.right.left = new TreeNode(11);
        root.left.left.right = new TreeNode(10);
        Tree tree = new Tree(root);
        tree.diagonalSum();
    }
}

```

Output:

```
9, 19, 42
```

Exercise:

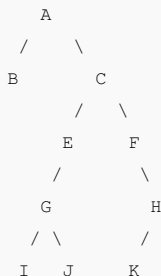
This problem was for diagonals from top to bottom and slope -1. Try the same problem for slope +1.

This article is contributed by **Kumar Gautam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

142. Find the closest leaf in a Binary Tree

Given a Binary Tree and a key 'k', find distance of the closest leaf from 'k'.

Examples:



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'

We strongly recommend to minimize your browser and try this yourself first

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.

```
// A C++ program to find the closest leaf of a given key in Binary Tree
```

```

// A C++ program to find the closest leaf of a given key in Binary tree
#include <iostream>
#include <climits>
using namespace std;

/* A binary tree Node has key, pointer to left and right children */
struct Node
{
    char key;
    struct Node* left, *right;
};

/* Helper function that allocates a new node with the
given data and NULL left and right pointers. */
Node *newNode(char k)
{
    Node *node = new Node;
    node->key = k;
    node->right = node->left = NULL;
    return node;
}

// A utility function to find minimum of x and y
int getMin(int x, int y)
{
    return (x < y)? x :y;
}

// A utility function to find distance of closest leaf of the tree
// rooted under given root
int closestDown(struct Node *root)
{
    // Base cases
    if (root == NULL)
        return INT_MAX;
    if (root->left == NULL && root->right == NULL)
        return 0;

    // Return minimum of left and right, plus one
    return 1 + getMin(closestDown(root->left), closestDown(root->right));
}

// Returns distance of the closest leaf to a given key 'k'. The array
// ancestors is used to keep track of ancestors of current node and
// 'index' is used to keep track of current index in 'ancestors[]'
int findClosestUtil(struct Node *root, char k, struct Node *ancestors[100],
int index)
{
    // Base case
    if (root == NULL)
        return INT_MAX;

    // If key found
    if (root->key == k)
    {
        // Find the closest leaf under the subtree rooted with given k
        int res = closestDown(root);

        // Traverse all ancestors and update result if any parent node
        // gives smaller distance
        for (int i = index-1; i>=0; i--)
            res = getMin(res, index - i + closestDown(ancestors[i]));
        return res;
    }
}

```



```

    }

    // If key node found, store current node and recur for left and
    // right childrens
    ancestors[index] = root;
    return getMin(findClosestUtil(root->left, k, ancestors, index+1),
                  findClosestUtil(root->right, k, ancestors, index+1))
}

// The main function that returns distance of the closest key to 'k'.
// mainly uses recursive function findClosestUtil() to find the closes
// distance.
int findClosest(struct Node *root, char k)
{
    // Create an array to store ancestors
    // Assumption: Maximum height of tree is 100
    struct Node *ancestors[100];

    return findClosestUtil(root, k, ancestors, 0);
}

/* Driver program to test above functions*/
int main()
{
    // Let us construct the BST shown in the above figure
    struct Node *root = newNode('A');
    root->left = newNode('B');
    root->right = newNode('C');
    root->right->left = newNode('E');
    root->right->right = newNode('F');
    root->right->left->left = newNode('G');
    root->right->left->left->left = newNode('I');
    root->right->left->left->right = newNode('J');
    root->right->right->right = newNode('H');
    root->right->right->right->left = newNode('K');

    char k = 'H';
    cout << "Distance of the closest key from " << k << " is "
    << findClosest(root, k) << endl;
    k = 'C';
    cout << "Distance of the closest key from " << k << " is "
    << findClosest(root, k) << endl;
    k = 'E';
    cout << "Distance of the closest key from " << k << " is "
    << findClosest(root, k) << endl;
    k = 'B';
    cout << "Distance of the closest key from " << k << " is "
    << findClosest(root, k) << endl;

    return 0;
}

```

Output:

```

Distance of the closest key from H is 1
Distance of the closest key from C is 2
Distance of the closest key from E is 2
Distance of the closest key from B is 0

```

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 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.

This article is contributed by Shubham. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

143. Remove nodes on root to leaf paths of 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 $\geq k$, 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 .

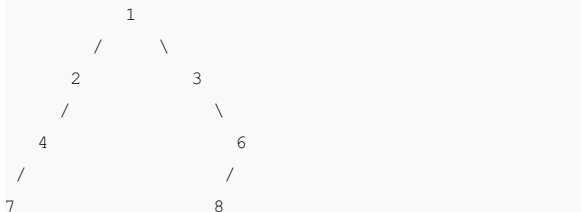
Consider the following example Binary Tree



Input: Root of above Binary Tree

$k = 4$

Output: The tree should be changed to following



There are 3 paths

- i) $1 \rightarrow 2 \rightarrow 4 \rightarrow 7$ path length = 4
- ii) $1 \rightarrow 2 \rightarrow 5$ path length = 3
- iii) $1 \rightarrow 3 \rightarrow 6 \rightarrow 8$ 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.

We strongly recommend to minimize your browser and try this yourself first

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 for a node whose child or cho

- 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 $\geq k$. In that case it needs not to be deleted.

A C++ based code on this approach is below

```
// C++ program to remove nodes on root to leaf paths of length < K
#include<iostream>
using namespace std;

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

//New node of a tree
Node *newNode(int data)
{
    Node *node = new Node;
    node->data = data;
    node->left = node->right = NULL;
    return node;
}

// 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 utility method to remove the short path
// nodes.
Node *removeShortPathNodes(Node *root, int k)
{
    int pathLen = 0;
    return removeShortPathNodesUtil(root, 1, k);
}

//Method to print the tree in inorder fashion.
void printInorder(Node *root)
{
    if (root)
    {
        printInorder(root->left);
        cout << root->data << " ";
        printInorder(root->right);
    }
}

// Driver method.
int main()
{
    int k = 4;
    Node *root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->left->left->left = newNode(7);
    root->right->right = newNode(6);
    root->right->right->left = newNode(8);
    cout << "Inorder Traversal of Original tree" << endl;
    printInorder(root);
    cout << endl;
    cout << "Inorder Traversal of Modified tree" << endl;
    Node *res = removeShortPathNodes(root, k);
    printInorder(res);
    return 0;
}

```

Output:

```

Inorder Traversal of Original tree
7 4 2 5 1 3 8 6

Inorder Traversal of Modified tree
7 4 2 1 3 8 6

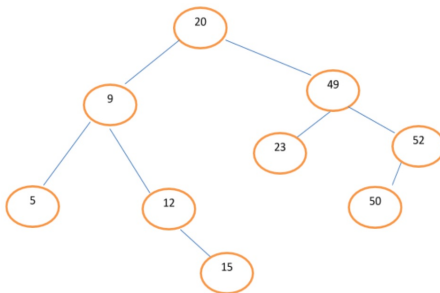
```

Time complexity of the above solution is $O(n)$ where n is number of nodes in given Binary Tree.

This article is contributed by **Kumar Gautam**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

144. Find sum of all left leaves in a 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 $5+23+50 = 78$.



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

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.

Following is C++ implementation of above idea.

```
// A C++ program to find sum of all left leaves
#include <iostream>
using namespace std;

/* A binary tree Node has key, pointer to left and right
children */
struct Node
{
    int key;
    struct Node* left, *right;
};

/* Helper function that allocates a new node with the
given data and NULL left and right pointer. */
Node *newNode(char k)
{
    Node *node = new Node;
    node->key = k;
    node->right = node->left = NULL;
}
```

```

        return node;
    }

// A utility function to check if a given node is leaf or not
bool isLeaf(Node *node)
{
    if (node == NULL)
        return false;
    if (node->left == NULL && node->right == NULL)
        return true;
    return false;
}

// 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;
}

/* Driver program to test above functions*/
int main()
{
    // Let us construct the Binary Tree shown in the
    // above figure
    struct Node *root = newNode(20);
    root->left = newNode(9);
    root->right = newNode(49);
    root->right->left = newNode(23);
    root->right->right = newNode(52);
    root->right->right->left = newNode(50);
    root->left->left = newNode(5);
    root->left->right = newNode(12);
    root->left->right->right = newNode(12);
    cout << "Sum of left leaves is "
         << leftLeavesSum(root);
    return 0;
}

```

Output:

```
Sum of left leaves is 78
```

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 $O(n)$. Thanks to Xin Tong (geeksforgeeks user [trent.tong](#)) for suggesting this method.

```
// A C++ program to find sum of all left leaves
#include <iostream>
using namespace std;

/* A binary tree Node has key, pointer to left and right
   children */
struct Node
{
    int key;
    struct Node* left, *right;
};

/* Helper function that allocates a new node with the
   given data and NULL left and right pointer. */
Node *newNode(char k)
{
    Node *node = new Node;
    node->key = k;
    node->right = node->left = NULL;
    return node;
}

/* 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;
}

/* Driver program to test above functions*/
int main()
{
    // Let us construct the Binary Tree shown in the
    // above figure
```

```

int sum = 0;
struct Node *root      = newNode(20);
root->left              = newNode(9);
root->right             = newNode(49);
root->right->left        = newNode(23);
root->right->right       = newNode(52);
root->right->right->left  = newNode(50);
root->left->left          = newNode(5);
root->left->right        = newNode(12);
root->left->right->right  = newNode(12);

cout << "Sum of left leaves is " << leftLeavesSum(root) << endl;
return 0;
}

```

Output:

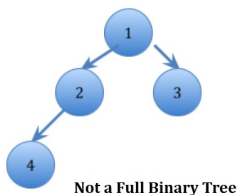
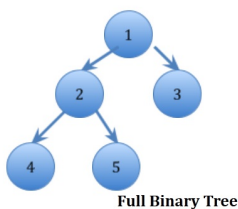
```
Sum of left leaves is 78
```

This article is contributed by **Manish**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

145. Check whether a binary tree is a full binary tree or 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:



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

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.

Following is the C code for checking if a binary tree is a full binary tree.

```
// C program to check whether a given Binary Tree is full or not
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

/* Tree node structure */
struct Node
{
    int key;
    struct Node *left, *right;
};

/* Helper function that allocates a new node with the
   given key and NULL left and right pointer. */
struct Node *newNode(char k)
{
    struct Node *node = (struct Node*)malloc(sizeof(struct Node));
    node->key = k;
    node->left = NULL;
    node->right = NULL;
    return node;
}

/* 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;
}

// Driver Program
int main()
{
    struct Node* root = NULL;
```

```

struct Node *root = NULL;
root = newNode(10);
root->left = newNode(20);
root->right = newNode(30);

root->left->right = newNode(40);
root->left->left = newNode(50);
root->right->left = newNode(60);
root->right->right = newNode(70);

root->left->left->left = newNode(80);
root->left->left->right = newNode(90);
root->left->right->left = newNode(80);
root->left->right->right = newNode(90);
root->right->left->left = newNode(80);
root->right->left->right = newNode(90);
root->right->right->left = newNode(80);
root->right->right->right = newNode(90);

if (isFullTree(root))
    printf("The Binary Tree is full\n");
else
    printf("The Binary Tree is not full\n");

return(0);
}

```

Output:

```
The Binary Tree is full
```

Time complexity of the above code is $O(n)$ where n is number of nodes in given binary tree.

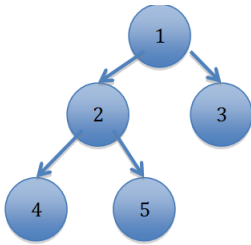
This article is contributed by **Gaurav Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

146. Check whether a binary tree is a complete tree or not | Set 2 (Recursive Solution)

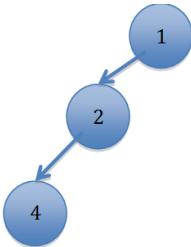
A complete binary tree is a binary tree whose all levels except the last level are completely filled and all the leaves in the last level are all to the left side. More information about complete binary trees can be found [here](#).

For Example:-

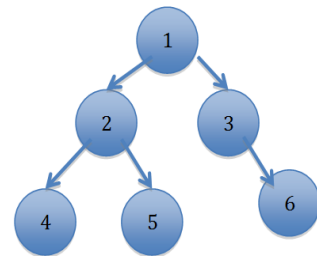
Below tree is a Complete Binary Tree (All nodes till the second last nodes are filled and all leaves are to the left side)



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).

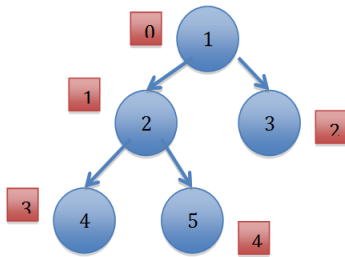


An iterative solution for this problem is discussed in below post.

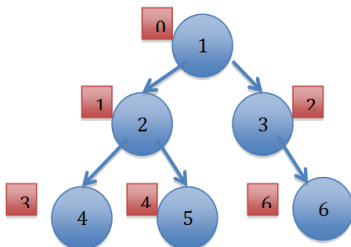
[Check whether a given Binary Tree is Complete or not | Set 1 \(Using Level Order Traversal\)](#)

In this post a recursive solution is discussed.

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.

1. Calculate the number of nodes (count) in the binary tree.
2. 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).
3. If the current node under examination is NULL, then the tree is a complete binary tree. Return true.
4. If index (i) of the current node is greater than or equal to the number of nodes in the binary tree (count) i.e. ($i \geq \text{count}$), then the tree is not a complete binary. Return false.
5. 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 C code for checking if a binary tree is a complete binary tree.

```

/* C program to checks if a binary tree complete ot not */
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

/* Tree node structure */
struct Node

```

```

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

/* Helper function that allocates a new node with the
   given key and NULL left and right pointer. */
struct Node *newNode(char k)
{
    struct Node *node = (struct Node*)malloc(sizeof(struct Node));
    node->key = k;
    node->right = node->left = NULL;
    return node;
}

/* This function counts the number of nodes in a binary tree */
unsigned int countNodes(struct Node* root)
{
    if (root == NULL)
        return (0);
    return (1 + countNodes(root->left) + countNodes(root->right));
}

/* 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));
}

// Driver program
int main()
{
    // Let us create tree in the last diagram above
    struct Node* root = NULL;
    root = newNode(1);
    root->left = newNode(2);
    root->right = newNode(3);
    root->left->left = newNode(4);
    root->left->right = newNode(5);
    root->right->right = newNode(6);

    unsigned int node_count = countNodes(root);
    unsigned int index = 0;

    if (isComplete(root, index, node_count))
        printf("The Binary Tree is complete\n");
    else
        printf("The Binary Tree is not complete\n");
    return (0);
}

```

Output:

```
The Binary Tree is not complete
```

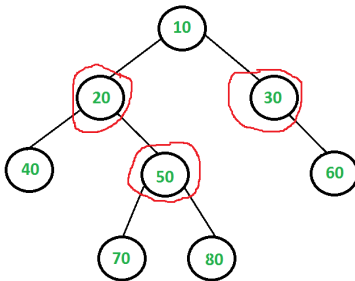
This article is contributed by **Gaurav Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

147. Vertex Cover Problem | Set 2 (Dynamic Programming Solution for Tree)

A **vertex cover** of an **undirected graph** is a subset of its vertices such that for every edge (u, v) of the graph, either 'u' or 'v' is in vertex cover. Although the name is Vertex Cover, the set covers all edges of the given graph.

The problem to find minimum size vertex cover of a graph is **NP complete**. But it can be solved in polynomial time for trees. In this post a solution for Binary Tree is discussed. The same solution can be extended for n-ary trees.

For example, consider the following binary tree. The smallest vertex cover is {20, 50, 30} and size of the vertex cover is 3.



The idea is to consider following two possibilities for root and recursively for all nodes down the root.

1) Root is part of vertex cover: In this case root covers all children edges. We recursively calculate size of vertex covers for left and right subtrees and add 1 to the result (for root).

2) Root is not part of vertex cover: In this case, both children of root must be included in vertex cover to cover all root to children edges. We recursively calculate size of vertex covers of all grandchildren and number of children to the result (for two children of root).

Below is C implementation of above idea.

```
// A naive recursive C implementation for vertex cover problem for a tree
#include <stdio.h>
```

```

#include <stdio.h>
#include <stdlib.h>

// A utility function to find min of two integers
int min(int x, int y) { return (x < y)? x: y; }

/* A binary tree node has data, pointer to left child and a pointer to
right child */
struct node
{
    int data;
    struct node *left, *right;
};

// The function returns size of the minimum vertex cover
int vCover(struct node *root)
{
    // The size of minimum vertex cover is zero if tree is empty or the
    // is only one node
    if (root == NULL)
        return 0;
    if (root->left == NULL && root->right == NULL)
        return 0;

    // Calculate size of vertex cover when root is part of it
    int size_incl = 1 + vCover(root->left) + vCover(root->right);

    // Calculate size of vertex cover when root is not part of it
    int size_excl = 0;
    if (root->left)
        size_excl += 1 + vCover(root->left->left) + vCover(root->left->right);
    if (root->right)
        size_excl += 1 + vCover(root->right->left) + vCover(root->right->right);

    // Return the minimum of two sizes
    return min(size_incl, size_excl);
}

// A utility function to create a node
struct node* newNode( int data )
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

// Driver program to test above functions
int main()
{
    // Let us construct the tree given in the above diagram
    struct node *root = newNode(20);
    root->left = newNode(8);
    root->left->left = newNode(4);
    root->left->right = newNode(12);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(14);
    root->right = newNode(22);
    root->right->right = newNode(25);

    printf ("Size of the smallest vertex cover is %d ", vCover(root));

    return 0;
}

```

```
}
```

Output:

```
Size of the smallest vertex cover is 3
```

Time complexity of the above naive recursive approach is exponential. It should be noted that the above function computes the same subproblems again and again. For example, vCover of node with value 50 is evaluated twice as 50 is grandchild of 10 and child of 20.

Since same subproblems are called again, this problem has Overlapping Subproblems property. So Vertex Cover problem has both properties (see [this](#) and [this](#)) of a dynamic programming problem. Like other typical [Dynamic Programming\(DP\) problems](#), re-computations of same subproblems can be avoided by storing the solutions to subproblems and solving problems in bottom up manner.

Following is C implementation of Dynamic Programming based solution. In the following solution, an additional field 'vc' is added to tree nodes. The initial value of 'vc' is set as 0 for all nodes. The recursive function vCover() calculates 'vc' for a node only if it is not already set.

```
/* Dynamic programming based program for Vertex Cover problem for
   a Binary Tree */
#include <stdio.h>
#include <stdlib.h>

// A utility function to find min of two integers
int min(int x, int y) { return (x < y)? x: y; }

/* A binary tree node has data, pointer to left child and a pointer to
   right child */
struct node
{
    int data;
    int vc;
    struct node *left, *right;
};

// A memoization based function that returns size of the minimum vertex
int vCover(struct node *root)
{
    // The size of minimum vertex cover is zero if tree is empty or the
    // is only one node
    if (root == NULL)
        return 0;
    if (root->left == NULL && root->right == NULL)
        return 0;

    // If vertex cover for this node is already evaluated, then return
    // to save recomputation of same subproblem again.
    if (root->vc != 0)
        return root->vc;

    // Calculate size of vertex cover when root is part of it
    int size_incl = 1 + vCover(root->left) + vCover(root->right);
```



```

// Calculate size of vertex cover when root is not part of it
int size_excl = 0;
if (root->left)
    size_excl += 1 + vCover(root->left->left) + vCover(root->left->right);
if (root->right)
    size_excl += 1 + vCover(root->right->left) + vCover(root->right->right);

// Minimum of two values is vertex cover, store it before returning
root->vc = min(size_incl, size_excl);

return root->vc;
}

```

```

// A utility function to create a node
struct node* newNode( int data )
{
    struct node* temp = (struct node *) malloc( sizeof(struct node) );
    temp->data = data;
    temp->left = temp->right = NULL;
    temp->vc = 0; // Set the vertex cover as 0
    return temp;
}

```

```

// Driver program to test above functions
int main()
{
    // Let us construct the tree given in the above diagram
    struct node *root = newNode(20);
    root->left = newNode(8);
    root->left->left = newNode(4);
    root->left->right = newNode(12);
    root->left->right->left = newNode(10);
    root->left->right->right = newNode(14);
    root->right = newNode(22);
    root->right->right = newNode(25);

    printf ("Size of the smallest vertex cover is %d ", vCover(root));

    return 0;
}

```

Output:

```
Size of the smallest vertex cover is 3
```

References:

<http://courses.csail.mit.edu/6.006/spring11/lectures/lec21.pdf>

Exercise:

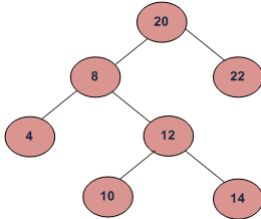
Extend the above solution for n-ary trees.

This article is contributed by **Udit Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

148. K'th 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?

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

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.

```
// C++ program to find k'th largest element in BST
```

```
#include<iostream>
```

```
using namespace std;
```

```
struct Node
```

```
{
```

```
    int key;
```

```
    Node *left, *right;
```

```
};
```

```
// A utility function to create a new BST node
```

```
Node *newNode(int item)
```

```
{
```

```
    Node *temp = new Node;
```

```
    temp->key = item;
```

```
    temp->left = temp->right = NULL;
```

```
    return temp;
```

```
}
```

```
// A function to find k'th largest element in a given tree.
```

```
void kthLargestUtil(Node *root, int k, int &res)
```

```

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);
}

```

```

/* A utility function to insert a new node with given key in BST */
Node* insert(Node* node, int key)

```

```

{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}

```

```

// Driver Program to test above functions

```

```

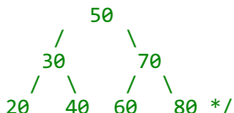
int main()

```

```

{
    /* Let us create following BST

```



```

Node *root = NULL;
root = insert(root, 50);

```

```

        insert(root, 30);
        insert(root, 20);
        insert(root, 40);
        insert(root, 70);
        insert(root, 60);
        insert(root, 80);

        int c = 0;
        for (int k=1; k<=7; k++)
            kthLargest(root, k);

    }
    return 0;
}

```

```

K'th largest element is 80
K'th largest element is 70
K'th largest element is 60
K'th largest element is 50
K'th largest element is 40
K'th largest element is 30
K'th largest element is 20

```

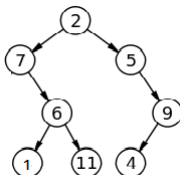
Time complexity: The code first traverses down to the rightmost node which takes $O(h)$ time, then traverses k elements in $O(k)$ time. Therefore overall time complexity is $O(h + k)$.

This article is contributed by **Chirag Sharma**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

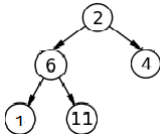
149. Given a binary tree, how do you remove all the half nodes?

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.



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

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 C implementation of this idea.

```

// C program to remove all half nodes
#include <stdio.h>
#include <stdlib.h>

struct node
{
    int data;
    struct node* left, *right;
};

struct node* newNode(int data)
{
    struct node* node = (struct node*)
        malloc(sizeof(struct node));
    node->data = data;
    node->left = node->right = NULL;
    return(node);
}

void printInoder(struct node*root)
{
    if (root != NULL)
    {
        printInoder(root->left);
        printf("%d ", root->data);
        printInoder(root->right);
    }
}

// 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;
}

// Driver program
int main(void)
{
    struct node *NewRoot=NULL;
    struct node *root = newNode(2);
    root->left      = newNode(7);
    root->right     = newNode(5);
    root->left->right = newNode(6);
    root->left->right->left=newNode(1);
    root->left->right->right=newNode(11);
    root->right->right=newNode(9);
    root->right->right->left=newNode(4);

    printf("Inorder traversal of given tree \n");
    printInoder(root);

    NewRoot = RemoveHalfNodes(root);

    printf("\nInorder traversal of the modified tree \n");
    printInoder(NewRoot);
    return 0;
}

```

Output:

```

Inorder traversal of given tree
7 1 6 11 2 5 4 9
Inorder traversal of the modified tree
1 6 11 2 4

```

Time complexity of the above solution is $O(n)$ as it does a simple traversal of binary tree.

This article is contributed by **Jyoti Saini**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

150. Advantages of BST over Hash Table

Hash Table supports following operations in $\Theta(1)$ 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 $O(\text{Log}n)$.

So Hash Table seems to be 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.

1. 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.
2. 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.
3. 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.
4. With BSTs, all operations are guaranteed to work in $O(\text{Log}n)$ time. But with Hashing, $\Theta(1)$ is average time and some particular operations may be costly, especially when table resizing happens.

This article is contributed by **Himanshu Gupta**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

151. Handshaking Lemma and Interesting Tree Properties

What is Handshaking Lemma?

Handshaking lemma is about undirected graph. In every finite undirected graph number of vertices with odd degree is always even. The handshaking lemma is a consequence of the degree sum formula (also sometimes called the handshaking lemma)

How is Handshaking Lemma useful in Tree Data structure?

Following are some interesting facts that can be proved using Handshaking lemma.

1) In a k -ary tree where every node has either 0 or k children, following property is always true.

$$L = (k - 1) * I + 1$$

Where L = Number of leaf nodes

I = Number of internal nodes

Proof:

Proof can be divided in two cases.

Case 1 (Root is Leaf): There is only one node in tree. The above formula is true for single node as $L = 1, I = 0$.

Case 2 (Root is Internal Node): For trees with more than 1 nodes, root is always internal node. The above formula can be proved using Handshaking Lemma for this case. A tree is an undirected acyclic graph.

Total number of edges in Tree is number of nodes minus 1, i.e., $|E| = L + I - 1$.

All internal nodes except root in the given type of tree have degree $k + 1$. Root has degree k . All leaves have degree 1. Applying the Handshaking lemma to such trees, we get following relation.

$$\text{Sum of all degrees} = 2 * (\text{Sum of Edges})$$

$$\begin{aligned} &\text{Sum of degrees of leaves} + \\ &\text{Sum of degrees for Internal Node except root} + \\ &\text{Root's degree} = 2 * (\text{No. of nodes} - 1) \end{aligned}$$

Putting values of above terms,

$$L + (I-1)*(k+1) + k = 2 * (L + I - 1)$$

$$L + k*I - k + I - 1 + k = 2*L + 2I - 2$$

$$L + K*I + I - 1 = 2*L + 2*I - 2$$

$$K*I + 1 - I = L$$

$$(K-1)*I + 1 = L$$

So the above property is proved using Handshaking Lemma, let us discuss one more interesting property.

2) 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

Proof:

Let number of nodes with 2 children be T. Proof can be divided in three cases.

Case 1: There is only one node, the relationship holds as $T = 0, L = 1$.

Case 2: Root has two children, i.e., degree of root is 2.

Sum of degrees of nodes with two children except root +
Sum of degrees of nodes with one child +
Sum of degrees of leaves + Root's degree = $2 * (\text{No. of Nodes} - 1)$

Putting values of above terms,
 $(T-1)*3 + S*2 + L + 2 = (S + T + L - 1)*2$

Cancelling 2S from both sides.
 $(T-1)*3 + L + 2 = (S + L - 1)*2$
 $T - 1 = L - 2$
 $T = L - 1$

Case 3: Root has one child, i.e., degree of root is 1.

Sum of degrees of nodes with two children +
Sum of degrees of nodes with one child except root +
Sum of degrees of leaves + Root's degree = $2 * (\text{No. of Nodes} - 1)$

Putting values of above terms,
 $T*3 + (S-1)*2 + L + 1 = (S + T + L - 1)*2$

Cancelling 2S from both sides.
 $3*T + L - 1 = 2*T + 2*L - 2$
 $T - 1 = L - 2$
 $T = L - 1$

Therefore, in all three cases, we get $T = L-1$.

We have discussed proof of two important properties of Trees using Handshaking Lemma. Many GATE questions have been asked on these properties, following are few links.

GATE-CS-2015 (Set 2) | Question 20

GATE-CS-2005 | Question 36

GATE-CS-2002 | Question 34

GATE-CS-2007 | Question 43

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above