Contents

1 Handshaking Lemma and Interesting Tree Properties **8**

Source 10

2 Applications of tree data structure **11**

Source 12

3 Tree Traversals **13**

Source 17

4 Write a C program to Calculate Size of a tree **18**

Source 20

5 Write C Code to Determine if Two Trees are Identical **21**

Source 23

6 Write a C Program to Find the Maximum Depth or Height of a Tree **24**

Source 26

7 Write a C program to Delete a Tree. **27**

Source 30

8 Write an Efficient C Function to Convert a Binary Tree into its Mirror Tree **31**

Source 34

9 If you are given two traversal sequences, can you construct the binary tree? **35**

Source 36

10 Given a binary tree, print out all of its root-to-leaf paths one per line. **37**

Source 40

11 The Great Tree-List Recursion Problem. **41**

Source 41

12 Level Order Tree Traversal **42**

Source 47

13 Program to count leaf nodes in a binary tree **48**

Source 50

14 Level order traversal in spiral form **51**

Source 56

15 Check for Children Sum Property in a Binary Tree. **57**

Source 59

16 Convert an arbitrary Binary Tree to a tree that holds Children Sum Property **60**

Source 64

17 Diameter of a Binary Tree **65**

Source 68

18 How to determine if a binary tree is height-balanced? **69**

Source 73

19 Inorder Tree Traversal without Recursion **74**

Source 78

20 Inorder Tree Traversal without recursion and without stack! **79**

Source 81

21 Root to leaf path sum equal to a given number **82**

Source 84

22 Construct Tree from given Inorder and Preorder traversals **85**

Source 88

23 Given a binary tree, print all root-to-leaf paths **89**

Source 91

24 Double Tree **92**

Source 95

25 Maximum width of a binary tree **96**

Source 101

26 Foldable Binary Trees **103**

Source 109

27 Print nodes at k distance from root **110**

Source 111

28 Get Level of a node in a Binary Tree **112**

Source 114

29 Print Ancestors of a given node in Binary Tree **115**

Source 117

30 Check if a given Binary Tree is SumTree **118**

Source 122

31 Check if a binary tree is subtree of another binary tree | Set 1 **123**

Source 126

32 Connect nodes at same level **127**

Source 130

33 Connect nodes at same level using constant extra space **131**

Source 136

34 Populate Inorder Successor for all nodes **137**

Source 140

35 Convert a given tree to its Sum Tree **141**

Source 143

36 Vertical Sum in a given Binary Tree **144**

Source 146

37 Find the maximum sum leaf to root path in a Binary Tree **147**

Source 150

38 Construct Special Binary Tree from given Inorder traversal **151**

Source 154

39 Construct a special tree from given preorder traversal **155**

Source 157

40 Check whether a given Binary Tree is Complete or not | Set 1 (Iterative Solution) 159

Source 163

41 Boundary Traversal of binary tree 164

Source 167

42 Construct Full Binary Tree from given preorder and postorder traversals 168

Source 171

43 Iterative Preorder Traversal 172

Source 174

44 Morris traversal for Preorder 175

Source 177

45 Linked complete binary tree & its creation 178

Source 182

46 Ternary Search Tree 183

Source 187

47 Segment Tree | Set 1 (Sum of given range) 188

Source 196

48 Dynamic Programming | Set 26 (Largest Independent Set Problem) 197

Source 201

49 Iterative Postorder Traversal | Set 1 (Using Two Stacks) 202

Source 206

50 Iterative Postorder Traversal | Set 2 (Using One Stack) 207

Source 211

51 Reverse Level Order Traversal 212

Source 216

52 Construct Complete Binary Tree from its Linked List Representation 217

Source 220

53 Convert a given Binary Tree to Doubly Linked List | Set 1 221

Source 224

54 Tree Isomorphism Problem **225**

Source . . . 227

55 Find all possible interpretations of an array of digits **228**

Source . . . 232

56 Iterative Method to find Height of Binary Tree **233**

Source . . . 235

57 Custom Tree Problem **237**

Source . . . 241

58 Convert a given Binary Tree to Doubly Linked List | Set 2 **242**

Source . . . 245

59 Print ancestors of a given binary tree node without recursion **246**

Source . . . 250

60 Difference between sums of odd level and even level nodes of a Binary Tree **251**

Source . . . 253

61 Print Postorder traversal from given Inorder and Preorder traversals **254**

Source . . . 256

62 Find depth of the deepest odd level leaf node **257**

Source . . . 259

63 Check if all leaves are at same level **260**

Source . . . 262

64 Print Left View of a Binary Tree **263**

Source . . . 264

65 Remove all nodes which don't lie in any path with sum>= k **265**

Source . . . 270

66 Extract Leaves of a Binary Tree in a Doubly Linked List **271**

Source . . . 274

67 Deepest left leaf node in a binary tree **275**

Source . . . 277

68 Find next right node of a given key **278**

Source 281

69 Sum of all the numbers that are formed from root to leaf paths **282**

Source 284

70 Convert a given Binary Tree to Doubly Linked List | Set 3 **285**

Source 287

71 Lowest Common Ancestor in a Binary Tree | Set 1 **288**

Source 295

72 Find distance between two given keys of a Binary Tree **296**

Source 299

73 Print all nodes that are at distance k from a leaf node **300**

Source 302

74 Check if a given Binary Tree is height balanced like a Red-Black Tree **303**

Source 305

75 Print all nodes at distance k from a given node **306**

Source 309

76 Print a Binary Tree in Vertical Order | Set 1 **310**

Source 313

77 Construct a tree from Inorder and Level order traversals **314**

Source 318

78 Find the maximum path sum between two leaves of a binary tree **319**

Source 322

79 Reverse alternate levels of a perfect binary tree **323**

Source 326

80 Check if two nodes are cousins in a Binary Tree **328**

Source 330

81 Check if a binary tree is subtree of another binary tree | Set 2 **331**

Source 335

82 Serialize and Deserialize a Binary Tree **336**
Source . . . 341

83 Print nodes between two given level numbers of a binary tree **342**
Source . . . 345

84 Find the closest leaf in a Binary Tree **346**
Source . . . 349

85 Convert a Binary Tree to Threaded binary tree **350**
Source . . . 353

86 Print Nodes in Top View of Binary Tree **354**
Source . . . 357

87 Bottom View of a Binary Tree **358**
Source . . . 361

88 Perfect Binary Tree Specific Level Order Traversal **362**
Source . . . 365

89 Minimum no. of iterations to pass information to all nodes in the tree **366**
Source . . . 373

90 Clone a Binary Tree with Random Pointers **374**
Source . . . 381

91 Given a binary tree, how do you remove all the half nodes? **383**
Source . . . 385

92 Vertex Cover Problem | Set 2 (Dynamic Programming Solution for Tree) **386**
Source . . . 390

93 Check whether a binary tree is a full binary tree or not **391**
Source . . . 394

94 Find sum of all left leaves in a given Binary Tree **395**
Source . . . 399

95 Remove nodes on root to leaf paths of length **400**
Source . . . 403

96 Maximum Path Sum in a Binary Tree **404**
Source . . . 407

Chapter 1

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.

```
Sum of all degrees  = 2 * (Sum of Edges)
```

```
Sum of degrees of leaves +
Sum of degrees for Internal Node except root +
Root's degree = 2 * (No. of nodes - 1)

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 * (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 * (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 3) | Question 35

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

Source

http://www.geeksforgeeks.org/handshaking-lemma-and-interesting-tree-properties/

Chapter 2

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


```
    /
2)  If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in mode
3)  We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists).
4)   Like Linked Lists and unlike Arrays, Pointer implementation of trees don't have an upper limit on number
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 di
```

Source

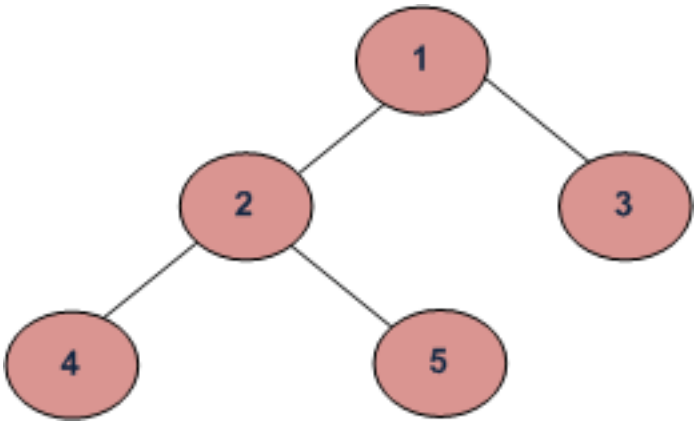
http://www.geeksforgeeks.org/applications-of-tree-data-structure/

Category: Trees

Chapter 3

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.



Example Tree

Depth First Traversals:
(a) Inorder
(b) Preorder
(c) Postorder

Breadth First or Level Order Traversal
Please see thispost 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 itraversal s 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 sutree */
     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) = (-)(n) (Theta of n)

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

T(n) = 2T(|_n/2_|) + c

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

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

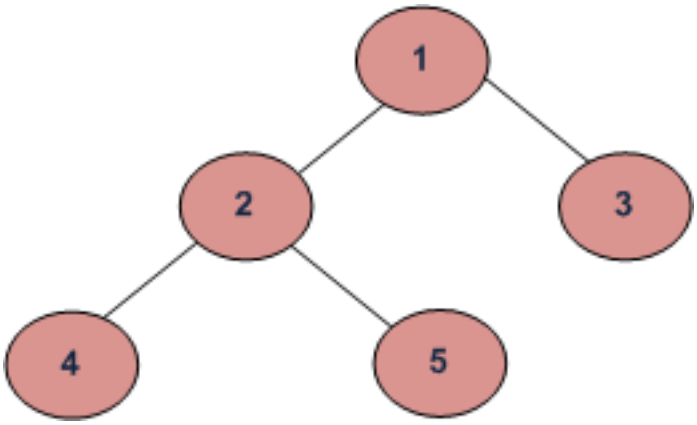
Source

http://www.geeksforgeeks.org/618/

Chapter 4

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.



Example Tree

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

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

Algorithm:

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

```
#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)

Source

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

Category: Trees Tags: Size of a Tree, Tree Size, TreeSize

Chapter 5

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

Source

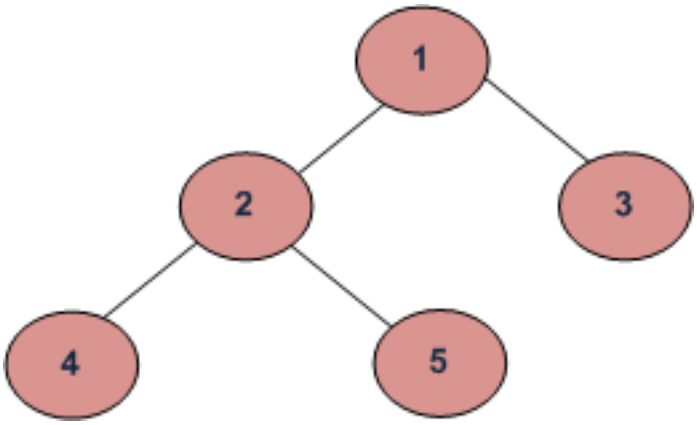
http://www.geeksforgeeks.org/write-c-code-to-determine-if-two-trees-are-identical/

Category: Trees Tags: Tree Traveral, Trees

Chapter 6

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

Maximum depth or height of the below tree is 3.



Example Tree

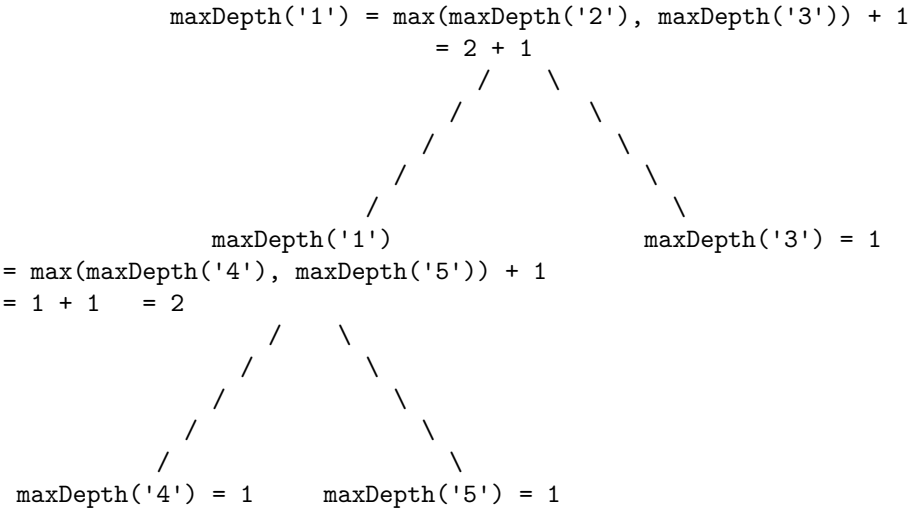
Recursively calculate height of left and right subtrees of a node and assign height to the node as max of the heights of two children plus 1. See below pseudo code and program for details.

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 Traversalfor details)

References:
http://cslibrary.stanford.edu/110/BinaryTrees.html

Source

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

Category: Trees Tags: Height of a Tree, Tree Traveral, Trees

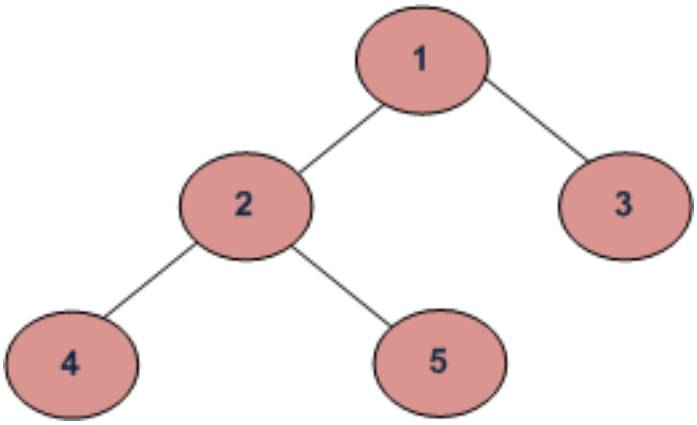
Chapter 7

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



Example Tree

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)

Source

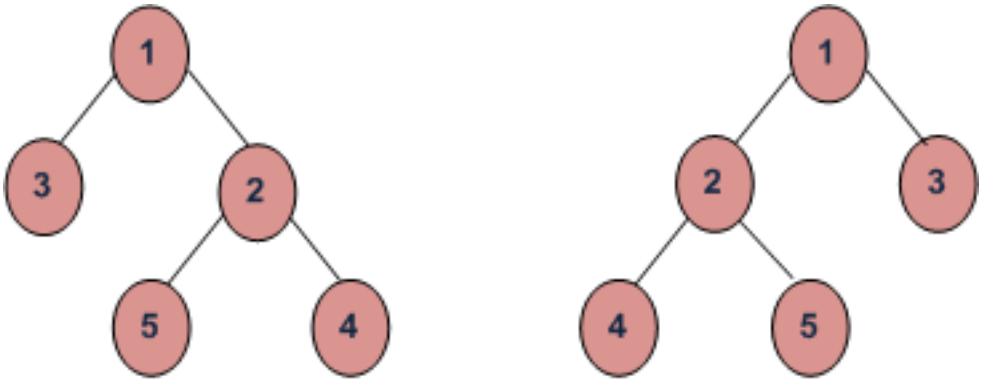
http://www.geeksforgeeks.org/write-a-c-program-to-delete-a-tree/

Category: Trees Tags: Delete Tree, Tree Traveral, Trees

Chapter 8

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));
  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...
       4
      / \
     2   5
    / \
   1   3

 is changed to...
       4
      / \
     5   2
        / \
       3   1
*/
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);

  getchar();
  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)

Source

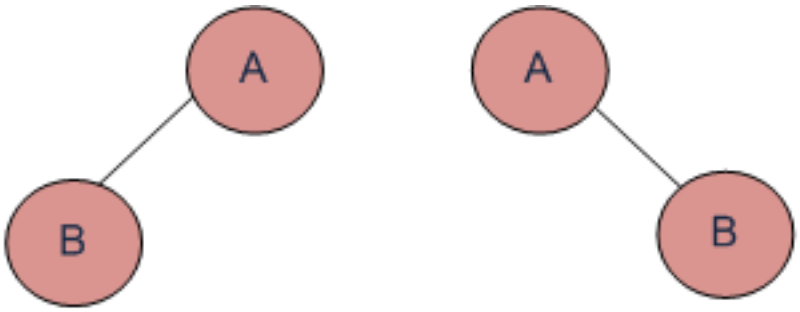
http://www.geeksforgeeks.org/write-an-efficient-c-function-to-convert-a-tree-into-its-mirror-tree/

Category: Trees Tags: Convert to Mirror, Get the Mirror, Mirror Tree, Tree Traveral, Trees

Chapter 9

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.

Source

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

Category: Trees Tags: Binary Tree, Tree Traveral

Chapter 10

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

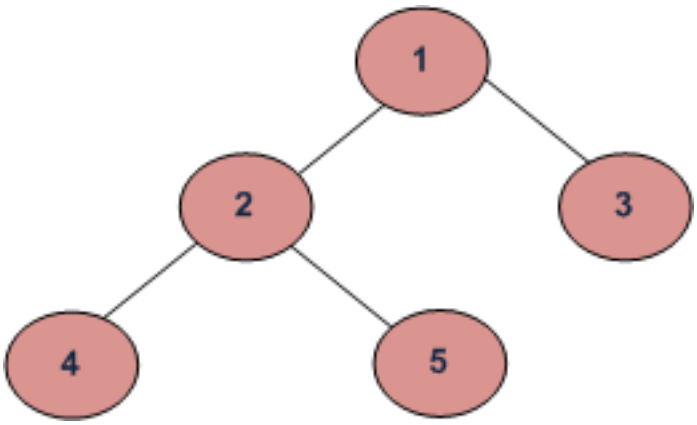
Asked by Varun Bhatia

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:



Example Tree

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

Source

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

Category: Trees

Post navigation

← Write a program to add two numbers in base 14 Lowest Common Ancestor in a Binary Search Tree. →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 11

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

Source

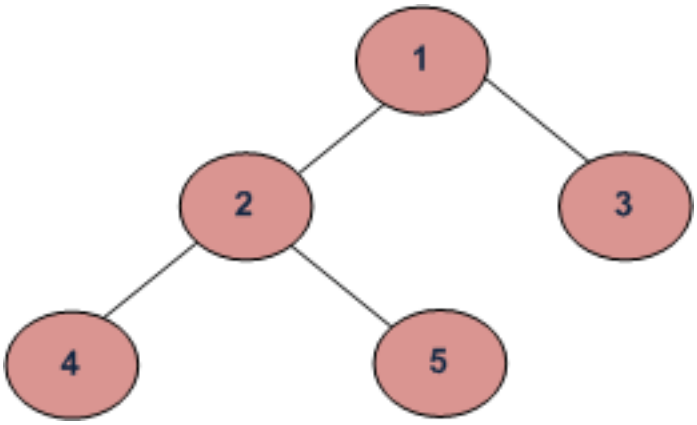
http://www.geeksforgeeks.org/the-great-tree-list-recursion-problem/

Category: Linked Lists Trees

Chapter 12

Level Order Tree Traversal

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



Example Tree

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

METHOD 1 (Use function to print a given level)

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

```
 /*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 protoypes*/
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) + .. + O(1) which is O(n^2).

METHOD 2 (Use Queue)

Algorithm:
For each node, first the node is visited and then it's child nodes are put in a FIFO queue.

```
 printLevelorder(tree)
1) Create an empty queue q
2) temp_node = root /*start from root*/
3) Loop while temp_node is not NULL
    a) print temp_node->data.
    b) Enqueue temp_node's children (first left then right children) to q
    c) Dequeue a node from q and assign it's value to temp_node
```

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;
};

/* frunction 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;

  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.

Source

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

Category: Trees

Post navigation

← Compute the minimum or maximum of two integers without branching Program to count leaf nodes in a binary tree →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

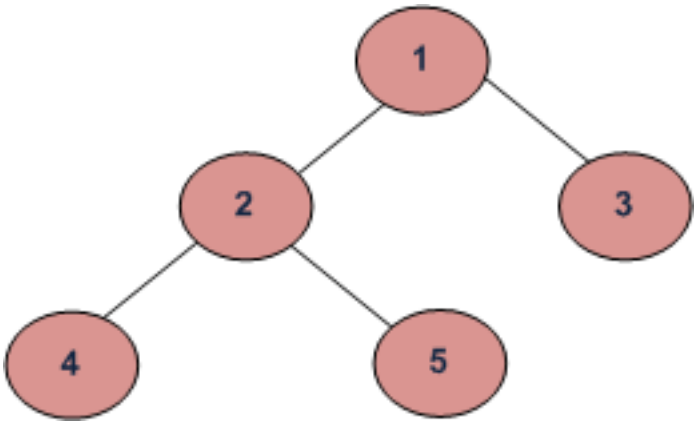
Chapter 13

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
```



Example Tree

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.

Source

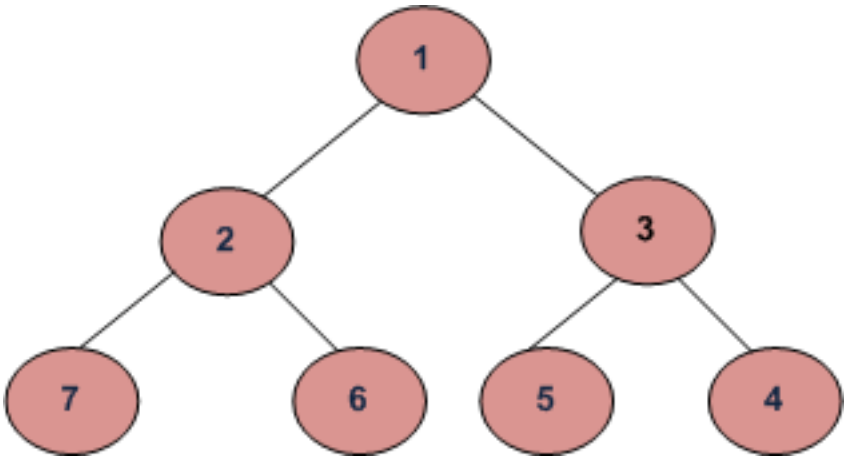
http://www.geeksforgeeks.org/write-a-c-program-to-get-count-of-leaf-nodes-in-a-binary-tree/

Category: Trees

Chapter 14

Level order traversal in spiral form

Write a function to print spiral order traversal of a tree. For below tree, function should print 1, 2, 3, 4, 5, 6, 7.



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

Function to print level order traversal of tree

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

Function to print all nodes at a given level

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

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

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

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

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

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

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

// A utility functiont 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.

Source

http://www.geeksforgeeks.org/level-order-traversal-in-spiral-form/

Category: Trees

Post navigation

← Babylonian method for square root Data Structures and Algorithms | Set 7 →

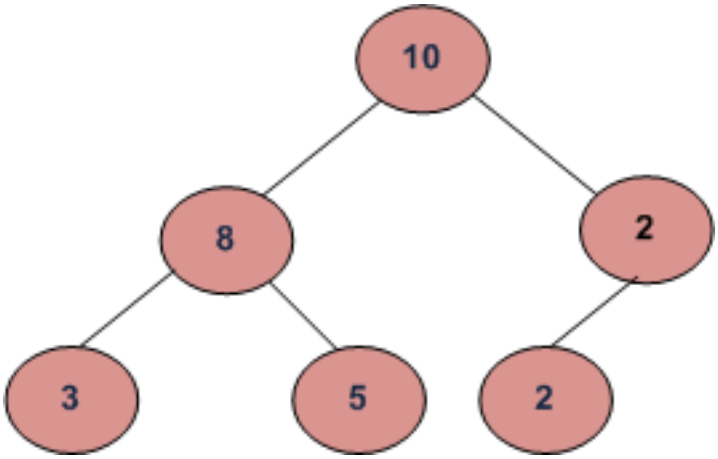
Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 15

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 data*/
  int left_data = 0,  right_data = 0;

  /* If node is NULL or it's a leaf node then
     return true */
  if(node == NULL ||
     (node->left == NULL && node->right == NULL))
    return 1;
  else
  {
    /* If left child is not present then 0 is used
       as data of left child */
    if(node->left != NULL)
      left_data = node->left->data;

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

    /* if the node and both of its children satisfy the
       property return 1 else 0*/
    if((node->data == left_data + right_data)&&
        isSumProperty(node->left) &&
        isSumProperty(node->right))
      return 1;
    else
      return 0;
  }
}

/*
 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.

Source

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

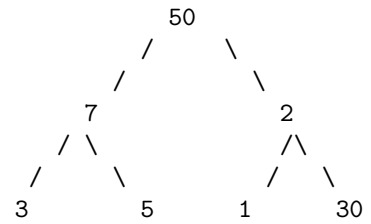
Category: Trees

Chapter 16

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.

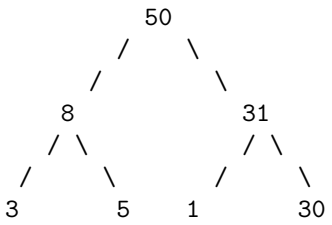
```
diff = node's children sum - node's data
```

If diff is 0 then nothing needs to be done.

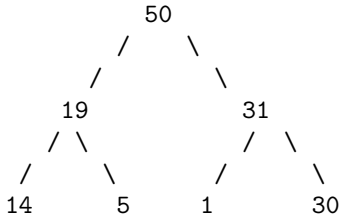
If diff > 0 (node's data is smaller than node's children sum) increment the node's data by diff.

If diff 50 / \ / \ **8** 2 / \ /\ / \ / \ 3 5 1 30

Then convert the right subtree (increment 2 to 31)



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



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

Implementation:

```
/* Program to convert an aribitary 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 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 */
    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.

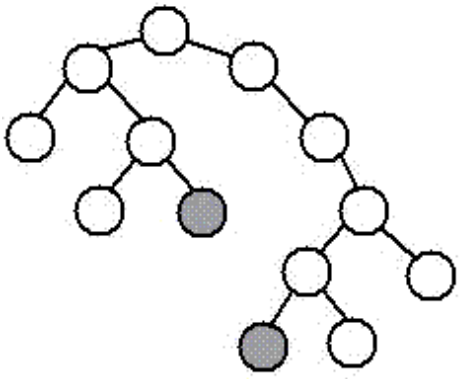
Source

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

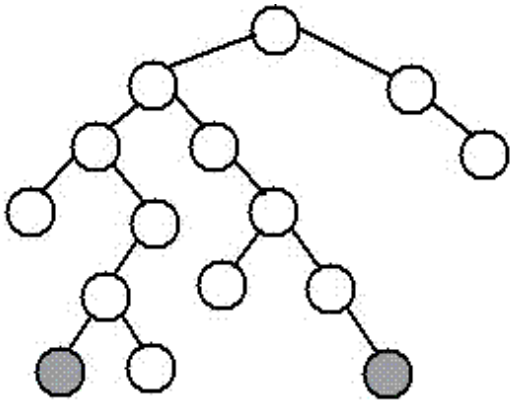
Chapter 17

Diameter of a Binary Tree

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



diameter, 9 nodes, through root



diameter, 9 nodes, NOT through root

The diameter of a tree T is the largest of the following quantities:

* the diameter of T's left subtree
* the diameter of T's right subtree
* the longest path between leaves that goes through the root of T (this can be computed from the heights of the subtrees of T)

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);

/* 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 f 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 ldiameter */
  ldiameter = diameterOpt(root->left, &lh);
  rdiameter = diameterOpt(root->right, &rh);

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

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

Time Complexity: O(n)

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.

Source

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

Category: Trees

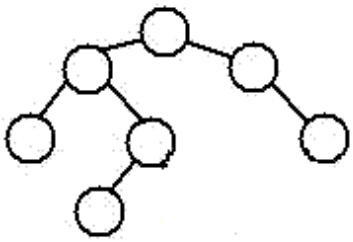
Chapter 18

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

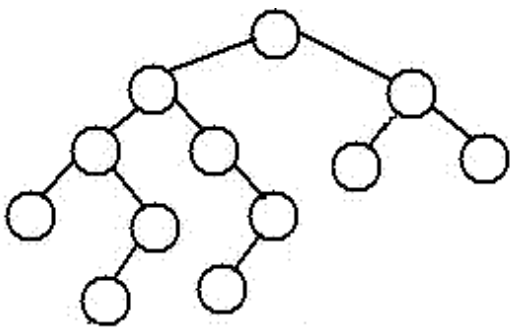
A tree where no leaf is much farther away from the root than any other leaf. Different balancing schemes allow different definitions of "much farther" and different amounts of work to keep them balanced.

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;

   /* 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))
    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.

Source

http://www.geeksforgeeks.org/how-to-determine-if-a-binary-tree-is-balanced/

Category: Trees

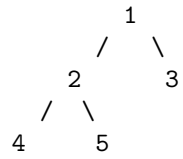
Chapter 19

Inorder Tree Traversal without Recursion

Using Stackis 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%20Traversal.pps

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.

Source

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

Category: Trees Tags: Tree Traveral

Chapter 20

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 predecssor */
      else
      {
        pre->right = NULL;
        printf(" %d ",current->data);
        current = current->right;
      } /* End of if condition pre->right == NULL */
    } /* End of if condition current->left == NULL*/
  } /* End of while */
}

/* 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;

  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);

  getchar();
  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.

Source

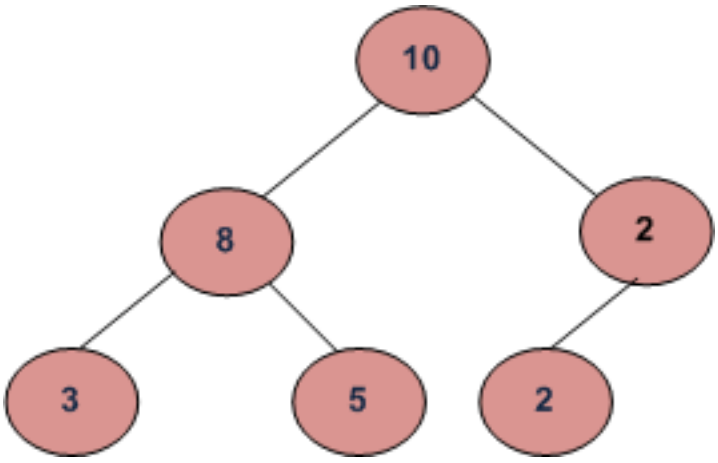
http://www.geeksforgeeks.org/inorder-tree-traversal-without-recursion-and-without-stack/

Category: Trees Tags: Tree Traveral

Chapter 21

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

Source

http://www.geeksforgeeks.org/root-to-leaf-path-sum-equal-to-a-given-number/

Category: Trees

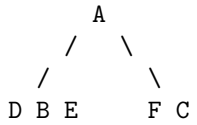
Chapter 22

Construct Tree from given Inorder and Preorder traversals

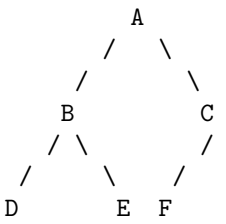
Let us consider the below traversals:

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

  if(inStrt > inEnd)
     return NULL;

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

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

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

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

  return tNode;
}

/* 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 funtcion 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.

Source

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

Category: Trees Tags: Inorder Traversal, Preorder Traversal, Tree Traveral

Post navigation

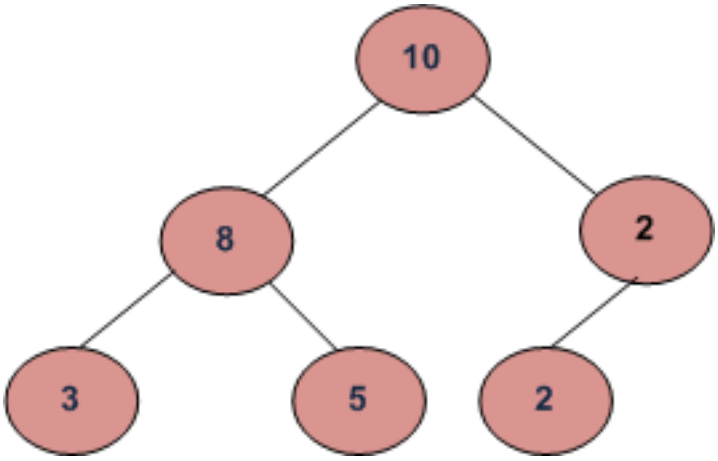
← Do not use sizeof for array parameters Given a binary tree, print all root-to-leaf paths →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 23

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);
  }
}


/* 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.

Source

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

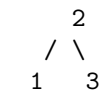
Category: Trees

Chapter 24

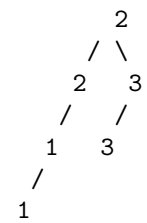
Double Tree

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.

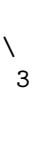
So the tree…



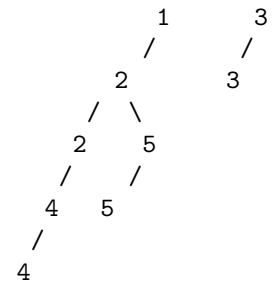
is changed to…



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);
  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.

Source

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

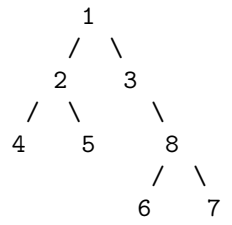
Category: Trees

Chapter 25

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.



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 protoypes*/
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));
  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 bunary tree is:
          1
```

```
        /  \
       2    3
     /  \     \
    4   5     8
              /  \
             6   7
  */
  printf("Maximum width is %d \n", getMaxWidth(root));
  getchar();
  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, DivyaCand 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 Rajaand jagdishfor 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.

Source

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

Category: Trees

Post navigation

← Run Length Encoding Intersection of two Sorted Linked Lists →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 26

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)
       10
     /    \
    7      15
     \    /
      9  11

(b)
        10
       /  \
      7    15
     /      \
    9       11

(c)
        10
       /  \
      7   15
     /    /
    5   11

(d)

         10
       /   \
      7     15
    /  \    /
   9   10  12
```

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 ajaymfor 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 mrror 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 originial 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* 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.

Source

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

Category: Trees

Post navigation

← Reverse a Linked List in groups of given size Practice Questions for Recursion | Set 3 →

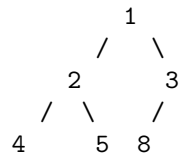
Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 27

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.

Source

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

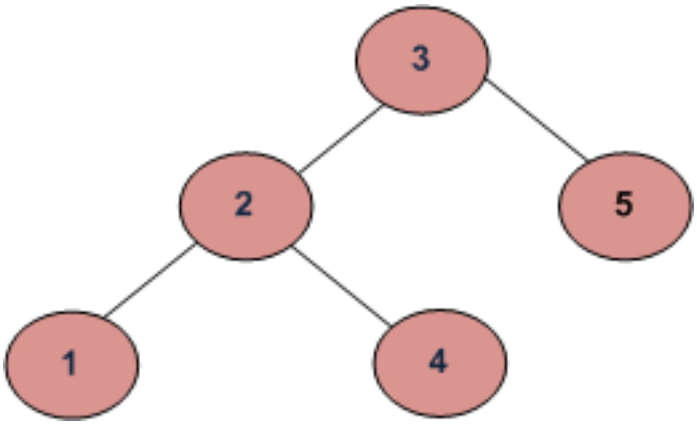
Category: Trees

Chapter 28

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 prandeeyfor 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 is
   present in tree, otherwise returns 0.*/
int getLevelUtil(struct node *node, int data, int level)
{
```

```
    if (node == NULL)
        return 0;

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

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

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

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

/* 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.

Source

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

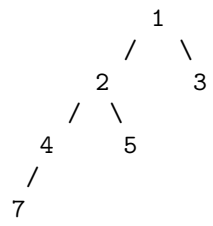
Category: Trees

Chapter 29

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 wgpshashankfor 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);

  getchar();
  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.

Source

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

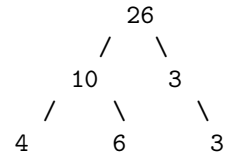
Category: Trees

Chapter 30

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;
};

/* Utillity 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
    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.

Source

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

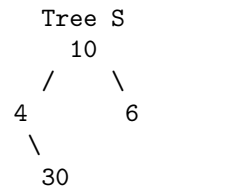
Category: Trees

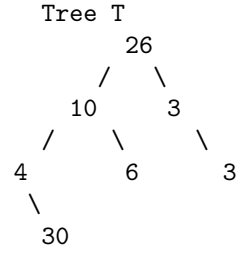
Chapter 31

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.

Source

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

Category: Trees

Chapter 32

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 compete binary tree */
void connectRecur(struct node* p)
{
  // Base case
  if (!p)
    return;

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

  // Set the nextRight pointer for p's right child
```

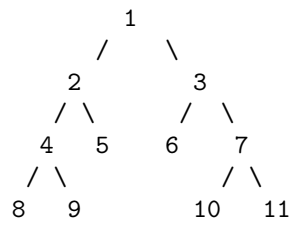
```
  // p->nextRight will be NULL if p is the right most child at its level
  if (p->right)
    p->right->nextRight = (p->nextRight)? p->nextRight->left: NULL;

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

/* 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);
  printf("nextRight of %d is %d \n", root->left->left->data,
        root->left->left->nextRight? root->left->left->nextRight->data: -1);
```

```
  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 postfor more solutions.

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

Source

http://www.geeksforgeeks.org/connect-nodes-at-same-level/

Chapter 33

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

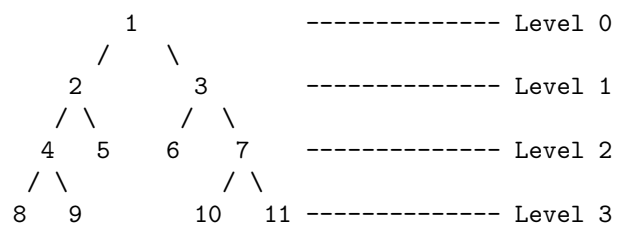
```
Input Tree
       A
      / \
     B   C
    / \   \
   D   E   F

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

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 *p)
{
    // Set the nextRight for root
    p->nextRight = NULL;

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

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

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

    /* Set the nextRight pointer for p's left child */
    if (p->left)
    {
       if (p->right)
```

```
        {
            p->left->nextRight = p->right;
            p->right->nextRight = getNextRight(p);
        }
        else
            p->left->nextRight = getNextRight(p);

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

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

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

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

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

An Iterative Solution

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

```
#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 as p.
   This function is used to getNExt right of p's right child
   If right child of is NULL then this can also be sued 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 childrem nodes of p and children nodes of all other nodes
          at same level as p */
        while (q != NULL)
        {
            // Set the nextRight pointer for p's left child
            if (q->left)
```

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

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

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

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

/* 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            90
    */
    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.

Source

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

Category: Trees

Post navigation

← Connect nodes at same level Find the maximum element in an array which is first increasing and then decreasing →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 34

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 Inorder */
void populateNext(struct node* p)
{
```

```
    // The first visited node will be the rightmost node
    // next of the rightmost node will be NULL
    static struct node *next = NULL;

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

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

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

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

/* 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 paramater.

```
// 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.

Source

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

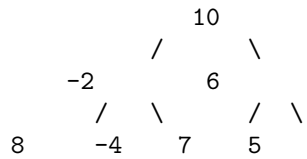
Category: Trees

Chapter 35

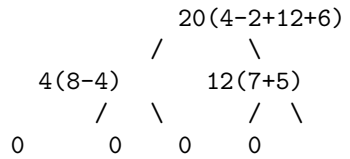
Convert a given tree to its Sum Tree

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 values 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 as
    // 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 toSumTree()
  printf("Inorder Traversal of the resultant tree is: \n");
  printInorder(root);

  getchar();
  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 thisforum 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.

Source

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

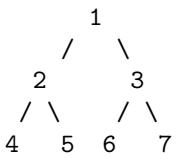
Category: Trees

Chapter 36

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.

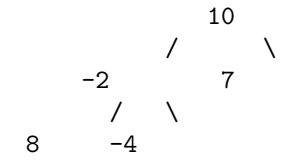
Source

http://www.geeksforgeeks.org/vertical-sum-in-a-given-binary-tree/

Chapter 37

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) )
    {
        printf("%d ", root->data);
        return true;
    }

    return false;
}

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

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

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

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

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

    struct node *target_leaf;
    int max_sum = INT_MIN;

    // find the target leaf and maximum sum
```

```
    getTargetLeaf (node, &max_sum, 0, &target_leaf);

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

    return max_sum;  // return maximum sum
}

/* 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.

Source

http://www.geeksforgeeks.org/find-the-maximum-sum-path-in-a-binary-tree/

Category: Trees

Chapter 38

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
         40
       /   \
      10     30
     /         \
    5          28
```

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.

```
         40
       /    \
  {1,5,10}   {30,15,28,20}
```

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

```
         40
       /   \
      10     30
     /         \
    5          28
```

```
    /          /  \
   1         15    20
```

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.

```
/* 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 subtress */
    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 funtcion 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 */

    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 Insorder 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

Source

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

Category: Trees

Post navigation

← Database Management Systems | Set 11 Multiline macros in C →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 39

Construct a special tree from given preorder traversal

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

Source: Amazon Interview Question

Example:

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

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

```
/* A 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 n)
{
    int index = *index_ptr; // store the current value of index in pre[]

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

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

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

    return temp;
}

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

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


/* 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
          10
         /  \
        30   15
       /  \
      20   5 */
    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 Tree: \n");
    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

Source

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

Category: Trees

Post navigation

← Sieve of Eratosthenes Playing with Destructors in C++ →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 40

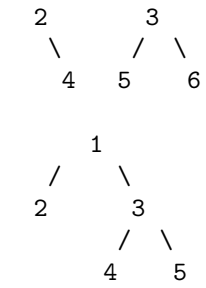
Check whether a given Binary Tree is Complete or not | Set 1 (Iterative Solution)

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.

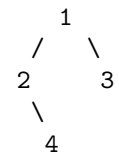
```
The following trees are examples of Complete Binary Trees
    1
  /   \
 2     3

       1
    /    \
   2       3
  /
 4

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

```
The following trees are examples of Non-Complete Binary Trees
    1
      \
       3

       1
    /    \
```



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;
};

/* frunction prototypes for functions needed for Queue data
   structure. A queue is needed for level order tarversal */
struct node** createQueue(int *, int *);
void enQueue(struct node **, int *, struct node *);
struct node *deQueue(struct node **, int *);
bool isQueueEmpty(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(!isQueueEmpty(&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 isQueueEmpty(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.

Source

http://www.geeksforgeeks.org/check-if-a-given-binary-tree-is-complete-tree-or-not/

Category: Trees

Post navigation

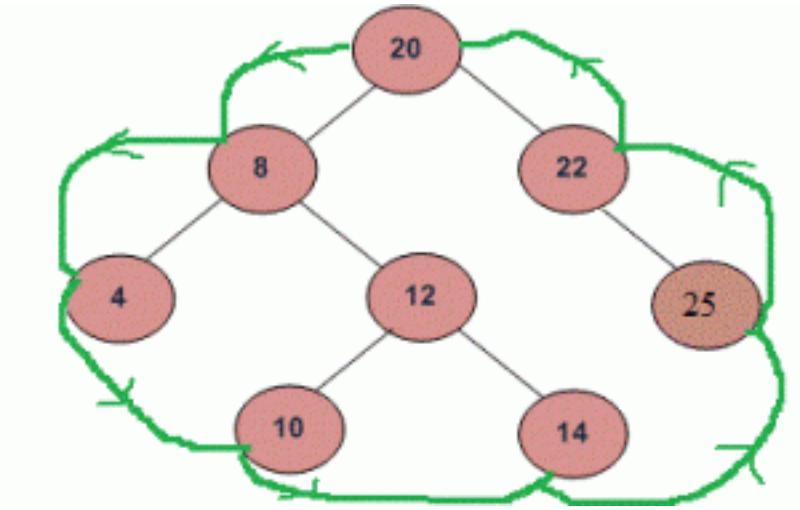
← Microsoft Interview | Set 3 Amazon Interview | Set 4 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 41

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 boundry 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

Source

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

Category: Trees

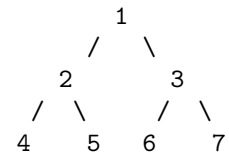
Chapter 42

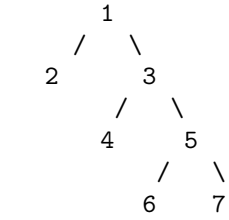
Construct Full Binary Tree from given preorder and postorder traversals

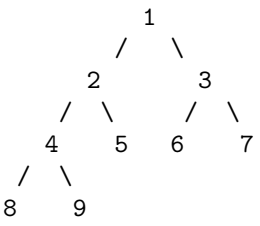
Given two arrays that represent preorder and postorder traversals of a full binary tree, construct the binary tree.

A **Full Binary Tree** is a binary tree where every node has either 0 or 2 children

Following are examples of Full Trees.

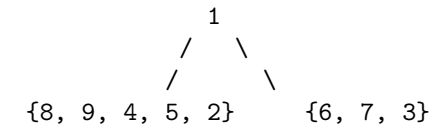




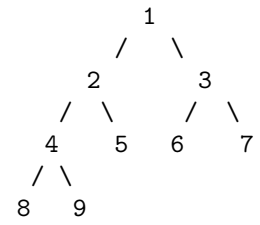


It is not possible to construct a general Binary Tree from preorder and postorder traversals (See this). But if know that the Binary Tree is Full, we can construct the tree without ambiguity. Let us understand this with the help of following example.

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) );

    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 at
    // preIndex from preorder and make it root, and increment preIndex
    struct node* root = newNode ( pre[*preIndex] );
    ++*preIndex;

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

    // Search the next element of pre[] in post[]
    int i;
    for (i = l; i <= h; ++i)
        if (pre[*preIndex] == post[i])
            break;

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

    return root;
}

// The main function to construct Full Binary Tree from given preorder and
// postorder traversals. This function mainly uses constructTreeUtil()
struct node *constructTree (int pre[], int post[], int size)
{
    int preIndex = 0;
    return constructTreeUtil (pre, post, &preIndex, 0, size - 1, size);
}

// 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.

Source

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

Category: Trees

Post navigation

← Adobe Interview | Set 2 Lexicographic rank of a string →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 43

Iterative Preorder Traversal

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

Refer thisfor 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(node *root)
{
    // Base Case
    if (root == NULL)
       return;

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

    /* Pop all items one by one. Do following for every popped item
       a) print it
       b) push its right child
       c) push its left child
    Note that right child is pushed first so that left is processed first */
    while (nodeStack.empty() == false)
    {
        // Pop the top item from stack and print it
        struct node *node = nodeStack.top();
        printf ("%d ", node->data);
        nodeStack.pop();

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

// 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

Source

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

Category: Trees

Chapter 44

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 points to
            // this node
            if (current->right == root)
            {
                current->right = NULL;
                root = root->right;
            }

            // If right child doesn't point to this node, then print this
            // node and make right child point to this node
            else
            {
                printf("%d ", root->data);
                current->right = root;
                root = root->left;
            }
        }
    }
}

// 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.

Source

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

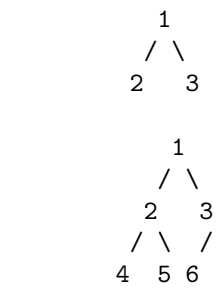
Category: Trees

Chapter 45

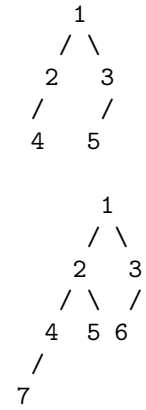
Linked complete binary tree & its creation

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:



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 created 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 node* ));

    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 children
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))
    {
```

```
        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.

Source

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

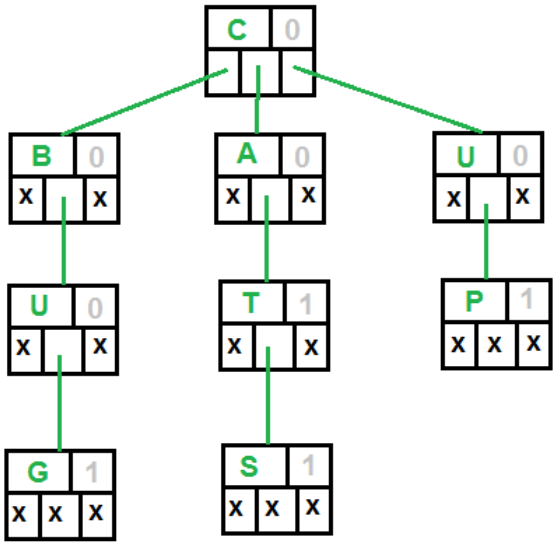
Chapter 46

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?



Ternary Search Tree for CAT, BUG, CATS, UP

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, travese
// and search operations
#include <stdio.h>
#include <stdlib.h>
#define MAX 50

// A node of ternary search tree
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 greate 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 respectively\n");
    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.

Source

http://www.geeksforgeeks.org/ternary-search-tree/

Category: Trees Tags: Advance Data Structures, Advanced Data Structures

Post navigation

← [TopTalent.in] Exclusive Interview with Ravi Kiran from BITS, Pilani who got placed in Google, Microsoft and Facebook Amazon Interview | Set 17 →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 47

Segment Tree | Set 1 (Sum of given range)

Let us consider the following problem to understand Segment Trees.

We have an array arr[0 . . . n-1]. We should be able to
1 Find the sum of elements from index l to r where 0 2 Change value of a specified element of the array arr[i] = x where 0

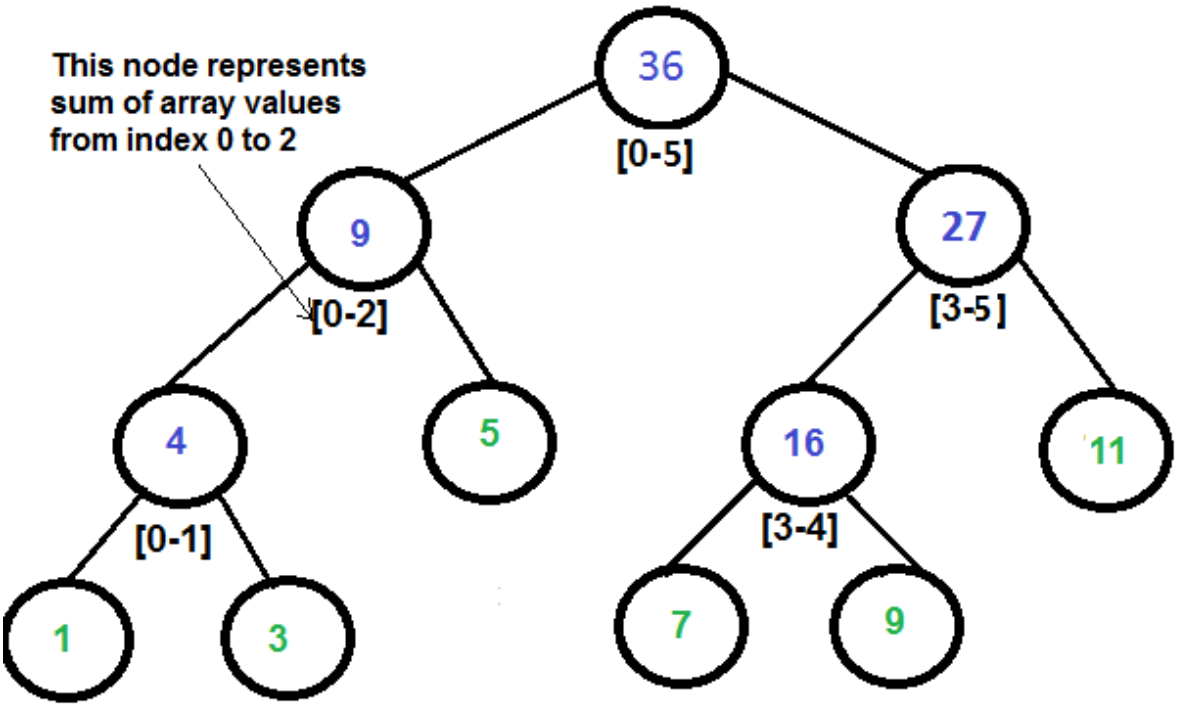
A **simple solution** is to run a loop from l to r and calculate sum of elements in given range. To update a value, simply do arr[i] = x. The first operation takes O(n) time and second operation takes O(1) time.

Another solution is to create another array and store sum from start to i at the ith 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(Logn) 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 . . . 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 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.

C

```
// C program to show segment tree operations like construction, query
// and update
#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
    si    --> 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[si]
    qs & qe  --> Starting and ending indexes of query range */
int getSumUtil(int *st, int ss, int se, int qs, int qe, int si)
{
    // 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[si];

    // 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*si+1) +
           getSumUtil(st, mid+1, se, qs, qe, 2*si+2);
}

/* A recursive function to update the nodes which have the given
   index in their range. The following are parameters
    st, si, 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 si)
{
    // 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[si] = st[si] + diff;
    if (se != ss)
    {
        int mid = getMid(ss, se);
        updateValueUtil(st, ss, mid, i, diff, 2*si + 1);
        updateValueUtil(st, mid+1, se, i, diff, 2*si + 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 (quey 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

    //Height of segment tree
    int x = (int)(ceil(log2(n)));

    //Maximum size of segment tree
    int max_size = 2*(int)pow(2, x) - 1;

    // Allocate memory
    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;
}
```

Java

```
// Java Program to show segment tree operations like construction,
// query and update
class SegmentTree
{
    int st[]; // The array that stores segment tree nodes

    /* Constructor to construct segment tree from given array. This
       constructor  allocates memory for segment tree and calls
       constructSTUtil() to  fill the allocated memory */
    SegmentTree(int arr[], int n)
    {
        // Allocate memory for segment tree
        //Height of segment tree
        int x = (int) (Math.ceil(Math.log(n) / Math.log(2)));

        //Maximum size of segment tree
        int max_size = 2 * (int) Math.pow(2, x) - 1;

        st = new int[max_size]; // Memory allocation

        constructSTUtil(arr, 0, n - 1, 0);
    }

    // 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
      si    --> 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[si]
      qs & qe  --> Starting and ending indexes of query range */
    int getSumUtil(int ss, int se, int qs, int qe, int si)
```

```
{
    // 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[si];

    // 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(ss, mid, qs, qe, 2 * si + 1) +
            getSumUtil(mid + 1, se, qs, qe, 2 * si + 2);
}

/* A recursive function to update the nodes which have the given
   index in their range. The following are parameters
    st, si, 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 ss, int se, int i, int diff, int si)
{
    // 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[si] = st[si] + diff;
    if (se != ss) {
        int mid = getMid(ss, se);
        updateValueUtil(ss, mid, i, diff, 2 * si + 1);
        updateValueUtil(mid + 1, se, i, diff, 2 * si + 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 n, int i, int new_val)
{
    // Check for erroneous input index
    if (i < 0 || i > n - 1) {
        System.out.println("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(0, n - 1, i, diff, 0);
}

// Return sum of elements in range from index qs (quey start) to
// qe (query end).  It mainly uses getSumUtil()
int getSum(int n, int qs, int qe)
{
    // Check for erroneous input values
    if (qs < 0 || qe > n - 1 || qs > qe) {
        System.out.println("Invalid Input");
        return -1;
    }
    return getSumUtil(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 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, si * 2 + 1) +
             constructSTUtil(arr, mid + 1, se, si * 2 + 2);
    return st[si];
}

// Driver program to test above functions
public static void main(String args[])
{
    int arr[] = {1, 3, 5, 7, 9, 11};
    int n = arr.length;
    SegmentTree  tree = new SegmentTree(arr, n);

    // Build segment tree from given array

    // Print sum of values in array from index 1 to 3
    System.out.println("Sum of values in given range = " +
                       tree.getSum(n, 1, 3));

    // Update: set arr[1] = 10 and update corresponding segment
    // tree nodes
    tree.updateValue(arr, n, 1, 10);

    // Find sum after the value is updated
```

```
        System.out.println("Updated sum of values in given range = " +
                tree.getSum(n, 1, 3));
    }
}
//This code is contributed by Ankur Narain Verma
```

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(Logn). To query a sum, we process at most four nodes at every level and number of levels is O(Logn).

The time complexity of update is also O(Logn). To update a leaf value, we process one node at every level and number of levels is O(Logn).

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.

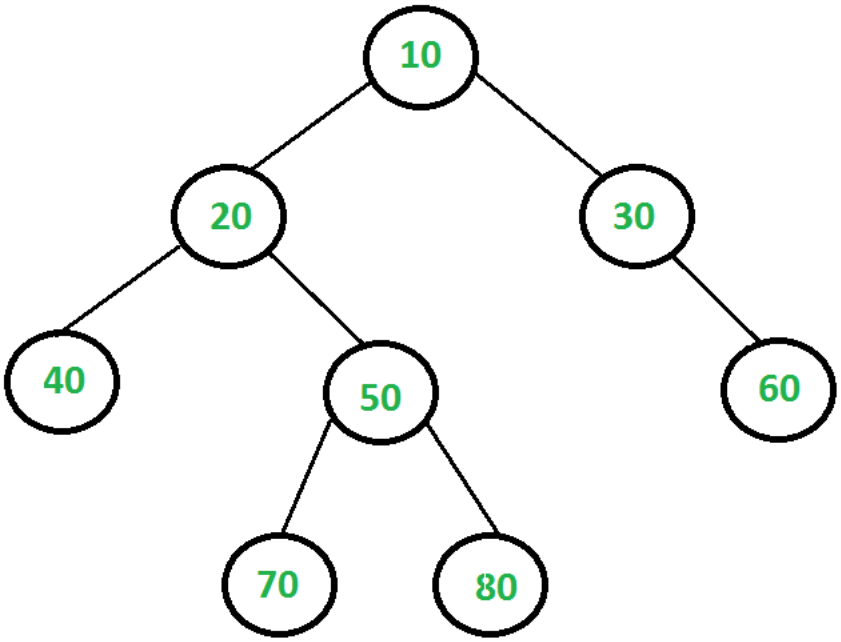
Source

http://www.geeksforgeeks.org/segment-tree-set-1-sum-of-given-range/

Chapter 48

Dynamic Programming | Set 26 (Largest Independent Set Problem)

Given a Binary Tree, find size of the **L**argest **I**ndependent **S**et(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) = MAX { (1 + sum of LISS for all grandchildren of X),
                (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 suproblems are called again, this problem has Overlapping Subprolems property. So LISS problem has both properties (see thisand 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 in
//  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;
}

// 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.

Source

http://www.geeksforgeeks.org/largest-independent-set-problem/

Chapter 49

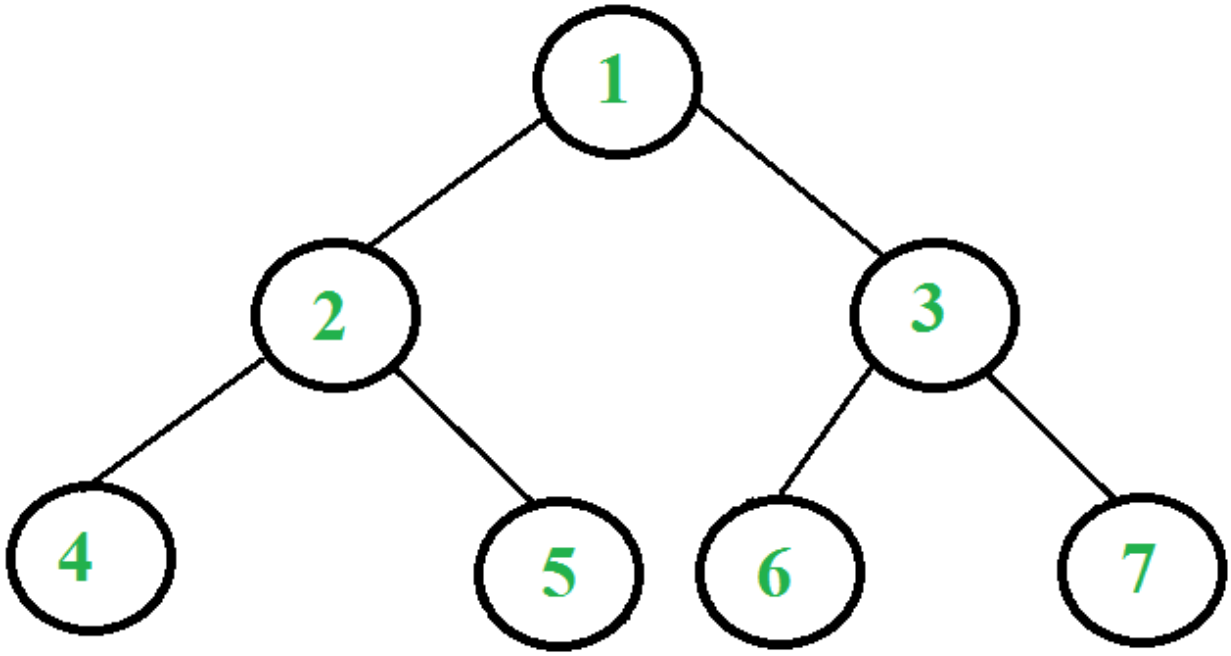
Iterative Postorder Traversal | Set 1 (Using Two Stacks)

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

Source

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

Category: Trees Tags: stack

Post navigation

← Flatten a multilevel linked list Iterative Postorder Traversal | Set 2 (Using One Stack) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 50

Iterative Postorder Traversal | Set 2 (Using One Stack)

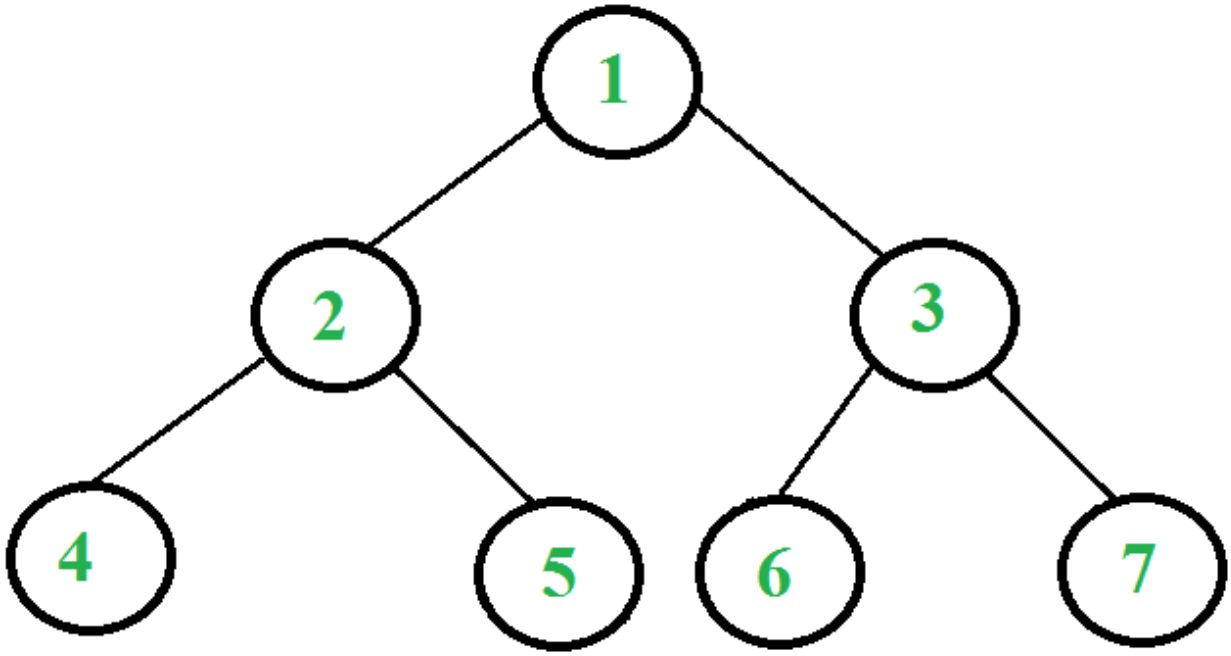
We have discussed a simple iterative postorder traversal using two stacks in the previous post. In this post, an approach with only one stack is discussed.

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

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

Source

http://www.geeksforgeeks.org/iterative-postorder-traversal-using-stack/

Category: Trees Tags: stack

Post navigation

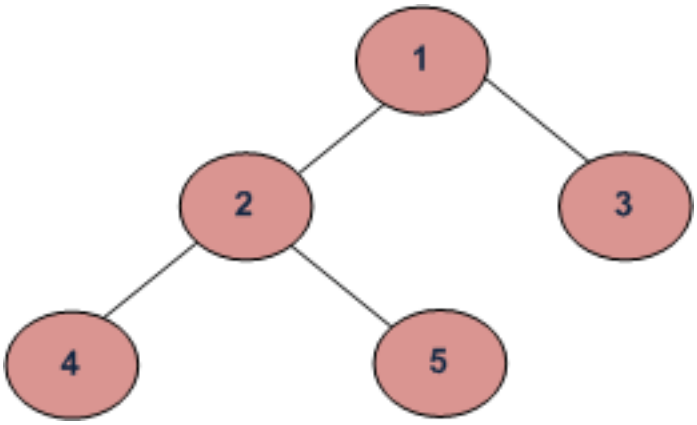
← Iterative Postorder Traversal | Set 1 (Using Two Stacks) Generate n-bit Gray Codes →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 51

Reverse Level Order Traversal

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



Example Tree

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

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

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

```
// 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 protoypes*/
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 i;
    for (i=h; i>=1; i--) //THE ONLY LINE DIFFERENT FROM NORMAL LEVEL ORDER
        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) + .. + 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-in-reverse.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(node* root)
{
    stack <node *> S;
    queue <node *> Q;
    Q.push(root);

    // Do something like normal level order traversal order. Following are the
    // 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

        /* 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.

Source

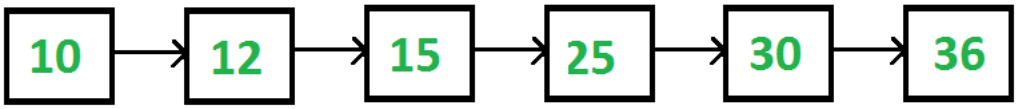
http://www.geeksforgeeks.org/reverse-level-order-traversal/

Chapter 52

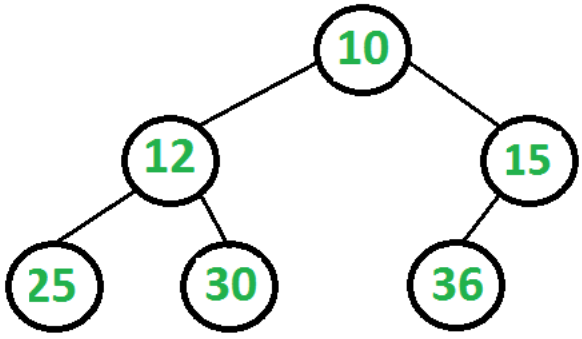
Construct Complete Binary Tree from its Linked List Representation

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.



The above linked list represents following binary tree



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

1. Create an empty queue.
2. Make the first node of the list as root, and enqueue it to the queue.
3. Until we reach the end of the list, do the following.
……**a.** Dequeue one node from the queue. This is the current parent.

……**b.** Traverse two nodes in the list, add them as children of the current parent.
……**c.** Enqueue the two nodes into the queue.

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 into the
// 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 queue
    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.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 them
        // 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.

Source

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

Category: Trees Tags: Queue

Chapter 53

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.

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 predecssor 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 step 3
//  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.

Source

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

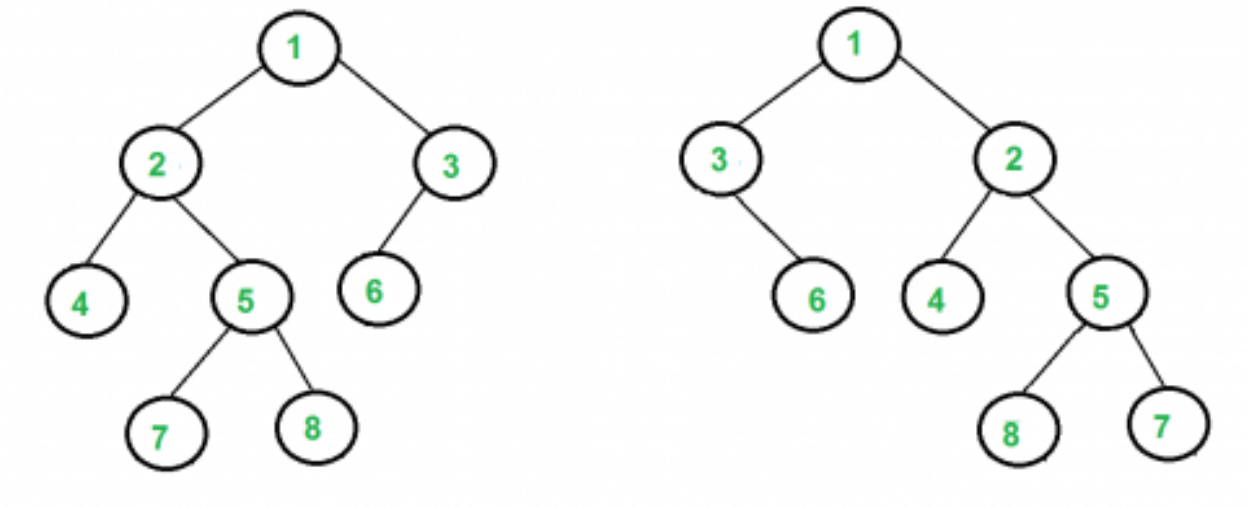
Category: Trees

Chapter 54

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.

Source

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

Category: Trees

Chapter 55

Find all possible interpretations of an array of digits

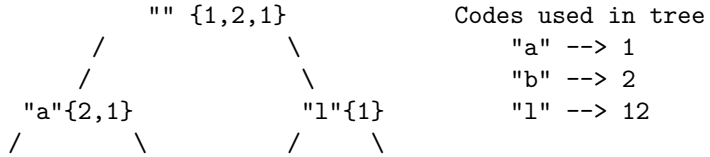
Consider a coding system for alphabets to integers where ‘a’ is represented as 1, ‘b’ as 2, .. ‘z’ as 26. Given an array of digits (1 to 9) as input, write a function that prints all valid interpretations of input array.

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

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



```
              /          \        /       \
           "ab"{1}        "au"    "la"      null
            /    \
           /      \
        "aba"      null
```

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", "v", "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), l(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  l
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

Source

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

Category: Trees Tags: Facebook

Post navigation

← Expression Evaluation Microsoft Interview | Set 19 →

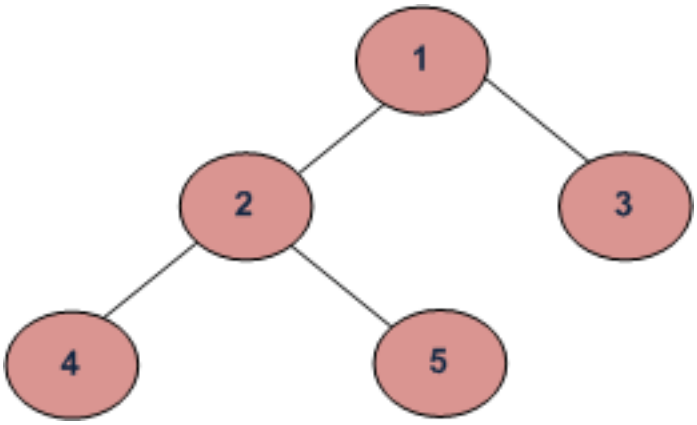
Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 56

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.



Example Tree

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 Bianry Tree
int treeHeight(node *root)
{
    // Base Case
    if (root == NULL)
        return 0;

    // Create an empty queue for level order tarversal
    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 lelvel.
        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

Source

http://www.geeksforgeeks.org/iterative-method-to-find-height-of-binary-tree/

Category: Trees

Chapter 57

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 two 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 printTreeIdented(String indent) {
            System.out.println(indent + "-->" + c);
            for (TreeNode child : children) {
              if (child != null)
                child.printTreeIdented(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

Source

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

Chapter 58

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 DLL
#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 pointers.
    // While traversing, change right pointer of nodes.
    while (root && root->left != NULL)
    {
        prev = root;
        root = root->left;
        root->right = prev;
    }

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

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

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

// Traverses the DLL from left tor 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 = BTToDLL(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

Source

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

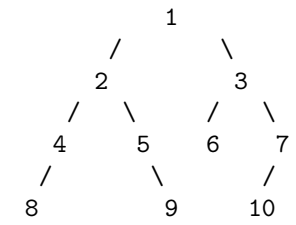
Category: Trees

Chapter 59

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 into
        // the stack so that their right subtrees can be traversed later
        while (root && root->data != key)
        {
            push(stack, root);   // push current node
            root = root->left;  // move to next node
        }

        // If the node whose ancestors are to be printed is found,
        // then break the while loop.
        if (root && root->data == key)
            break;

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

            // If the popped node is right child of top, then remove the top
```

```
            // as well. Left child of the top must have processed before.
            // Consider the following tree for example and key = 3.  If we
            // remove the following loop, the program will go in an
            // infinite loop after reaching 5.
            //          1
            //        /   \
            //       2     3
            //         \
            //           4
            //             \
            //              5
            while (!isEmpty(stack) && peek(stack)->right == root)
               root = pop(stack);
        }

        // if stack is not empty then simply set the root as right child
        // of top and start traversing right sub-tree.
        root = isEmpty(stack)? NULL: peek(stack)->right;
    }

    // If stack is not empty, print contents of stack
    // Here assumption is that the key is there in tree
    while (!isEmpty(stack))
        printf("%d ", pop(stack)->data);
}

// 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 contrubuted by**Chandra Prakash**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

Source

http://www.geeksforgeeks.org/print-ancestors-of-a-given-binary-tree-node-without-recursion/

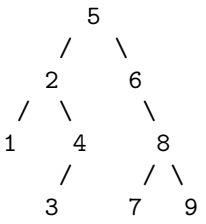
Category: Trees Tags: stack, StackAndQueue

Chapter 60

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

Given a a Binary Tree, find the difference between the sum of nodes at odd level and the sum of nodes at even level. Consider root as level 1, left and right children of root as level 2 and so on.

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



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

Source

http://www.geeksforgeeks.org/difference-between-sums-of-odd-and-even-levels/

Category: Trees

Chapter 61

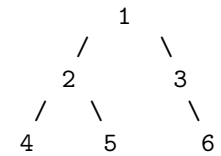
Print Postorder traversal from given Inorder and Preorder traversals

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}
```

Trversals in the above example represents following tree



A **naive method** is to first construct the tree, then use simple recursive method to print postorder traversal of the constructed tree.

We can print postorder traversal without constructing the tree. The idea is, root is always the first item in preorder traversal and it must be the last item in postorder traversal. We first recursively print left subtree, then recursively print right subtree. Finally, print root. To find boundaries of left and right subtrees in pre[] and in[], we search root in in[], all elements before root in in[] are elements of left subtree and all elements after root are elements of right subtree. In pre[], all elements after index of root in in[] are elements of right subtree. And elements before index (including the element at index and excluding the first element) are elements of left subtree.

```
// C++ program to print postorder traversal from preorder and inorder traversals
```

```
#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 traversals
void printPostOrder(int in[], int pre[], int n)
{
   // The first element in pre[] is always root, search it
   // in in[] to find left and right subtrees
   int root = search(in, pre[0], n);

   // If left subtree is not empty, print left subtree
   if (root != 0)
      printPostOrder(in, pre+1, root);

   // If right subtree is not empty, print right subtree
   if (root != n-1)
      printPostOrder(in+root+1, pre+root+1, n-root-1);

   // Print root
   cout << pre[0] << " ";
}

// 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

Source

http://www.geeksforgeeks.org/print-postorder-from-given-inorder-and-preorder-traversals/

Category: Trees

Post navigation

← NP-Completeness | Set 1 (Introduction) Morgan Stanley Interview | Set 3 →

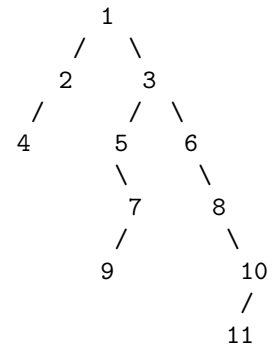
Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 62

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
int depthOfOddLeafUtil(Node *root,int level)
{
    // Base Case
    if (root == NULL)
        return 0;

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

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

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

// 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.

Source

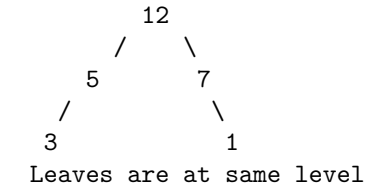
http://www.geeksforgeeks.org/find-depth-of-the-deepest-odd-level-node/

Category: Trees

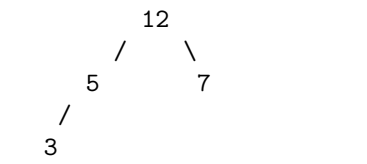
Chapter 63

Check if all leaves are at same level

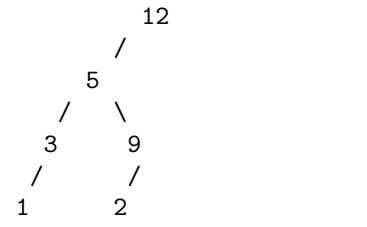
Given a Binary Tree, check if all leaves are at same level or not.




```
Leaves are at same level
```




```
Leaves are Not at same level
```




```
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 leafs 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 thirdt 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.

Source

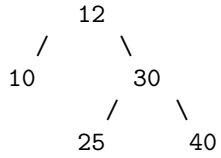
http://www.geeksforgeeks.org/check-leaves-level/

Category: Trees

Chapter 64

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

Source

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

Category: Trees

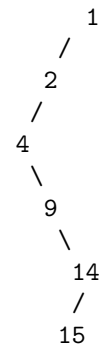
Chapter 65

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

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

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

```
Consider the following Binary Tree
          1
      /      \
     2        3
   /   \     /  \
  4     5   6    7
 / \    /       /
8   9  12      10
   / \           \
  13  14         11
      /
     15

For input k = 20, the tree should be changed to following
(Nodes with values 6 and 8 are deleted)
          1
      /      \
     2        3
   /   \        \
  4     5        7
   \    /       /
    9  12      10
   / \           \
  13  14         11
      /
     15

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
    // chilren 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.

Source

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

Category: Trees

Post navigation

← How to make a C++ class whose objects can only be dynamically allocated? How to change the output of printf() in main() ? →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 66

Extract Leaves of a Binary Tree in a Doubly Linked List

Given a Binary Tree, extract all leaves of it in a **D**oubly **L**inked **L**ist (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
        1
     /     \
    2       3
   / \       \
  4   5       6
 / \         / \
7   8       9   10


Output:
Doubly Linked List
785910

Modified Tree:
        1
     /     \
    2       3
   /         \
  4           6
```

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 as
// head of doubly linked list.  left pointer of tree is used as prev in DLL
// and right pointer is used as next
struct Node* extractLeafList(struct Node *root, struct Node **head_ref)
{
   // Base cases
   if (root == NULL)  return NULL;

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

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

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

       return NULL;  // Return new root
   }

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

   return root;
}

// 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)
{
     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 Trvaersal 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 Trvaersal 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.

Source

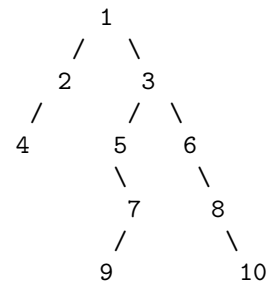
http://www.geeksforgeeks.org/connect-leaves-doubly-linked-list/

Category: Trees

Chapter 67

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 Coder011for 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

Source

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

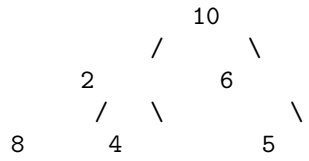
Category: Trees

Chapter 68

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 tarversal
    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->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

Source

http://www.geeksforgeeks.org/find-next-right-node-of-a-given-key/

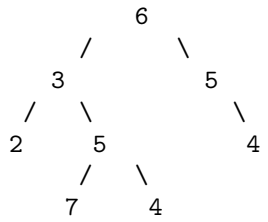
Category: Trees

Chapter 69

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

Source

http://www.geeksforgeeks.org/sum-numbers-formed-root-leaf-paths/

Category: Trees

Chapter 70

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.

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
    {
        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

Source

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

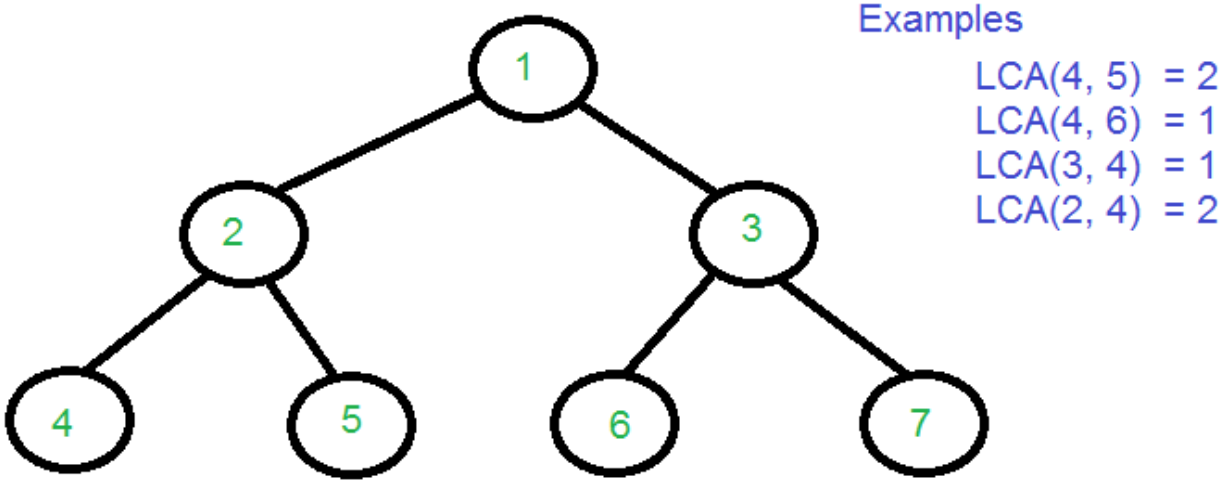
Chapter 71

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 Bianry Tree node
struct Node
{
    int key;
    struct Node *left, *right;
};

// 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;
}

// 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, bool &v2)
{
    // Base case
    if (root == NULL) return NULL;

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

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

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

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

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

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

    // Else return false
    return false;
}

// This function returns LCA of n1 and n2 only if both n1 and n2 are present
// in tree, otherwise returns NULL;
Node *findLCA(Node *root, int n1, int n2)
```

```
{
    // Initialize n1 and n2 as not visited
    bool v1 = false, v2 = false;

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

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

    // Else return NULL
    return NULL;
}

// 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

Source

http://www.geeksforgeeks.org/lowest-common-ancestor-binary-tree-set-1/

Category: Trees

Post navigation

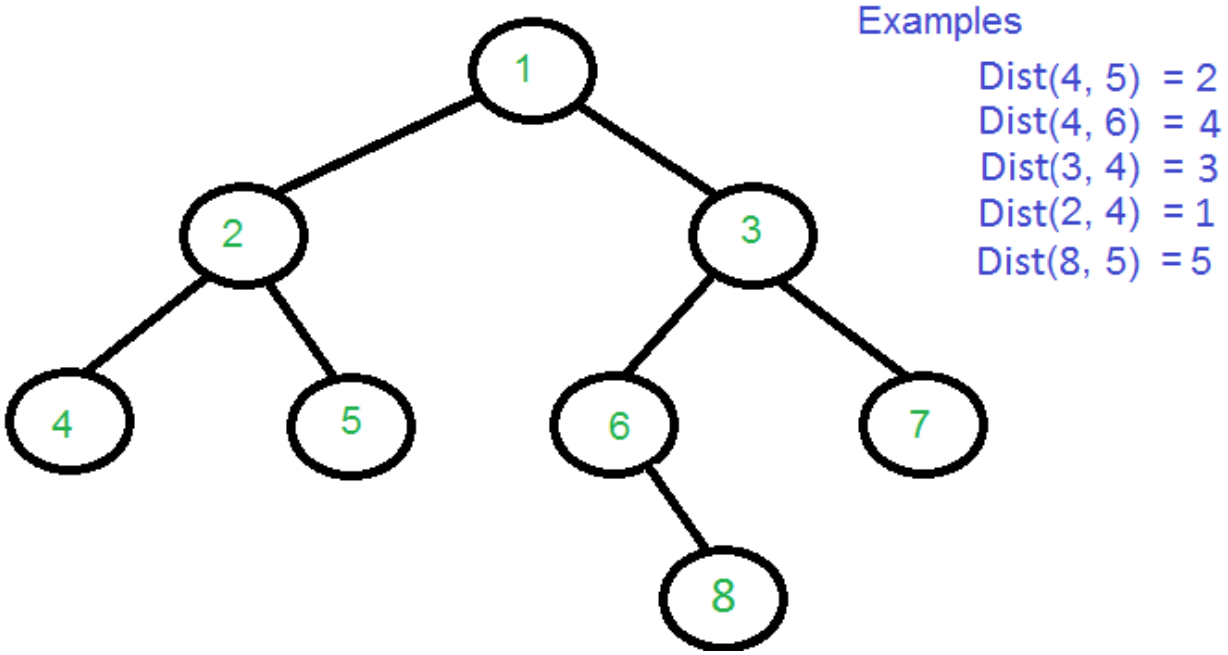
← Interesting Facts about Bitwise Operators in C Find distance between two given keys of a Binary Tree →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 72

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 -1
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, lvl+1);
    Node *right_lca = findDistUtil(root->right, n1, n2, d1, d2, dist, lvl+1);

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

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

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

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

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

    // If n2 is ancestor of n1, consider n2 as root and find level
    // of n1 in subtree rooted with n2
    if (d2 != -1)
```

```
    {
        dist = findLevel(lca, n1, 0);
        return dist;
    }

    return -1;
}

// 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, 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

Source

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

Category: Trees

Post navigation

← Lowest Common Ancestor in a Binary Tree | Set 1 Print a long int in C using putchar() only →

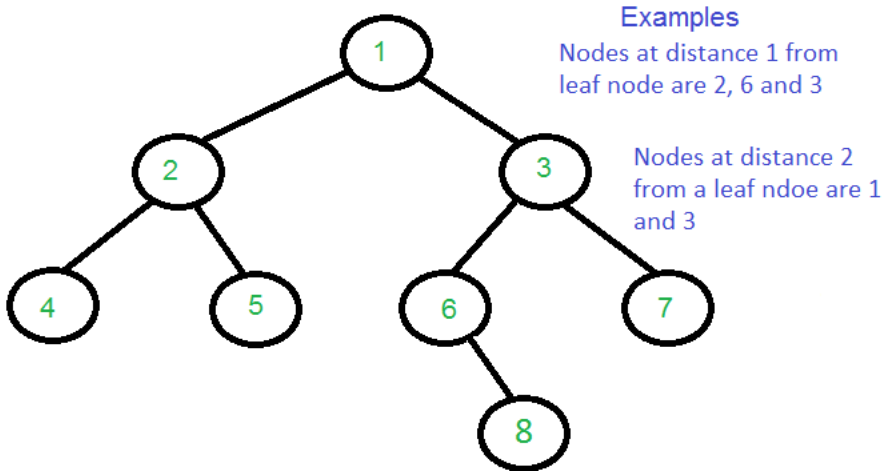
Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 73

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 k
                 distance away from many leaves, we want to print it 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 nuber 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

Source

http://www.geeksforgeeks.org/print-nodes-distance-k-leaf-node/

Category: Trees

Chapter 74

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.

```
   12
     \
      14
        \
         16
Cannot be a Red-Black Tree
with any color assignment
Max height of 12 is 1
Min height of 12 is 3
```

```
      40
     /   \
   10     100
          /  \
         60   150
It can be Red-Black Tree
```

```
        10
       /   \
     5      100
           /   \
          50   150
         /
        40
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 returns tree if the Binary tree is balanced like a Red-Black
// tree. This function also sets value in maxh and minh (passed by
// reference). maxh and minh are set as maximum and minimum heights of root.
bool isBalancedUtil(Node *root, int &maxh, int &minh)
{
    // Base case
    if (root == NULL)
    {
        maxh = minh = 0;
        return true;
    }

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

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

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

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

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

    return false;
}
```

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

/* 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

Source

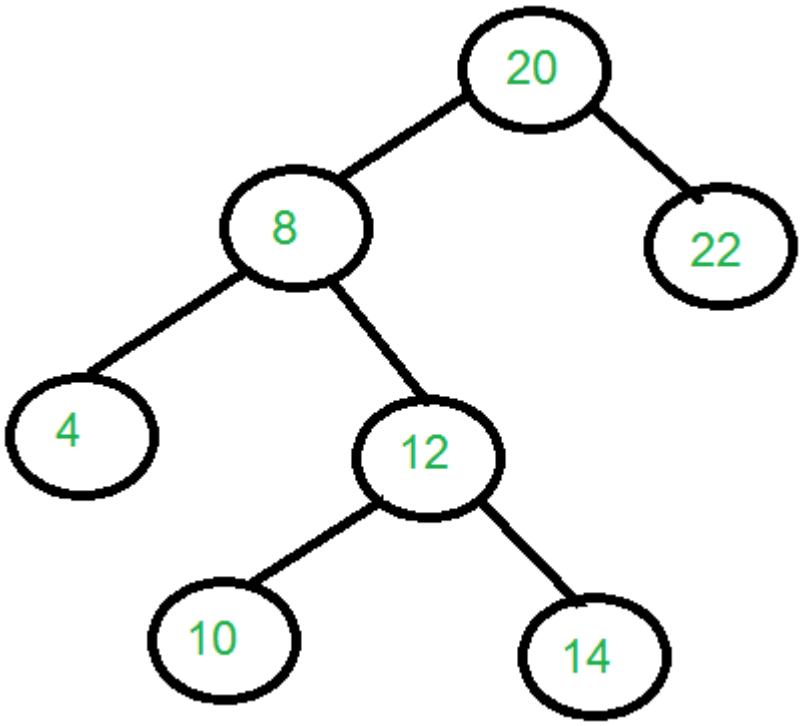
http://www.geeksforgeeks.org/check-given-binary-tree-follows-height-property-red-black-tree/

Category: Trees

Chapter 75

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 thisfor 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 dl = printkdistanceNode(root->left, target, k);

    // Check if target node was found in left subtree
    if (dl != -1)
    {
         // If root is at distance k from target, print root
         // Note that dl is Distance of root's left child from target
         if (dl + 1 == k)
            cout << root->data << endl;

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

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

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

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

```
// 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:

```
4
20
```

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

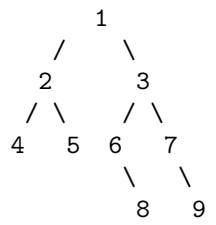
Source

http://www.geeksforgeeks.org/print-nodes-distance-k-given-node-binary-tree/

Chapter 76

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.
// 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 resepect 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
7
9
```

Time Complexity: Time complexity of above algorithm is O(w*n) where w is width of Binary Tree and n is number of nodes in Binary Tree. In worst case, the value of w can be O(n) (consider a complete tree for example) and time complexity can become $O(n^2)$.

This problem can be solved more efficiently using the technique discussed in thispost. 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

Source

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

Category: Trees

Post navigation

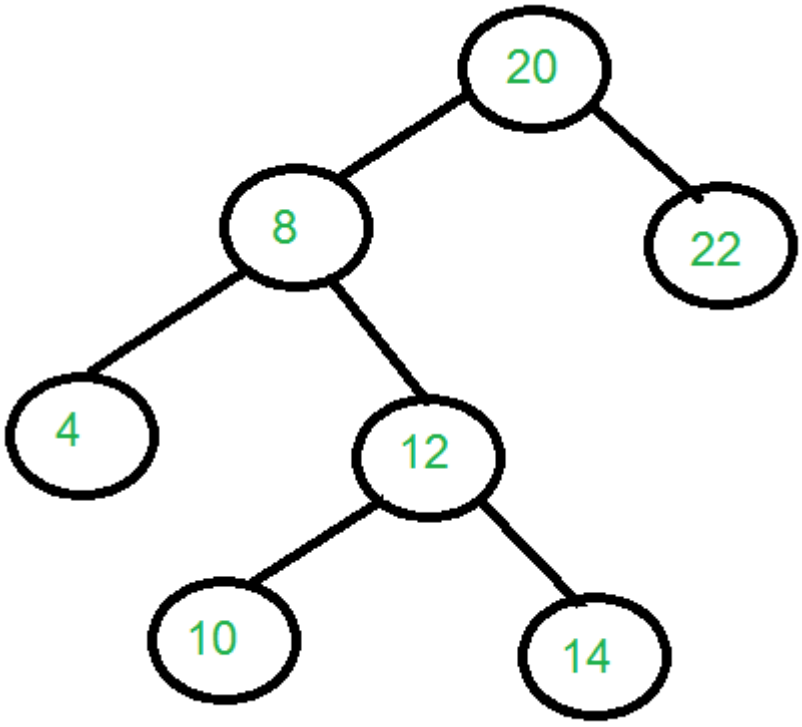
← Interval Tree Integer Promotions in C →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 77

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
```

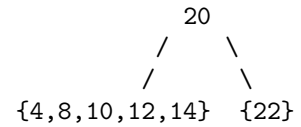
We strongly recommend to minimize the browser and try this yourself first.
The following post can be considered as a prerequisite for this.

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.



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 *extrackKeys(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 inStrt and inEnd should be 0 and n -1.
   The function doesn't do any error checking for cases
   where inorder and levelorder do not form a tree */
Node* buildTree(int in[], int level[], int inStrt, int inEnd, int n)
{

    // If start index is more than the end index
    if (inStrt > inEnd)
        return NULL;

    /* The first node in level order traversal is root */
    Node *root = newNode(level[0]);

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

```
        return root;

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

    // Extract left subtree keys from level order traversal
    int *llevel  = extrackKeys(in, level, inIndex, n);

    // Extract right subtree keys from level order traversal
    int *rlevel  = extrackKeys(in + inIndex + 1, level, n-inIndex-1, n);

    /* construct left and right subtress */
    root->left = buildTree(in, llevel, inStrt, inIndex-1, n);
    root->right = buildTree(in, rlevel, inIndex+1, inEnd, n);

    // Free memory to avoid memory leak
    delete [] llevel;
    delete [] rlevel;

    return root;
}

/* Uti;ity 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 Insorder 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

Source

http://www.geeksforgeeks.org/construct-tree-inorder-level-order-traversals/

Category: Trees

Post navigation

← Amazon Interview | Set 75 (For SDE-1) Red-Black Tree | Set 3 (Delete) →

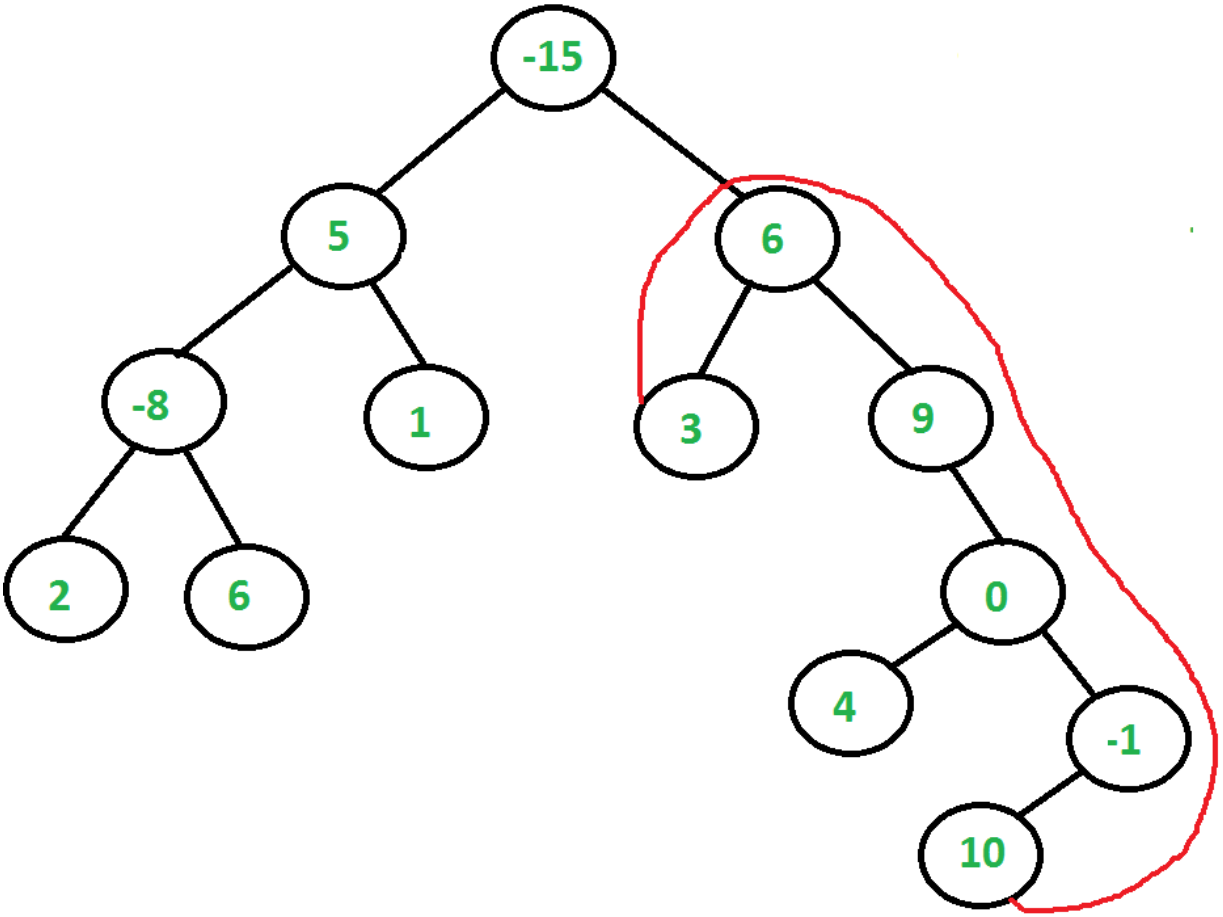
Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 78

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).

If one side of root is empty, then function should return minus infinite (INT_MIN in case of C/C++)



A simple solution is to traverse the tree and do following for every traversed node X.
1) Find maximum sum from leaf to root in left subtree of X (we can use this post for this and next steps)
2) Find maximum sum from leaf to root in right subtree of X.
3) Add the above two calculated values and X->data and compare the sum with the maximum value obtained so far and update the maximum value.
4) Return the maximum value.

The time complexity of above solution is $O(n^2)$

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->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 <bits/stdc++.h>
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.
// If one side of root is empty, then it returns INT_MIN
int maxPathSumUtil(struct Node *root, int &res)
{
    // Base cases
    if (root==NULL) return 0;
    if (!root->left && !root->right) return root->data;
```

```
    // Find maximum sum in left and right subtree. Also
    // find maximum root to leaf sums in left and right
    // subtrees and store them in ls and rs
    int ls = maxPathSumUtil(root->left, res);
    int rs = maxPathSumUtil(root->right, res);


    // If both left and right children exist
    if (root->left && root->right)
    {
        // Update result if needed
        res = max(res, ls + rs + root->data);

        // Return maxium possible value for root being
        // on one side
        return max(ls, rs) + root->data;
    }

    // If any of the two children is empty, return
    // root sum for root being on one side
    return (!root->left)? rs + root->data:
                          ls + root->data;
}

// 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 = INT_MIN;
    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(root);
    return 0;
```

```
}
```

Output:

```
Max pathSum of the given binary tree is 27.
```

Thanks to Saurabh Vats for suggesting corrections in original approach.

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

Source

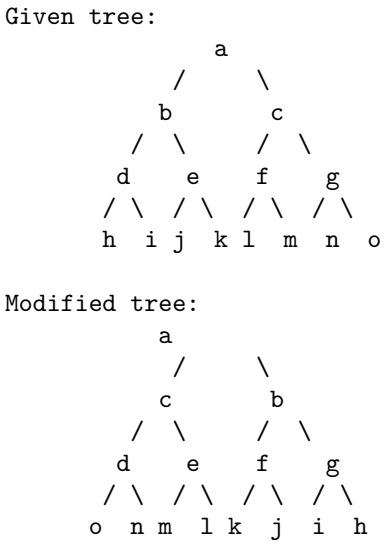
http://www.geeksforgeeks.org/find-maximum-path-sum-two-leaves-binary-tree/

Category: Trees

Chapter 79

Reverse alternate levels of a perfect binary tree

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

```
    modifyTree(root->left, arr, index, l+1);

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

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

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

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

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

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

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

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

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

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

    reverseAlternate(root);

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

    return 0;
}
```

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.

Source

http://www.geeksforgeeks.org/reverse-alternate-levels-binary-tree/

Category: Trees

Post navigation

← [TopTalent.in] Exclusive Interview with Prashanth from IIT Madras who landed a job at Microsoft, Redmond Calculate the angle between hour hand and minute hand →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 80

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

```
     6
   /   \
  3     5
 / \   / \
7   8 1   3
Say two node be 7 and 1, result is TRUE.
Say two nodes are 3 and 5, result is FALSE.
Say two nodes are 7 and 5, result is FALSE.
```

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;
}
```

Ouput:

```
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

Source

http://www.geeksforgeeks.org/check-two-nodes-cousins-binary-tree/

Category: Trees

Post navigation

← Amazon Interview | Set 99 (On-Campus) Search in an almost sorted array →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

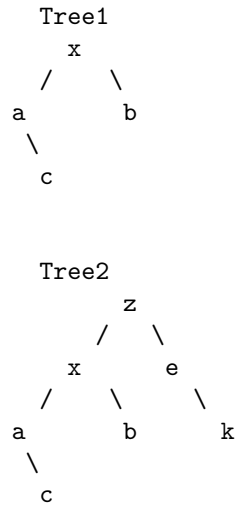
Chapter 81

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 arew substrings of inorder and preorder traversals of T respectively.

Following are detailed steps.

1) Find inorder and preorder traversals of T, store them in two auxiliary arrays inT[] and preT[].

2) Find inorder and preorder traversals of S, store them in two auxiliary arrays inS[] and preS[].

3) If inS[] is a subarray of inT[] and preS[] is a subarray preT[], then S is a subtree of T. Else not.

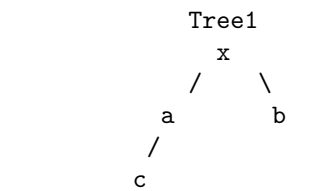
We can also use postorder traversal in place of preorder in the above algorithm.

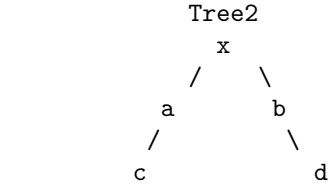
Let us consider the above example

```
Inorder and Preorder traversals of the big tree are.
inT[]   =  {a, c, x, b, z, e, k}
preT[]  =  {z, x, a, c, b, e, k}

Inorder and Preorder traversals of small tree are
inS[]  = {a, c, x, b}
preS[] = {x, a, c, b}

We can easily figure out that inS[] is a subarray of
inT[] and preS[] is a subarray of preT[].
```

EDIT

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






```
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 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

Source

http://www.geeksforgeeks.org/check-binary-tree-subtree-another-binary-tree-set-2/

Category: Trees

Post navigation

← Amazon Interview | Set 97 (On-Campus for SDE1) Euler Circuit in a Directed Graph →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 82

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 -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 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);
}

// This function stores a tree in a file pointed by fp
void serialize(Node *root, FILE *fp)
{
    // If current node is NULL, store marker
    if (root == NULL)
    {
        fprintf(fp, "%d ", MARKER);
        return;
    }

    // Else, store current node and recur for its children
    fprintf(fp, "%d ", root->key);
    serialize(root->left, fp);
    serialize(root->right, fp);
}

// This function constructs a tree from a file pointed by 'fp'
void deSerialize(Node *&root, FILE *fp)
{
    // Read next item from file. If theere are no more items or next
    // item is marker, then return
    int val;
    if ( !fscanf(fp, "%d ", &val) || val == MARKER)
       return;

    // Else create node with this item and recur for children
    root = newNode(val);
    deSerialize(root->left, fp);
    deSerialize(root->right, fp);
}

// 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);
```

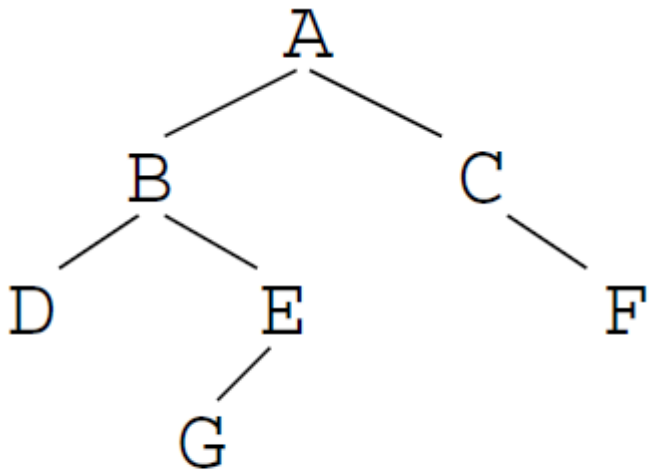
```
    root->right              = newNode(22);
    root->left->left         = newNode(4);
    root->left->right        = newNode(12);
    root->left->right->left  = newNode(10);
    root->left->right->right = newNode(14);

    // 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 storeed 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.

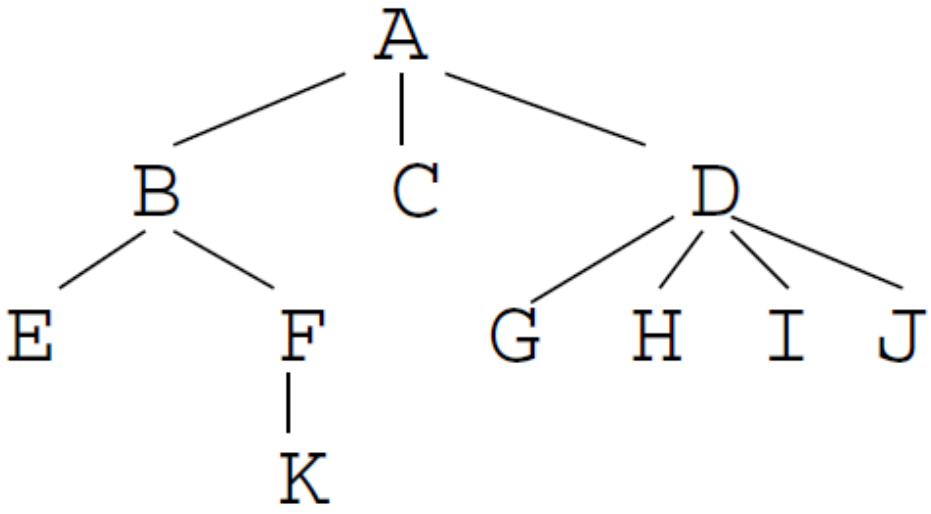
Can we optimize it further?
The above solution can be optimized in many ways. If we take a closer look at above serialized trees, we can observer that all leaf nodes require two markers. One simple optimization is to store a separate bit with every node to indicate that the node is internal or external. This way we don't have to store two markers with every leaf node as leaves can be identified by extra bit. We still need marker for internal nodes with one child. For example in the following diagram ' is used to indicate an internal node set bit, and '/' is used as NULL marker. The diagram is taken from here.



A′B′DE′G/C′/F

Please note that there are always more leaf nodes than internal nodes in a Binary Tree (Number of leaf nodes is number of internal nodes plus 1, so this optimization makes sense.

How to serialize n-ary tree?
In an n-ary tree, there is no designated left or right child. We can store an 'end of children' marker with every node. The following diagram shows serialization where ')' is used as end of children marker. We will soon be covering implementation for n-ary tree. The diagram is taken from here.



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

Source

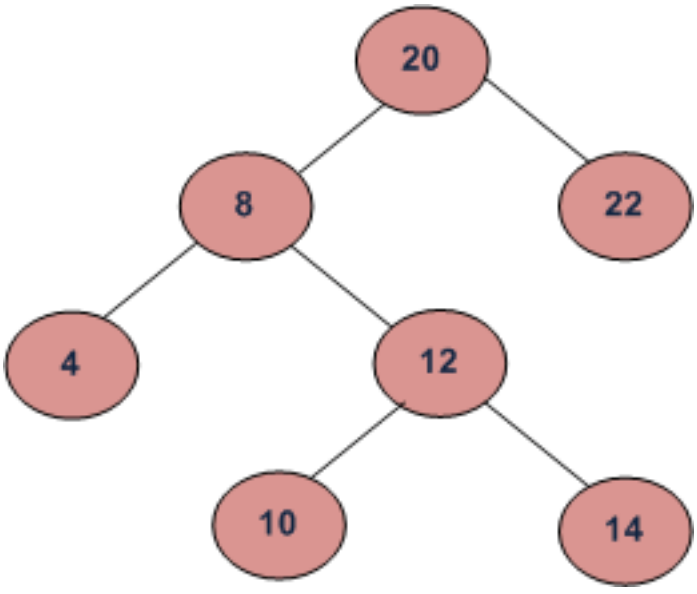
http://www.geeksforgeeks.org/serialize-deserialize-binary-tree/

Chapter 83

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 berween given two levels.
#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++;

            // 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

Source

http://www.geeksforgeeks.org/given-binary-tree-print-nodes-two-given-level-numbers/

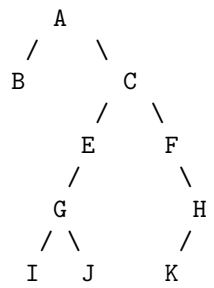
Category: Trees

Chapter 84

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 closesr leaf of a given key in Binary Tree
#include <iostream>
#include <climits>
using namespace std;

/* A binary tree Node has key, pocharer 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 pocharers. */
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 cloest 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 curremt index in 'ancestors[]'
int findClosestUtil(struct Node *root, char k, struct Node *ancestors[],
                                                int index)
{
    // Base case
    if (root == NULL)
        return INT_MAX;

    // If key found
    if (root->key == k)
    {
        //  Find the cloest leaf under the subtree rooted with given key
        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'. It
// mainly uses recursive function findClosestUtil() to find the closes
// distance.
int findClosest(struct Node *root, char k)
{
    // Create an array to store ancestors
    // Assumptiom: 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 << "Distace of the closest key from " << k << " is "
         << findClosest(root, k) << endl;
    k = 'C';
    cout << "Distace of the closest key from " << k << " is "
         << findClosest(root, k) << endl;
    k = 'E';
    cout << "Distace of the closest key from " << k << " is "
         << findClosest(root, k) << endl;
    k = 'B';
    cout << "Distace of the closest key from " << k << " is "
         << findClosest(root, k) << endl;

    return 0;
}
```

Output:

```
 Distace of the closest key from H is 1
Distace of the closest key from C is 2
Distace of the closest key from E is 2
Distace 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 can that traverses ancestors array can be optimized to not traverse ancestors which are at more distance than current result.

Exercise:
Extend the above solution to print not only distance, but the key of closest leaf also.

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

Source

http://www.geeksforgeeks.org/find-closest-leaf-binary-tree/

Category: Trees

Post navigation

← Nuts & Bolts Problem (Lock & Key problem) Print all possible strings that can be made by placing spaces →

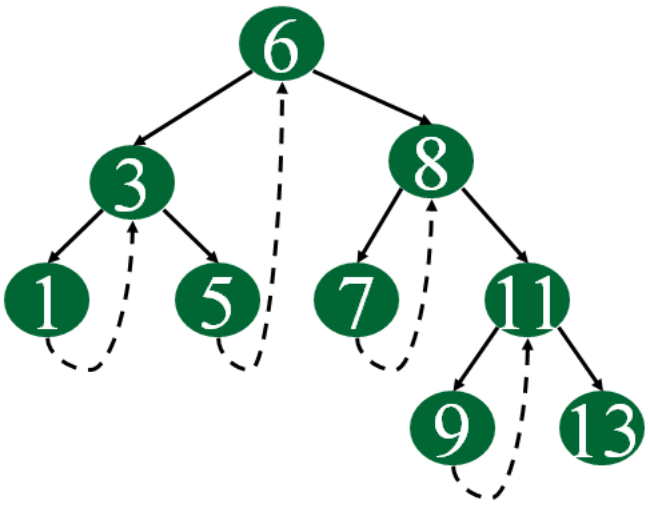
Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 85

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 queuue and make it the right of current node. We also set isThreaded to true to indicate that the right pointer is a threaded link.

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 threadded 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

Source

http://www.geeksforgeeks.org/convert-binary-tree-threaded-binary-tree/

Category: Trees

Chapter 86

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.

```
       1
    /     \
   2       3
  /  \    / \
 4    5  6   7
Top view of the above binary tree is
4 2 1 3 7
```

```
        1
      /   \
    2       3
      \
        4
          \
            5
             \
               6
Top view of the above binary tree is
2 1 3 6
```

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.

Source

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

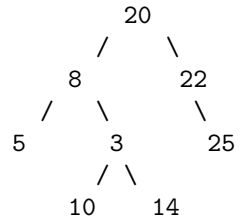
Category: Trees

Chapter 87

Bottom View of a Binary Tree

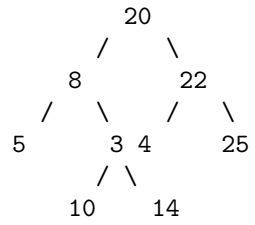
Given a Binary Tree, we need to print the bottom view from left to right. A node x is there in output if x is the bottommost node at its horizontal distance. Horizontal distance of left child of a node x is equal to horizontal distance of x minus 1, and that of right child is horizontal distance of x plus 1.

Examples:



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 Bianry Tree.
1. We put tree nodes in a queue for the level order traversal.

2. Start with the horizontal distance(hd) 0 of the root node, keep on adding left child to queue along with the horizontal distance as hd-1 and right child as hd+1.
3. Also, use a TreeMap which stores key value pair sorted on key.
4. Every time, we encounter a new horizontal distance or an existing horizontal distance put the node data for the horizontal distance as key. For the first time it will add to the map, next time it will replace the value. This will make sure that the bottom most element for that horizontal distance is present in the map and if you see the tree from beneath that you will see that element.

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.

Source

http://www.geeksforgeeks.org/bottom-view-binary-tree/

Chapter 88

Perfect Binary Tree Specific Level Order Traversal

Given a Perfect Binary Tree like below:
(click on image to get a clear view)

Print the level order of nodes in following specific manner:

```
1 2 3 4 7 5 6 8 15 9 14 10 13 11 12 16 31 17 30 18 29 19 28 20 27 21 26  22 25 23 24
```

i.e. print nodes in level order but nodes should be from left and right side alternatively. Here 1st and 2nd levels are trivial.
While 3rd level: 4(left), 7(right), 5(left), 6(right) are printed.
While 4th level: 8(left), 15(right), 9(left), 14(right), .. are printed.
While 5th level: 16(left), 31(right), 17(left), 30(right), .. are printed.

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:

```
 Specific Level Order traversal of binary tree is
1 2 3 4 7 5 6 8 15 9 14 10 13 11 12 16 31 17 30 18 29 19 28 20 27 21 26 22 25 23 24
```

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.

Source

http://www.geeksforgeeks.org/perfect-binary-tree-specific-level-order-traversal/

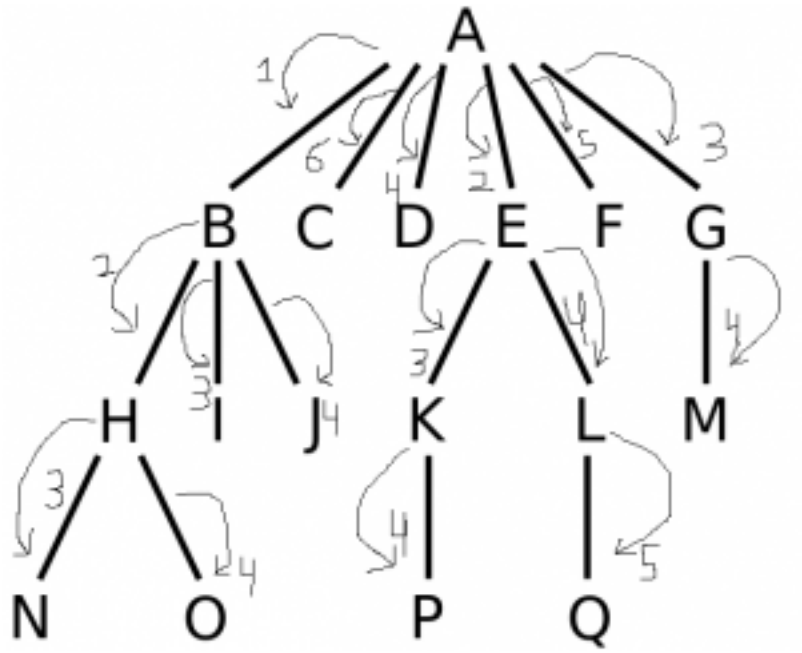
Chapter 89

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 one of its remaining children. Continuing in this way we have to find the minimum no of iterations required to pass the information to all nodes in the tree.

Minimum no of iterations for tree below is 6. The root A first passes information to B. In next iteration, A passes information to E and B passes information to H and so on.



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

This can be done using Post Order Traversal. The idea is to consider height and children count on each and every node.

If a child node i takes ci iterations to pass info below its subtree, then its parent will take (ci + 1) iterations to pass info to subtree rooted at that child i.
If parent has more children, it will pass info to them in subsequent iterations. Let's say children of a parent takes c1, c2, c3, c4, …, cn iterations to pass info in their own subtree, Now parent has to pass info to these n children one by one in n iterations. If parent picks child i in ith iteration, then parent will take (i + ci) iterations to pass info to child i and all it's subtree.
In any iteration, when parent passes info a child i+1, children (1 to i) which got info from parent already in previous iterations, will pass info to further down in subsequent iterations, if any child (1 to i) has its own child further down.
To pass info to whole tree in minimum iterations, it needs to be made sure that bandwidth is utilized as efficiently as possible (i.e. maximum passable no of nodes should pass info further down in any iteration)
The best possible scenario would be that in nth iteration, n different nodes pass info to their child.
Nodes with height = 0: (Trivial case) Leaf node has no children (no information passing needed), so no of iterations would be ZERO.
Nodes with height = 1: Here node has to pass info to all the children one by one (all children are leaf node, so no more information passing further down). Since all children are leaf, node can pass info to any child in any order (pick any child who didn't receive the info yet). One iteration needed for each child and so no of iterations would be no of children.So node with height 1 with n children will take n iterations.
Take a counter initialized with ZERO, loop through all children and keep incrementing counter.
Nodes with height > 1: Let's assume that there are n children (1 to n) of a node and minimum no iterations for all n children are c1, c2, …., cn.
To make sure maximum no of nodes participate in info passing in any iteration, parent should 1st pass info to that child who will take maximum iteration to pass info further down in subsequent iterations. i.e. in any iteration, parent should choose the child who takes maximum iteration later on. It can be thought of as a greedy approach where parent choose that child 1st, who needs maximum no of iterations so that all subsequent iterations can be utilized efficiently.
If parent goes in any other fashion, then in the end, there could be some nodes which are done quite early, sitting idle and so bandwidth is not utilized efficiently in further iterations.
If there are two children i and j with minimum iterations ci and cj where ci > cj, then If parent picks child j 1st then no of iterations needed by parent to pass info to both children and their subtree would be:max (1 + cj, 2 + ci) = 2 + ci
If parent picks child i 1st then no of iterations needed by parent to pass info to both children and their subtree would be: max(1 + ci, 2 + cj) = 1 + ci (So picking ci gives better result than picking cj)

This tells that parent should always choose child i with max ci value in any iteration.
SO here greedy approach is:
sort all ci values decreasing order,
let's say after sorting, values are c1 > c2 > c3 > …. > cn
take a counter c, set c = 1 + c1 (for child with maximum no of iterations)
for all children i from 2 to n, c = c + 1 + ci
then total no of iterations needed by parent is max(n, c)

Let ***minItr(A)*** be the minimum iteration needed to pass info from node A to it's all the sub-tree. Let ***child(A)*** be the count of all children for node A. So recursive relation would be:

```
1. Get minItr(B) of all children (B) of a node (A)
2. Sort all minItr(B) in descending order
3. Get minItr of A based on all minItr(B)
    minItr(A) = child(A)
    For children B from i = 0 to child(A)
             minItr(A) = max ( minItr(A), minItr(B) + i + 1)

Base cases would be:
If node is leaf, minItr = 0
```

```
If node's height is 1, minItr = children count
```

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
{
    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 children
// of node u, sort them in decreasing order and then get minimum iteration
// 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, 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
TestCase 9 - Minimum Iteration: 8
```

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.

Source

http://www.geeksforgeeks.org/minimum-iterations-pass-information-nodes-tree/

Category: Trees

Post navigation

← [TopTalent.in] Exclusive Interview with Abhishek who got into DE Shaw Intuit Interview | Set 3 (For SE-2) →

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 90

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 tre node in hashtable. Following are detailed steps.

1) Recursively traverse the given Binary and copy key value, left pointer and right pointer to clone tree. While copying, store the mapping from given tree node to clone tree node in a hashtable. In the following pseudo code, 'cloneNode' is currently visited node of clone tree and 'treeNode' is currently visited node of given tree.

```
cloneNode->key  = treeNode->key
cloneNode->left = treeNode->left
cloneNode->right = treeNode->right
map[treeNode] = cloneNode
```

2) Recursively traverse both trees and set random pointers using entries from hash table.

```
cloneNode->random = map[treeNode->random]
```

Following is C++ implementation of above idea. The following implementation uses mapfrom 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 pointers
#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 sutree */
    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";
```

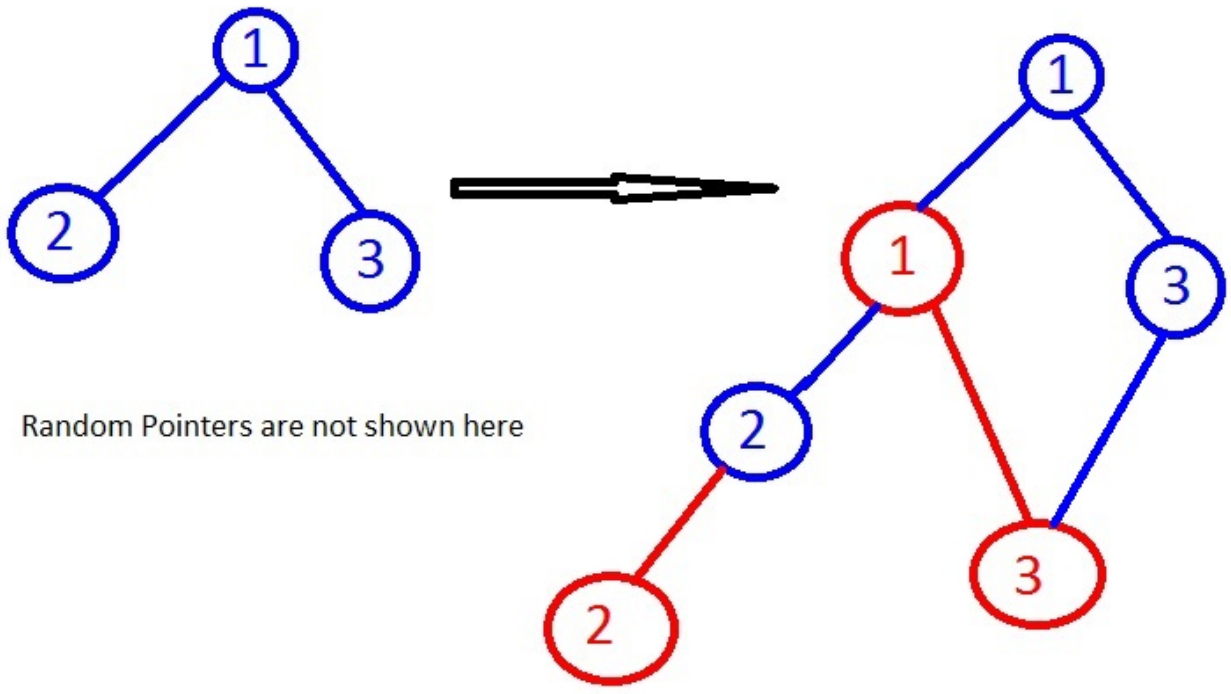
```
    printInorder(tree);

    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 wil be created (say cA) and it will be put as A — >> cA — >> B (B can be a NULL or a non-NULL left child). Right child pointer will be set correctly i.e. if for current node A, right child is C in original tree (A — >> C) then corresponding cloned nodes cA and cC will like cA —- >> cC



2. Set random pointer in cloned tree as per original tree
i.e. if node A's random pointer points to node B, then in cloned tree, cA will point to cB (cA and cB are new node in cloned tree corresponding to node A and B in original tree)

3. Restore left pointers correctly in both original and cloned tree

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 sutree */
    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
// in between current node and it's left child
// i.e. if current node is A and it's left child is B ( A --- >> B ),
//      then new cloned node with key A wil be created (say cA) and
//      it will be put as
//      A --- >> cA --- >> B
// Here B can be a NULL or a non-NULL left child
// Right child pointer will be set correctly
// i.e. if for current node A, right child is C in original tree
```

```
// (A --- >> C) then corresponding cloned nodes cA and cC will like
// cA ---- >> cC
Node* copyLeftRightNode(Node* treeNode)
{
    if (treeNode == NULL)
        return NULL;

    Node* left = treeNode->left;
    treeNode->left = newNode(treeNode->key);
    treeNode->left->left = left;
    if(left != NULL)
        left->left = copyLeftRightNode(left);

    treeNode->left->right = copyLeftRightNode(treeNode->right);
    return treeNode->left;
}

// This function sets random pointer in cloned tree as per original tree
// i.e. if node A's random pointer points to node B, then
// in cloned tree, cA wil point to cB (cA and cB are new node in cloned
// tree corresponding to node A and B in original tree)
void copyRandomNode(Node* treeNode, Node* cloneNode)
{
    if (treeNode == NULL)
        return;
    if(treeNode->random != NULL)
        cloneNode->random = treeNode->random->left;
    else
        cloneNode->random = NULL;

    if(treeNode->left != NULL && cloneNode->left != NULL)
        copyRandomNode(treeNode->left->left, cloneNode->left->left);
    copyRandomNode(treeNode->right, cloneNode->right);
}

// This function will restore left pointers correctly in
// both original and cloned tree
void restoreTreeLeftNode(Node* treeNode, Node* cloneNode)
{
    if (treeNode == NULL)
        return;
    if (cloneNode->left != NULL)
    {
        Node* cloneLeft = cloneNode->left->left;
        treeNode->left = treeNode->left->left;
        cloneNode->left = cloneLeft;
    }
    else
        treeNode->left = NULL;

    restoreTreeLeftNode(treeNode->left, cloneNode->left);
    restoreTreeLeftNode(treeNode->right, cloneNode->right);
}
```

```
//This function makes the clone of given tree
Node* cloneTree(Node* treeNode)
{
    if (treeNode == NULL)
        return NULL;
    Node* cloneNode = copyLeftRightNode(treeNode);
    copyRandomNode(treeNode, cloneNode);
    restoreTreeLeftNode(treeNode, cloneNode);
    return cloneNode;
}


/* 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 = 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.

Source

http://www.geeksforgeeks.org/clone-binary-tree-random-pointers/

Category: Trees Tags: Hashing

Post navigation

← MAQ Software Interview Experience | Set 2 Oracle Interview | Set 8 →

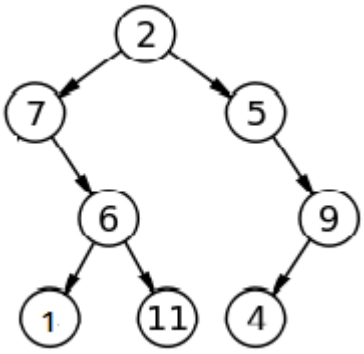
Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.

Chapter 91

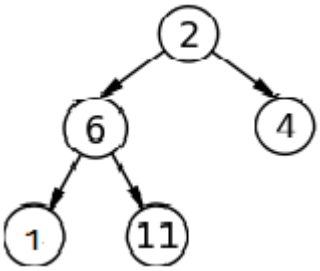
Given a binary tree, how do you remove all the half nodes?

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

Source

http://www.geeksforgeeks.org/given-a-binary-tree-how-do-you-remove-all-the-half-nodes/

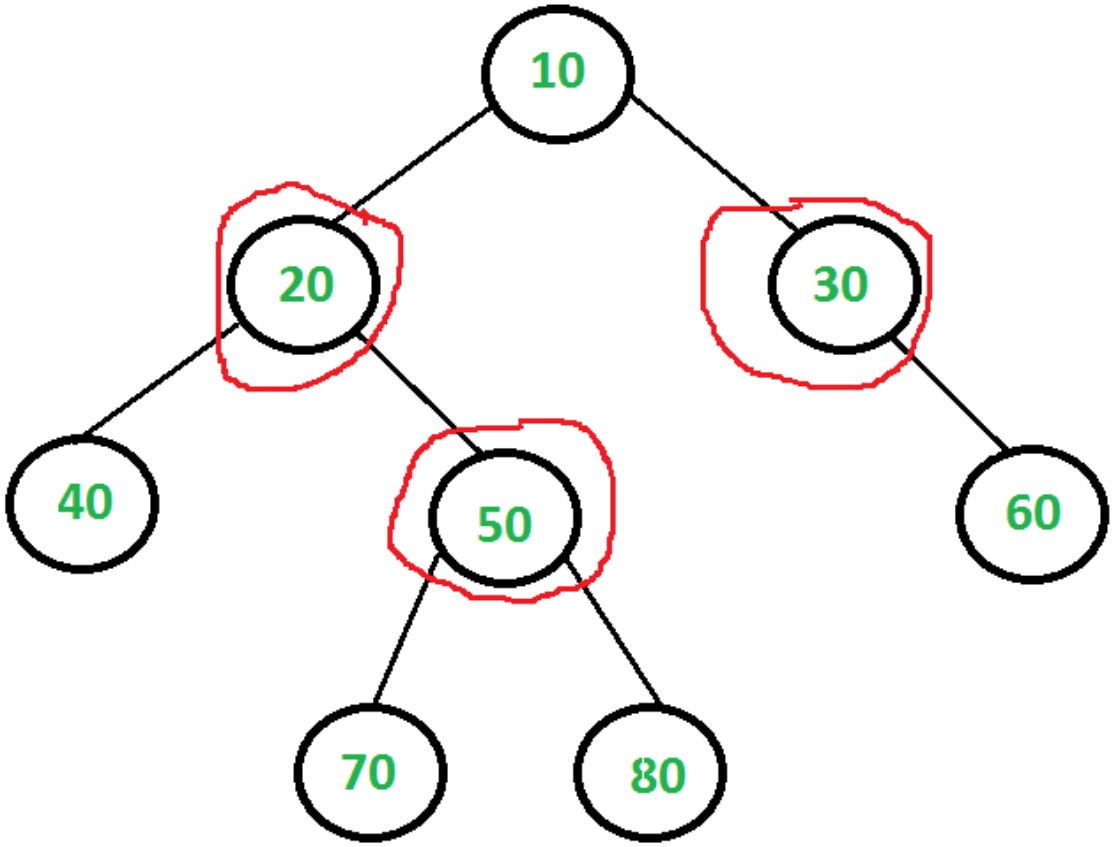
Category: Trees

Chapter 92

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 <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 there
    // 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 suproblems are called again, this problem has Overlapping Subprolems property. So Vertex Cover problem has both properties (see thisand 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 cover.
int vCover(struct node *root)
{
    // The size of minimum vertex cover is zero if tree is empty or there
    // 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 it
    // 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

Source

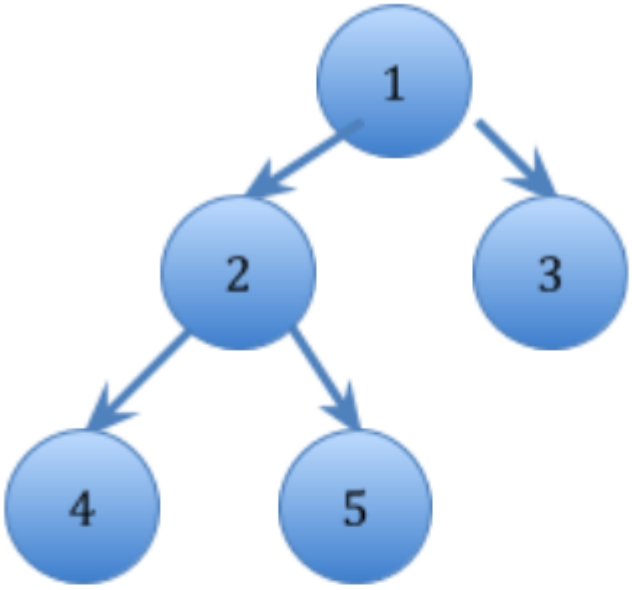
http://www.geeksforgeeks.org/vertex-cover-problem-set-2-dynamic-programming-solution-tree/

Chapter 93

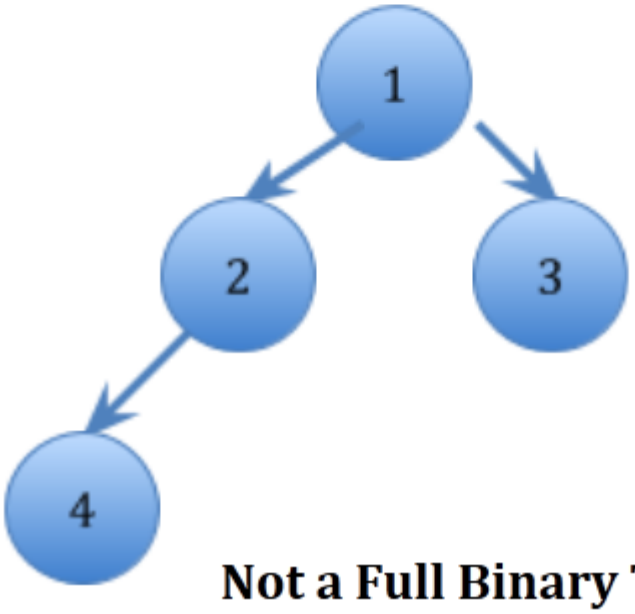
Check whether a binary tree is a full binary tree or not

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:



Full Binary Tree



Not a Full Binary Tree

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->right = node->left = 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;
    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

Source

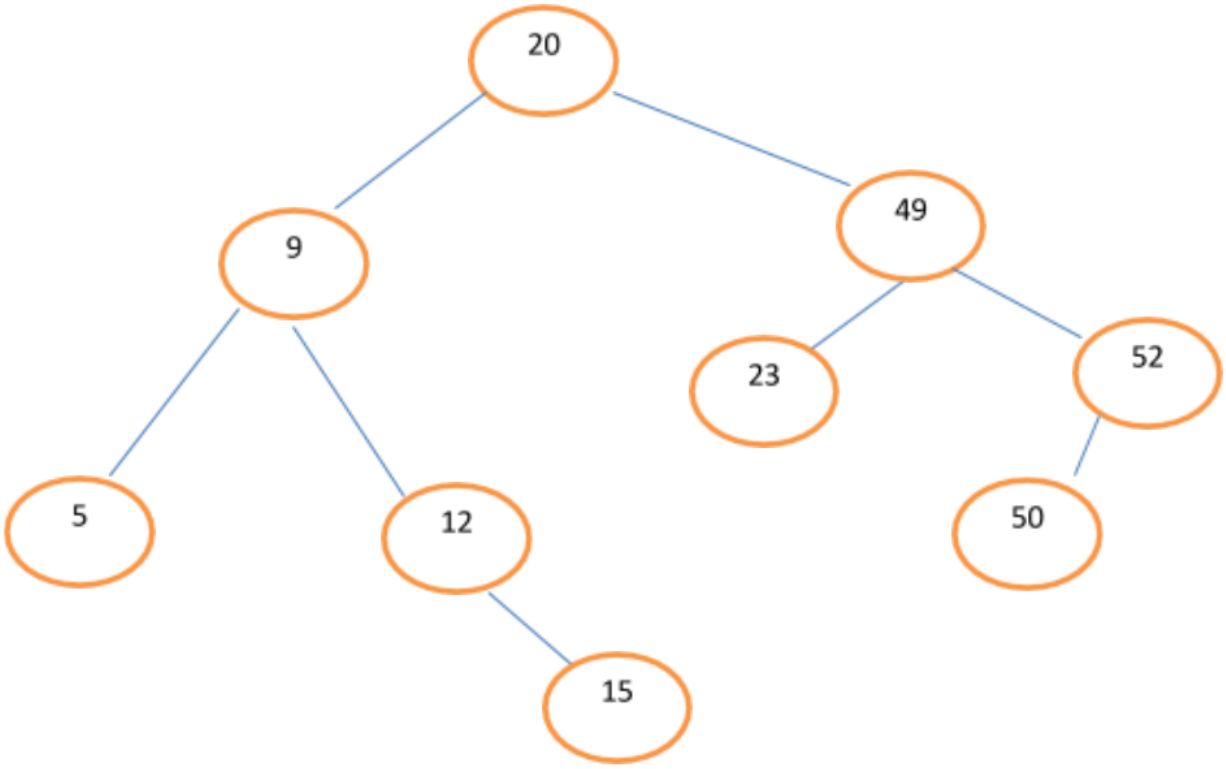
http://www.geeksforgeeks.org/check-whether-binary-tree-full-binary-tree-not/

Category: Trees

Chapter 94

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 userid 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

Source

http://www.geeksforgeeks.org/find-sum-left-leaves-given-binary-tree/

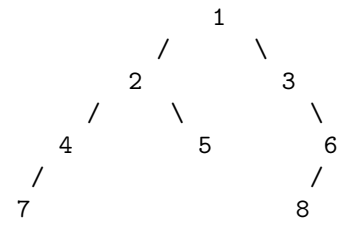
Category: Trees

Chapter 95

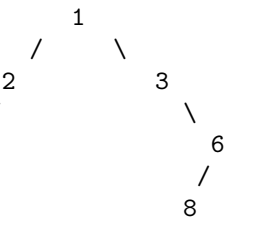
Remove nodes on root to leaf paths of length

Given a Binary Tree and a number k, remove all nodes that lie only on root to leaf path(s) of length smaller than k. If a node X lies on multiple root-to-leaf paths and if any of the paths has path length >= k, then X is not deleted from Binary Tree. In other words a node is deleted if all paths going through it have lengths smaller than k.

Consider the following example Binary Tree



```
Input: Root of above Binary Tree
       k = 4

Output: The tree should be changed to following
```



```
There are 3 paths
i) 1->2->4->7      path length = 4
ii) 1->2->5        path length = 3
iii) 1->3->6->8    path length = 4
There is only one path " 1->2->5 " of length smaller than 4.
The node 5 is the only node that lies only on this path, so
node 5 is removed.
Nodes 2 and 1 are not removed as they are parts of other paths
of length 4 as well.
```

If k is 5 or greater than 5, then whole tree is deleted.

If k is 3 or less than 3, then nothing is deleted.

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 >= 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 utitlity 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

Source

http://www.geeksforgeeks.org/remove-nodes-root-leaf-paths-length-k/

Category: Trees Tags: tree-traversal

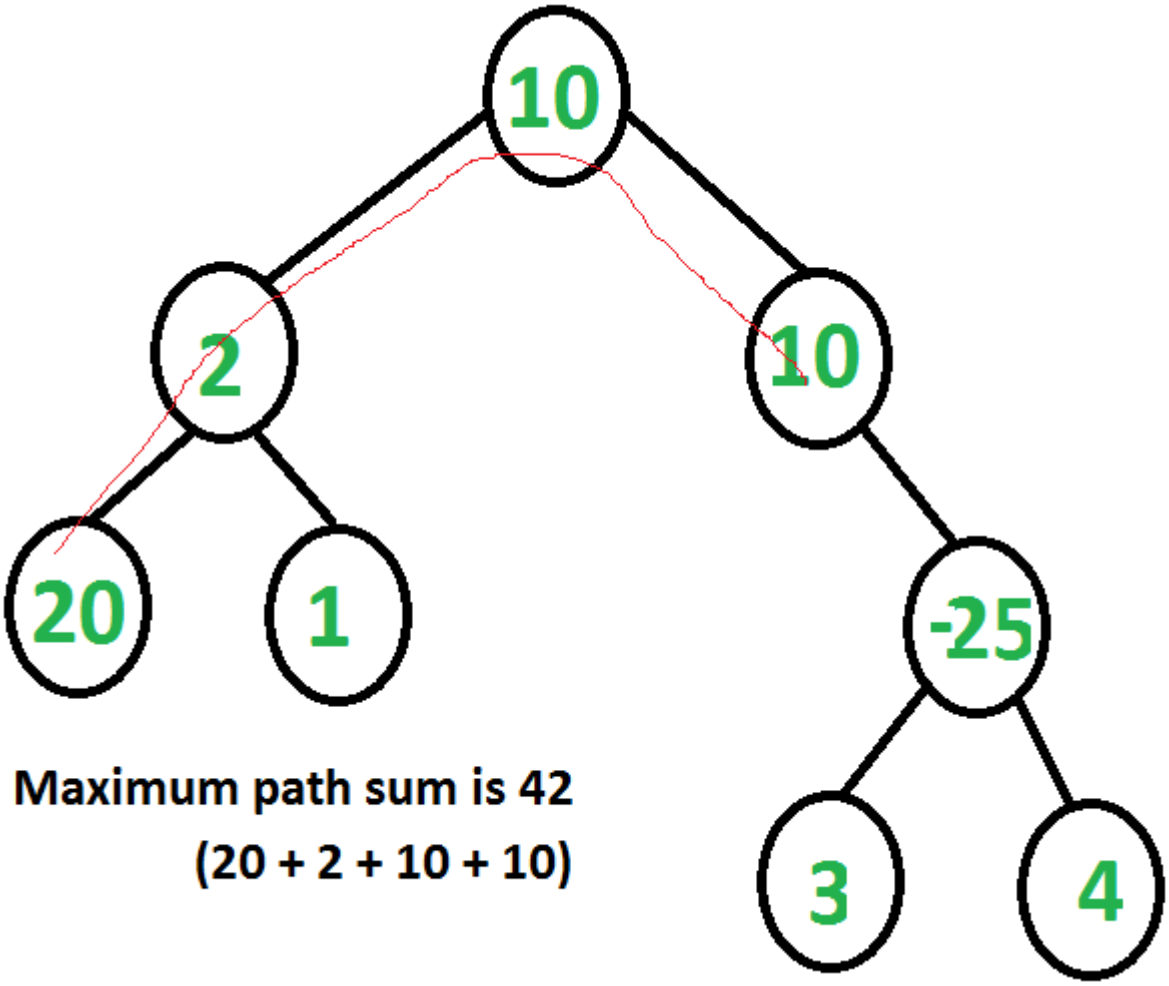
Chapter 96

Maximum Path Sum in a Binary Tree

Given a binary tree, find the maximum path sum. The path may start and end at any node in the tree.

Example:

```
Input: Root of below tree
       1
      / \
     2   3
Output: 6

See below diagram for another example.
1+2+3
```



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

For each node there can be four ways that the max path goes through the node:
1. Node only
2. Max path through Left Child + Node
3. Max path through Right Child + Node
4. Max path through Left Child + Node + Max path through Right Child

The idea is to keep trace of four paths and pick up the max one in the end. An important thing to note is, root of every subtree need to return maximum path sum such that at most one child of root is involved. This is needed for parent function call. In below code, this sum is stored in 'max_single' and returned by the recursive function.

```
// C program to remove all half Nodes
#include<bits/stdc++.h>
using namespace std;

// A binary tree node
struct Node
{
    int data;
    struct Node* left, *right;
```

```
};

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

// This function returns overall maximum path sum in 'res'
// And returns max path sum going through root.
int findMaxUtil(Node* root, int &res)
{
    //Base Case
    if (root == NULL)
        return 0;

    // l and r store maximum path sum going through left and
    // right child of root respectively
    int l = findMaxUtil(root->left,res);
    int r = findMaxUtil(root->right,res);

    // Max path for parent call of root. This path must
    // include at-most one child of root
    int max_single = max(max(l, r) + root->data, root->data);

    // Max Top represents the sum when the Node under
    // consideration is the root of the maxsum path and no
    // ancestors of root are there in max sum path
    int max_top = max(max_single, l + r + root->data);

    res = max(res, max_top); // Store the Maximum Result.

    return max_single;
}

// Returns maximum path sum in tree with given root
int findMaxSum(Node *root)
{
    // Initialize result
    int res = INT_MIN;

    // Compute and return result
    findMaxUtil(root, res);
    return res;
}

// Driver program
int main(void)
{
    struct Node *root = newNode(10);
    root->left        = newNode(2);
```

```
    root->right       = newNode(10);
    root->left->left  = newNode(20);
    root->left->right = newNode(1);
    root->right->right = newNode(-25);
    root->right->right->left   = newNode(3);
    root->right->right->right  = newNode(4);
    cout << "Max path sum is " << findMaxSum(root);
    return 0;
}
```

Output:

```
Max path sum is 42
```

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

This article is contributed by **Anmol Varshney** (FB Profile: https://www.facebook.com/anmolvarshney695). Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Source

http://www.geeksforgeeks.org/find-maximum-path-sum-in-a-binary-tree/

Category: Trees