Trees
Session 2

Tree Traversals

Depth First Traversals
(Inorder, Preorder and Postorder)

(a) Inorder Traversal (Left, Root, Right) :

1. Traverse the left subtree.
2. Visit the root.
3. Traverse the right subtree.

```
1. void printInorder(struct Node* node)
2. {
3.         if (node == NULL)
4.                 return;
5.
6.         printInorder(node->left);
7.         cout << node->data << " ";
8.         printInorder(node->right);
9. }
```

(b) Preorder Traversal (Root, Left, Right) :

1. Visit the root.
2. Traverse the left subtree.
3. Traverse the right subtree.

```
1. void printPreorder(struct Node* node)
2. {
3.         if (node == NULL)
4.                 return;
```

```
5.          cout << node->data << " ";
6.          printPreorder(node->left);
7.          printPreorder(node->right);
8.  }
```

(c) Postorder Traversal (Left, Right, Root) :

1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the root.

```
1.   void printPostorder(struct Node* node)
2.  {
3.          if (node == NULL)
4.                  return;
5.
6.          printPostorder(node->left);
7.          printPostorder(node->right);
8.          cout << node->data << " ";
9.  }
```

inorder-traversal

Postorder-traversal

preorder-traversal

Iterative Preorder Traversal

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

```
1.  void iterativePreorder(node* root)
2.  {
3.      if (root == NULL)
4.          return;
5.
6.      stack<node*>s;
7.      s.push(root);
8.
9.
10.     while (s.empty() == false)
11.     {
12.         struct node* Node =s.top();
13.         cout << Node->data;
14.         s.pop();
15.
16.         if (node->right)
17.             nodeStack.push(node->right);
18.         if (node->left)
19.             nodeStack.push(node->left);
20.     }
21. }
```

Iterative Inorder Traversal

Inorder Tree Traversal without Recursion

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.

```
1.  void inOrder(struct Node *root)
2.  {
3.          stack<Node *> s;
4.          Node *curr = root;
5.          while (curr != NULL || s.empty() == false)
6.          {
7.                  while (curr != NULL)
8.                  {
9.                          s.push(curr);
10.                         curr = curr->left;
11.                 }
12.                 curr = s.top();
13.                 s.pop();
14.                 cout << curr->data << " ";
15.                 curr = curr->right;
16.         }
17. }
```

Inorder Tree Traversal without recursion and without stack!

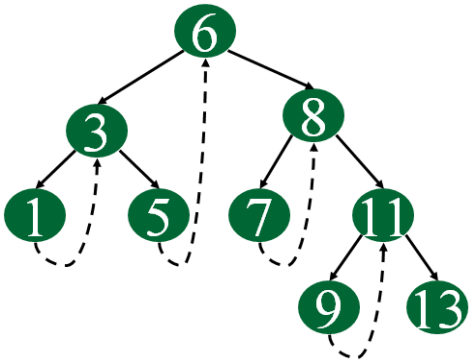
Threaded Binary Tree

Inorder traversal of a Binary tree can either be done using recursion or with the use of an auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

There are two types of threaded binary trees.

Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)

Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.



Morris Traversal

1. Initialize current as root

2. While current is not NULL

If the current does not have left child

a) Print current's data

b) Go to the right, i.e., current = current->right

Else

a) Find rightmost node in current left subtree OR

node whose right child == current.

If we found right child == current

a) Update the right child as NULL of that node whose right child is current

b) Print current's data

c) Go to the right, i.e. current = curent->right

Else

a) Make current as the right child of that rightmost node we found; and

b) Go to this left child, i.e., current = current->left

```
1.  void MorrisTraversal(struct tNode* root)
2.  {
3.          struct tNode *current, *pre;
4.          if (root == NULL)
5.                  return;
6.          current = root;
7.          while (current != NULL)
8.              {
9.                  if (current->left == NULL) {
10.                         printf("%d ", current->data);
11.                         current = current->right;
12.                 }
13.                 else
14.                 {
15.                         pre = current->left;
16.                         while (pre->right != NULL&& pre->right != current)
17.                                 pre = pre->right;
18.                         if (pre->right == NULL)
19.                         {
20.                                 pre->right = current;
21.                                 current = current->left;
22.                         }
23.                         else
24.                         {
25.                                 pre->right = NULL;
26.                                 printf("%d ", current->data);
27.                                 current = current->right;
28.                         }
29.}
30.}
```

Level Order Binary Tree Traversal

Method 1 Algorithm:

There are basically two functions in this method.

One is to print all nodes at a given level

another is to reach that level.

```
1.  void printLevelOrder(node* root)
2.  {
3.          int h = height(root);
4.          for (int i = 1; i <= h; i++)
5.                  printGivenLevel(root, i);
6.  }
7.  void printGivenLevel(node* root, int level)
8.  {
9.          if (root == NULL)
10.                 return;
11.         if (level == 1)
12.                 cout << root->data << " ";
13.         else if (level > 1)
14.         {
15.                 printGivenLevel(root->left, level-1);
16.                 printGivenLevel(root->right, level-1);
17.         }
18.}
19. int height(node* node)
20.{
21.         if (node == NULL)
22.                 return 0;
```

```
23.         else
24.         {
25.                 int lheight = height(node->left);
26.                 int rheight = height(node->right);
27.                 if (lheight > rheight)
28.                         return(lheight + 1);
29.                 else return(rheight + 1);
30.         }
31. }
```

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

2)**Method 2 (Using queue) Algorithm:**

1) Create an empty queue q
2) temp node = root
3) while(temp!=NULL)
   a) print temp_node->data.
   b) Enqueue temp_node's children (first left then right) to q
   c) Dequeue a node from q.

```
1.  void printLevelOrder(Node *root)
2.  {
3.          if (root == NULL)
4.            return;
5.
6.          queue<Node *> q;
```

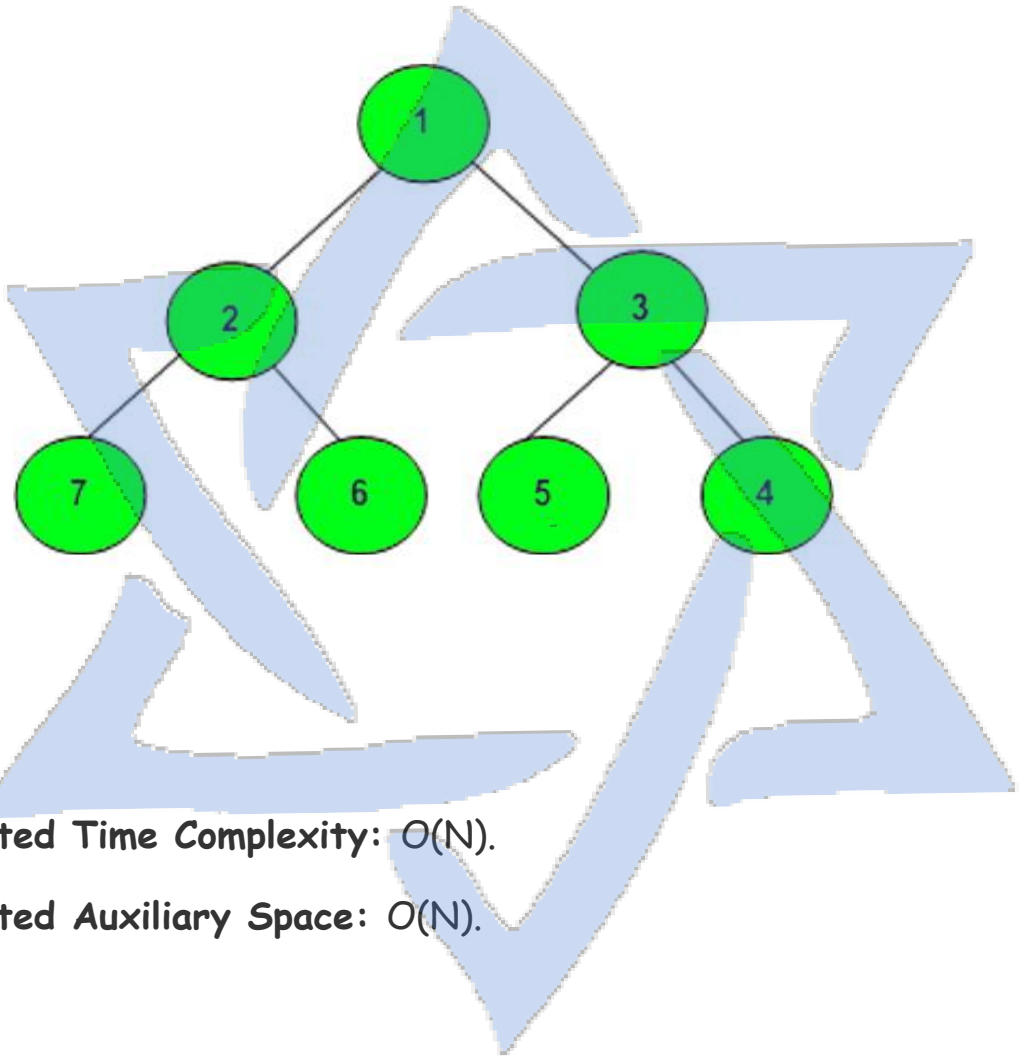
```
7.          q.push(root);
8.
9.          while (q.empty() == false)
10.         {
11.                 Node *node = q.front();
12.                 cout << node->data << " ";
13.                 q.pop();
14.
15.                 if (node->left != NULL)
16.                         q.push(node->left);
17.
18.                 if (node->right != NULL)
19.                         q.push(node->right);
20.         }
21. }
```

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

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

Level order traversal in spiral form

Complete the function to find spiral order traversal of a tree. For the tree below, function should return 1, 2, 3, 4, 5, 6, 7.



Expected Time Complexity: *O*(N).

Expected Auxiliary Space: *O*(N).

1) Take a bool and recursively do it .

```
1)  void printSpiral(struct node* root)
2)  {
3)    int h = height(root);
4)    int i;
5)    bool ltr = false;
6)    for(i = 1; i <= h; i++)
7)    {
8)      printGivenLevel(root, i, ltr);
9)      ltr = !ltr;
10)   }
11) }
12)
13)void printGivenLevel(struct node* root,
14)                int level, int ltr)
15){
16)   if (root == NULL)
17)     return;
18)   if (level == 1)
19)     cout << root->data << " ";
20)
21)   else if (level > 1)
22)     {
23)       if (ltr)
24)       {
25)         printGivenLevel(root->left,level - 1, ltr);
26)         printGivenLevel(root->right,level - 1, ltr);
27)       }
28)       else
```

```
29)         {
30)             printGivenLevel(root->right,level - 1, ltr);
31)         printGivenLevel(root->left,level - 1, ltr);
32)         }
33)     }
34)   }
35)
36)   int height(struct node* node)
37)   {
38)     if (node == NULL)
39)         return 0;
40)     else
41)  {
42)
43)         int lheight = height(node->left);
44)         int rheight = height(node->right);
45)
46)         if (lheight > rheight)
47)             return (lheight + 1);
48)         else
49)             return (rheight + 1);
50)     }
51)}
52)
53)
54)
```

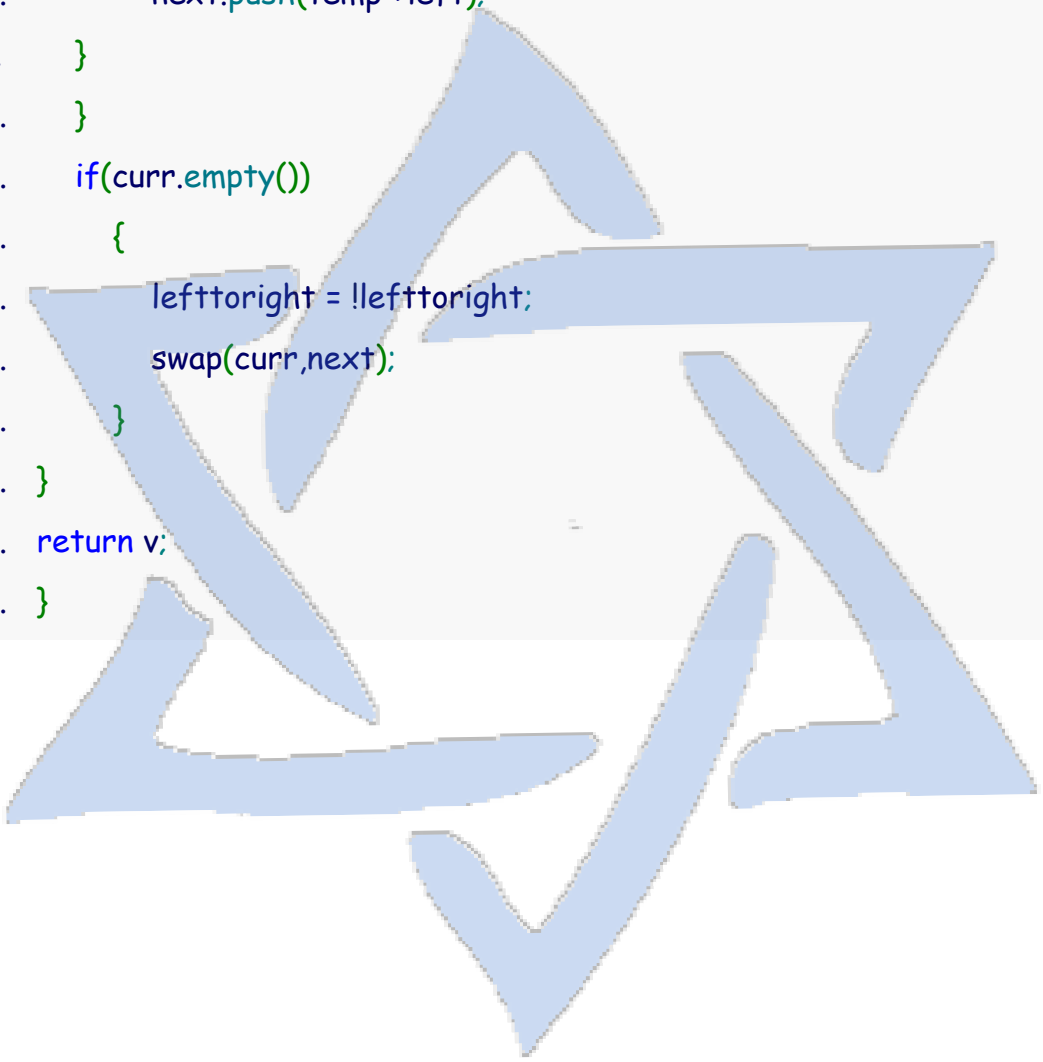
Time Complexity: Worst case time complexity of the above method is **O(n^2)**. Worst case occurs in case of skewed trees.

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.

```
1.  vector<int> findSpiral(Node *root)
2.  {
3.     vector<int>v;
4.     stack<Node*>curr;
5.     stack<Node*>next;
6.     if(root==NULL)
7.        return v;
8.     curr.push(root);
9.
10.    bool lefttoright=false;
11.    while(!curr.empty())
12.    {
13.       Node*temp=curr.top();
14.       curr.pop();
15.       if(temp)
16.       {
17.          v.push_back(temp->data);
18.       if(lefttoright)
19.       {
20.          if(temp->left)
21.             next.push(temp->left);
22.          if(temp->right)
```

```
23.             next.push(temp->right);
24.     }
25.     else
26.     {
27.         if(temp->right)
28.             next.push(temp->right);
29.         if(temp->left)
30.             next.push(temp->left);
31.     }
32.     }
33.     if(curr.empty())
34.         {
35.             lefttoright = !lefttoright;
36.             swap(curr,next);
37.         }
38. }
39. return v;
40. }
```



Reverse Level Order Traversal

Given a binary tree of size N, find its reverse level order traversal. ie- the traversal must begin from the last level.

Example :

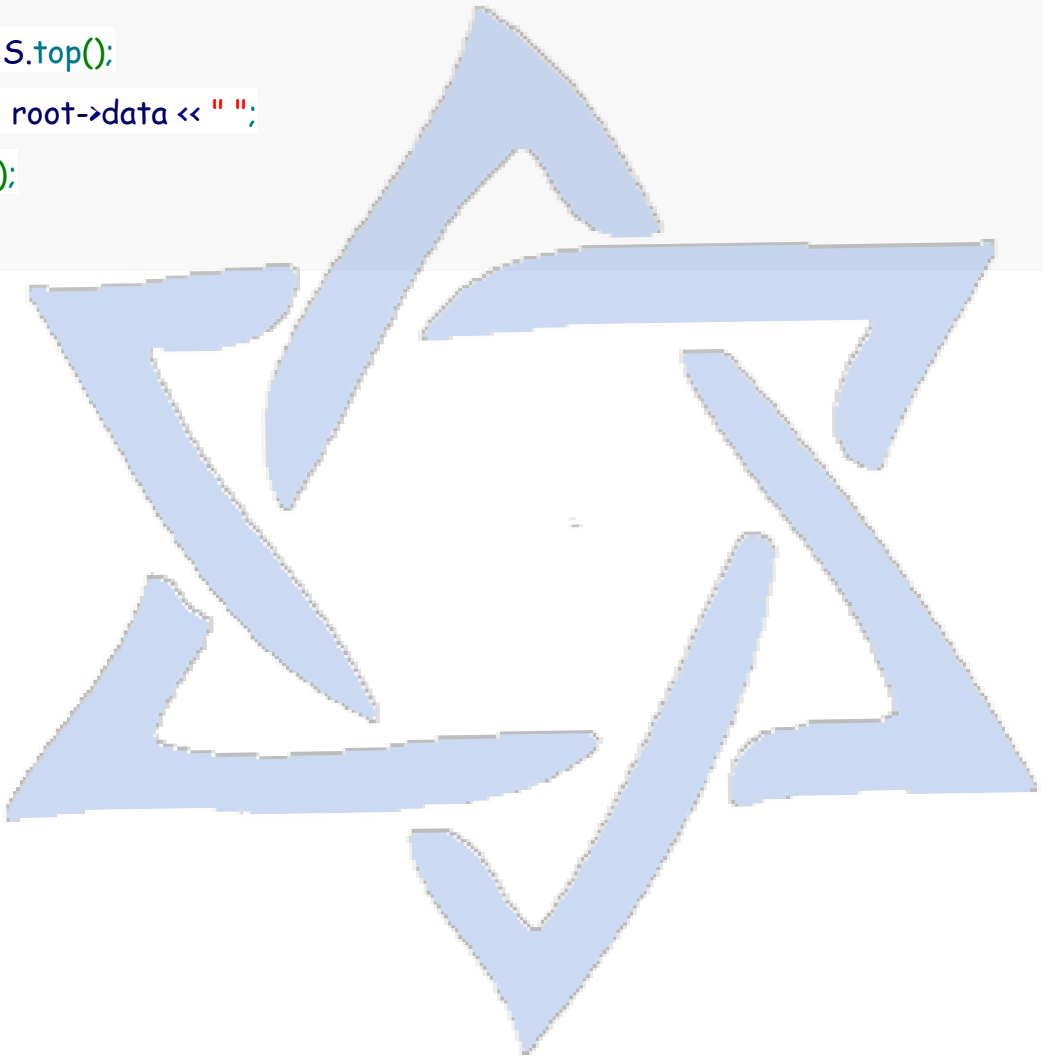
```
Input :
        10
       /  \
      20   30
     / \
    40  60

Output: 40 60 20 30 10
```

METHOD 1 (Recursive function to print a given level)

METHOD 2 (Using Queue and Stack)

```
1.  void reverseLevelOrder(node* root)
2.  {
3.    stack <node *> S;
4.    queue <node *> Q;
5.    Q.push(root);
6.
7.      while (Q.empty() == false)
8.    {
9.       root = Q.front();
10.     Q.pop();
11.     S.push(root);
12.
13.       if (root->right)
```

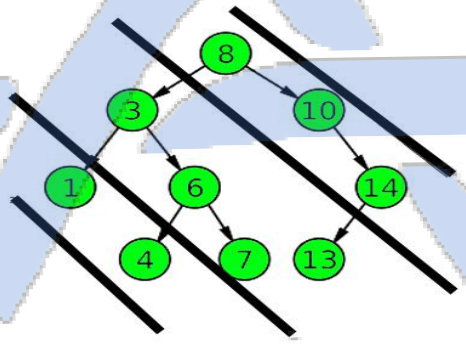
```
14.          Q.push(root->right);
15.
16.       if (root->left)
17.          Q.push(root->left);
18.    }
19.
20.     while (S.empty() == false)
21.    {
22.       root = S.top();
23.       cout << root->data << " ";
24.       S.pop();
25.    }
```



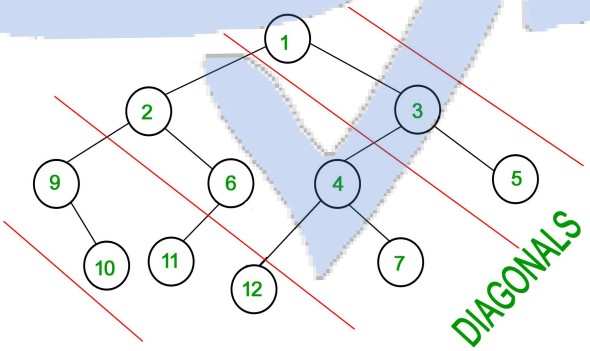
Diagonal Sum of a Binary Tree

Diagonal Traversal of Binary Tree

Given a Binary Tree, print the **diagonal traversal** of the binary tree.Consider lines of slope -1 passing between nodes. Given a Binary Tree, print all diagonal elements in a binary tree belonging to the same line.



Diagonal Traversal of binary tree : 8 10 14 3 6 7 13 1 4



```
1.  void diagonalPrintUtil(Node *root, int level,int index, map<int,
    vector<int>>&mp)
2.  {
3.     if (!root)
4.        return;
5.
6.     mp[level-index].push_back(root->data);
7.     diagonalPrintUtil(root->left,level+1,index-1,mp);
8.     diagonalPrintUtil(root->right,level+1,index+1,mp);
9.  }
10. vector<int> diagonal(Node *root)
11. {
12.    vector<int> V;
13.    map<int, vector<int>>mp;
14.    diagonalPrintUtil(root,0, 0, mp);
15.
16.    for(auto it:mp)
17.    {
18.       vector<int> v=it.second;
19.       for(int i=0;i<v.size();i++)
20.       {
21.          V.push_back(v[i]);
22.       }
23.    }
24.    return V;
25.  // your code here
26.}
```