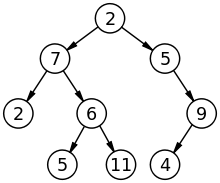
**Trees**

One of the most striking and widely used feature in data structures is Tree. In this note you are going learn about tree. And I am sure that by the end of the tutorial you will be able to clearly figure out the concepts of trees and I will discuss some of the classical problems on treesSo lets start with our discussion on trees.

How typically a tree looks like in data structure. Here is a sample-



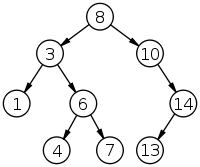
**Some terminologies used in trees:**

* **Root** – The top node in a tree.
* **Child** – The just next nodes connected downwards.
* **Parent** – The converse notion of child.
* **Siblings** – Nodes with the same parent.
* **Descendant** – a node reachable by repeated proceeding from parent to child.
* **Ancestor** – a node reachable by repeated proceeding from child to parent.
* **Leaf** – a node with no children.
* **Internal node** – a node with at least one child.
* **External node** – a node with no children.

**Binary Search Tree**

Now you might be surprised why used "search" keyword here. There is not much difference in terms of looks between binary tree and binary search tree. The difference is that the left sub tree nodes will have value smaller than that of node and the right sub tree nodes will have value greater than that of node.This allows searching in O(log n) for balanced binary search tree.

How typically a tree looks like in data structure. Here is a sample-



**Application of Trees**

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

**How trees are declared in programming?**

struct node

{

int data; //the element

struct node\*left; //pointer to left node

struct node\*right; //pointer to right node

};

**Creation of node.**

*Simple node*

Struct node root;

*Pointer to a node*

Struct node\*root;

root=(node\*)malloc(sizeof(node));

In the case of pointer to node, we have to explicitly allocate memory of node type to the pointer. (this method is preferable).

*Utility Function Returning Node*

struct node\*newnode(int element)

{

struct node\*temp=(node\*)malloc(sizeof(node));

temp->data=element;

temp->left=temp->right=NULL;

return temp;

}

**Insertion**

*Recursive Insert*

struct node\* insert(struct node\* root, int data)

{

if (root == NULL) //If the tree is empty, return a new,single node

return newNode(data);

else

{

//Otherwise, recur down the tree

if (data <= root->data)

root->left = insert(root->left, data);

else

root->right = insert(root->right, data);

//return the (unchanged) root pointer

return root;

}

}

*Iterative Insert*

struct node\* insert(struct node\* root, int data)

{

struct node \*n = newNode(data);

if (node == NULL)

return n;

struct node \*temp = root;

while(temp)

{

if (data < temp->data)

{

if (temp->left == NULL)

{

temp->left = n;

break;

}

temp = temp->left;

}

else if (data > temp->data)

{

if (temp->right == NULL)

{

temp->right = n;

break;

}

temp = temp->right;

}

else

break;

}

return root;

}

**Traversal**

There are mainly *three* types of tree traversal. But we consider one more type of traversal i.e. level order traversal.

*PreOrder Traversal* - In this traversal technique the traversal order is root-left-right.

void perorder(struct node\*root)

{

if(root)

{

printf("%d ",root->data); //Printf root->data

preorder(root->left); //go to left sub tree

preorder(root->right); //go to right sub tree

}

}

*InOrder Traversal* - In this traversal technique the traversal order is left-root-right. This is the only traversal in BST which gives sorted order data.

void inorder(struct node\*root)

{

if(root)

{

inorder(root->left); //go to left sub tree

printf("%d ",root->data); //Printf root->data

inorder(root->right); //go to right sub tree

}

}

*PostOrder Traversal* - In this traversal technique the traversal order is root-left-right.

void postorder(struct node\*root)

{

if(root)

{

postorder(root->left); //go to left sub tree

postorder(root->right); //go to right sub tree

printf("%d ",root->data); //Printf root->data

}

}

*Level Order Traversal* - In this traversal technique we print tree level wise. Level order traversal can be done using a queue, The concept is to enqueue the root of the tree and print the front element of the queue and enqueue the left and then right node of the tree perform the same while queue is not empty. For eg : The level order traversal for the BST shown above is - 8 3 10 1 6 14 4 7 13

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);

}

}

Time Complexity:O(n)

**Maximum Depth or Height of a Tree**

The idea is to do a postorder traversal and maintaining two variables to store left depth and right depth and return max of left and right depth.

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);

}

}

Time Complexity:O(n)

**Tree to its Mirror Tree**

The idea is to traverse the tree in postorder and after calling left and right subtree swap the pointers of the node.

void mirror(struct node\* node)

{

if (node)

{

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;

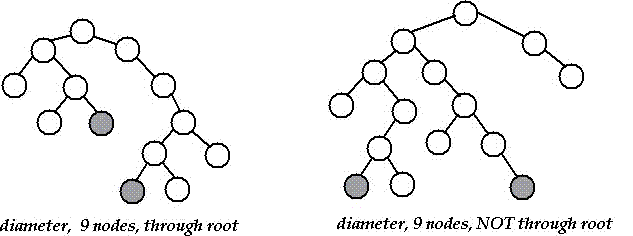
}

}

Time Complexity:O(n)

**Diameter of a 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).



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 = maxDepth(tree->left);

int rheight = maxDepth(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));

}

Time Complexity:O(n^2)

**Maximum Width of Binary Tree**

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.

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);

}

}

Time Complexity:O(n^2)

**Printing root to leaf paths one per line**

*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)

*Program:*

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 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");

}

Time Complexity:O(n)

**Print nodes at k distance from root**

Idea is to modify preorder traversal with maintaining a variable k, and with every iteration check it k is equal to zero if so then print that element, if not the decrement k by one.

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 ) ;

}

}

Time Complexity:O(n)

**Check if binary tree is BST or not.**

While doing In-Order traversal, we can keep track of previously visited node. If the value of the currently visited node is less than the previous value, then tree is not BST.

int isBST(struct node\* root)

{

static struct node \*prev = NULL;

// traverse the tree in inorder fashion and keep track of prev node

if (root)

{

if (!isBST(root->left))

return 0;

// Allows only distinct valued nodes

if (prev != NULL && root->data <= prev->data)

return 0;

prev = root;

return isBST(root->right);

}

return true;

}

Time Complexity:O(n)