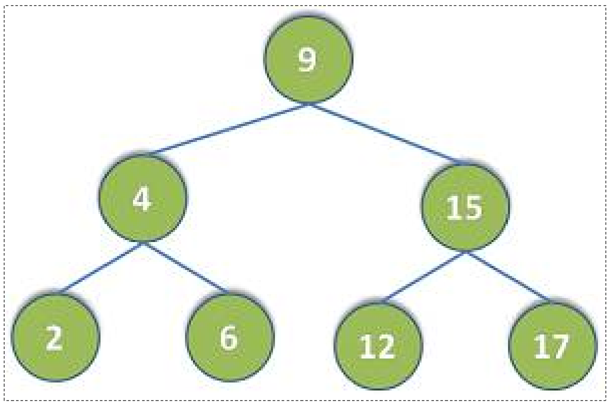
Binary Tree

- In computer science, a **binary tree** is a tree data structure in which each node has at most two children, which are referred to as the ***left*** **child** and the ***right*** **child**.

- Binary tree is the data structure to maintain data into memory of program. There exists many data structures, but they are chosen for usage on the basis of time consumed in insert/search/delete operations performed on data structures.

- Binary tree is one of the data structures that are efficient in insertion and searching operations. Binary tree works on **O(log N)** for insert/search/delete operations.

- Binary tree is basically tree in which each node can have two child nodes and each child node can itself be a small binary tree. To understand it, below is the example figure of binary tree.



Note: Binary tree works on the rule that child nodes which are lesser than root node keep on the left side and child nodes which are greater than root node keep on the right side.

We will understand binary tree through its operations.

- **Create binary tree**
- **Search into binary tree**
- **Delete binary tree**
- **Displaying binary tree**

Creation of binary tree

Binary tree is created by inserting root node and its child nodes.

```
void insert(node ** tree, int val) {
node *temp = NULL;
if(!(*tree)) {
  temp = (node *)malloc(sizeof(node));
  temp->left = temp->right = NULL;
  temp->data = val;
  *tree = temp;
  return;
}

if(val < (*tree)->data) {
```

```
22      insert(&(*tree)->left, val);
23   } else if(val > (*tree)->data) {
24     insert(&(*tree)->right, val);
25   }
26 }
```

This function would determine the position as per value of node to be added and new node would be added into binary tree.

[Lines 13-19] Check first if tree is empty, then insert node as root.

[Line 21] Check if node value to be inserted is lesser than root node value, then

- a. **[Line 22]** Call insert() function recursively while there is non-NULL left node
- b. **[Lines 13-19]** When reached to leftmost node as NULL, insert new node.

[Line 23] Check if node value to be inserted is greater than root node value, then

- a. **[Line 24]** Call insert() function recursively while there is non-NULL right node
- b. **[Lines 13-19]** When reached to rightmost node as NULL, insert new node.

Searching into binary tree

Searching is done as per value of node to be searched whether it is root node or it lies in left or right sub-tree.

```
46 node* search(node ** tree, int val) {
47 if(!(*tree)) {
48   return NULL;
49  }
50 if(val == (*tree)->data) {
51   return *tree;
52  } else if(val < (*tree)->data) {
53    search(&((*tree)->left), val);
54  } else if(val > (*tree)->data){
55    search(&((*tree)->right), val);
56  }
57 }
```

This search function would search for value of node whether node of same value already exists in binary tree or not. If it is found, then searched node is returned otherwise NULL (i.e. no node) is returned.

1. **[Lines 47-49]** Check first if tree is empty, then return NULL.
2. **[Lines 50-51]** Check if node value to be searched is equal to root node value, then return node
3. **[Lines 52-53]** Check if node value to be searched is lesser than root node value, then call search() function recursively with left node
4. **[Lines 54-55]** Check if node value to be searched is greater than root node value, then call search() function recursively with right node
5. Repeat step 2, 3, 4 for each recursion call of this search function until node to be searched is found.

Deletion of binary tree

Binary tree is deleted by removing its child nodes and root node.

```
38 void deltree(node * tree) {
39 if (tree) {
40   deltree(tree->left);
41   deltree(tree->right);
42   free(tree);
43  }
44 }
```

This function would delete all nodes of binary tree in the manner – **left node**, **right node** and **root node**.

[Line 39] Check first if root node is non-NULL, then

- a. **[Line 40]** Call deltree() function recursively while there is non-NULL left node
- b. **[Line 41]** Call deltree() function recursively while there is non-NULL right node
- c. **[Line 42]** Delete the node.

Displaying binary tree

Binary tree can be displayed in three forms – **pre-order**, **in-order** and **post-order**.

- **Pre-order** displays **root node**, **left node** and then **right node**.
- **In-order** displays **left node**, **root node** and then **right node**.
- **Post-order** displays **left node**, **right node** and then **root node**.

Below is the code snippet for display of binary tree.

```
void print_preorder(node * tree) {
if (tree) {
printf("%d\n",tree->data);
print_preorder(tree->left);
print_preorder(tree->right);
}
}
void print_inorder(node * tree) {
if (tree) {
print_inorder(tree->left);
printf("%d\n",tree->data);
print_inorder(tree->right);
}
}
void print_postorder(node * tree) {
if (tree) {
print_postorder(tree->left);
print_postorder(tree->right);
printf("%d\n",tree->data);
}
}
```

These functions would display binary tree in pre-order, in-order and post-order respectively.

Pre-order display

- a. **[Line 30]** Display value of root node.
- b. **[Line 31]** Call print_preorder() function recursively while there is non-NULL left node
- c. **[Line 32]** Call print_preorder() function recursively while there is non-NULL right node

In-order display

- a. **[Line 37]** Call print_inorder() function recursively while there is non-NULL left node
- b. **[Line38]** Display value of root node.
- c. **[Line 39]** Call print_inorder() function recursively while there is non-NULL right node

Post-order display

- a. **[Line 44]** Call print_postorder() function recursively while there is non-NULL left node
- b. **[Line 45]** Call print_postorder() function recursively while there is non-NULL right node
- c. **[Line46]** Display value of root node.

Working program:

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

struct bin_tree {
int data;
struct bin_tree * right, * left;
};
typedef struct bin_tree node;

void insert(node ** tree, int val)
{
    node *temp = NULL;
    if(!(*tree))
    {
        temp = (node *)malloc(sizeof(node));
        temp->left = temp->right = NULL;
        temp->data = val;
        *tree = temp;
        return;
    }

    if(val < (*tree)->data)
    {
        insert(&(*tree)->left, val);
    }
    else if(val > (*tree)->data)
    {
        insert(&(*tree)->right, val);
    }

}
```

```
void print_preorder(node * tree)
{
    if (tree)
    {
        printf("%d\n",tree->data);
        print_preorder(tree->left);
        print_preorder(tree->right);
    }

}

void print_inorder(node * tree)
{
    if (tree)
    {
        print_inorder(tree->left);
        printf("%d\n",tree->data);
        print_inorder(tree->right);
    }
}

void print_postorder(node * tree)
{
    if (tree)
    {
        print_postorder(tree->left);
        print_postorder(tree->right);
        printf("%d\n",tree->data);
    }
}

void deltree(node * tree)
{
    if (tree)
    {
        deltree(tree->left);
        deltree(tree->right);
        free(tree);
    }
}

node* search(node ** tree, int val)
{
    if(!(*tree))
    {
        return NULL;
    }

    if(val < (*tree)->data)
    {
        search(&((*tree)->left), val);
    }
    else if(val > (*tree)->data)
    {
        search(&((*tree)->right), val);
    }
    else if(val == (*tree)->data)
    {
        return *tree;
    }
}

void main()
{
    node *root;
    node *tmp;
    //int i;

    root = NULL;
    /* Inserting nodes into tree */
    insert(&root, 9);
    insert(&root, 4);
    insert(&root, 15);
    insert(&root, 6);
    insert(&root, 12);
    insert(&root, 17);
    insert(&root, 2);

    /* Printing nodes of tree */
    printf("Pre Order Display\n");
    print_preorder(root);

    printf("In Order Display\n");
    print_inorder(root);

    printf("Post Order Display\n");
    print_postorder(root);

    /* Search node into tree */
    tmp = search(&root, 4);
    if (tmp)
    {
        printf("Searched node=%d\n", tmp->data);
    }
    else
    {
        printf("Data Not found in tree.\n");
    }

    /* Deleting all nodes of tree */
    deltree(root);
}
```

Output of Program:

```
$ ./a.out
Pre Order Display
9
4
2
6
15
12
17
In Order Display
2
4
6
9
12
15
17
Post Order Display
2
6
4
12
17
15
9
Searched node=4
```