BINARY TREE – CREATION & TRAVERSAL

Binary Tree

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.

Recursively, a (non-empty) binary tree is a tuple (L, S, R), where L and R are binary trees or the empty set and S is a singleton set.

Types of Binary Trees

- A **rooted** binary tree has a root node and every node has at most two children.
- A **full** binary tree (also a **proper** or **plane** binary tree) is a tree in which every node has either 0 or 2 children.
- A **perfect** binary tree is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.
- In a **complete** binary tree every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible. It can have between **1** and 2^h nodes at the last level **h**. A complete binary tree can be efficiently represented using an array.
- A **balanced** binary tree has the minimum possible maximum height (a.k.a. depth) for the leaf nodes because, for any given number of leaf nodes, the leaf nodes are placed at the greatest height possible.

Properties of Binary Tree

- The number of nodes **n** in a full binary tree, is at least **n=2h+1** and at most $n=2^{h+1}-1$, where **h** is the height of the tree. A tree consisting of only a **root** node has a height of **0**.
- The number of leaf nodes **l** in a perfect binary tree, is **l=(n+1)/2.**
- A perfect binary tree with **l** leaves has **n=2l-1** nodes.
- In a balanced full binary tree,

$$h = \lceil \log_2(l) \rceil + 1 = \lceil \log_2((n+1)/2) \rceil + 1 = \lceil \log_2(n+1) \rceil$$

- The number of internal nodes in a **complete** binary tree of **n** nodes is **floor(n/2)**.

BINARY TREE – CREATION & TRAVERSAL

Manual Way of Creating a Binary Tree

```
/*  Creating and Traversing an Arbitrary Binary Tree.
    [ The user is required to visualize and configure the Binary tree]    */

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

    struct treeNode {
        int data;
        struct treeNode *lchild;
        struct treeNode *rchild;
    };
    typedef struct treeNode tNode;
    typedef tNode* tree;

    void inOrderT(tree);
    tree createNode(int );

int main(){
    tree root = NULL;
    /*  binTree[] := {40,10,70,NULL,30,60,NULL,NULL,NULL,20}   */
    root = createNode(40);
    root->lchild = createNode(10);
    root->rchild = createNode(70);
    (root->lchild)->rchild = createNode(30);
    (root->rchild)->lchild = createNode(60);
    ((root->lchild)->rchild)->lchild = createNode(20);
    printf("\nInorder Traversal...  : ");
    inOrderT(root);
    printf("\n");
    return 0;
}

tree createNode(int key){
    tree neww;
    neww = (tree) malloc(sizeof(tNode));
    /*Assume that treeNode is ALWAYS returned.. */
    neww->data = key;
    neww->lchild = NULL;
    neww->rchild = NULL;
    return neww;
}
```

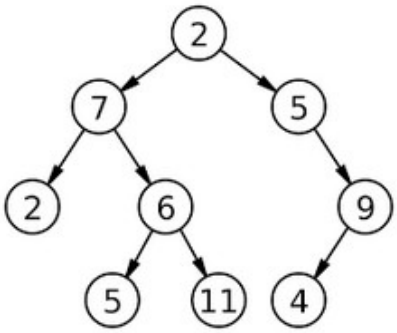
BINARY TREE – CREATION & TRAVERSAL

Consider a rooted binary tree of depth = 3.

Thus, the maximum nodes = 2^{3+1}-1 = 15

This tree can be represented using an array as -

[2, 7, 5, 2, 6, Null, 9, Null, Null, 5, 11, Null, Null, 4, Null]



It should be noted that,

- No of internal nodes = floor(n/2)=floor(9/2)= 4
- For each non-leaf node (root and internal node) located at index **i**, its left child is present at $\mathbf{left_{index}=2*i+1}$, and the right child is present at $\mathbf{right_{index}=2*i+2}$, respectively.

The **node structure** for a binary tree is defined as -

```
struct treeNode {
    int data;
    struct treeNode *lchild;
    struct treeNode *rchild;
};
typedef struct treeNode tNode;
typedef tNode* tree;
```

Let us assume a statically allocated array representing a binary tree whose last element is indicated by NDT [say Null] and the end-of-list is indicated by MAXVAL [say 999] and the non-existing tree nodes [say empty nodes] are indicated by NDT. [Here the empty nodes in the last level are represented using a single extra NDT lying at farthest left in the level]. The utility routines to find length of an array storing a binary tree, retrieve the index position of left-child and right-child are -

```
int lengthList(int list[]){
    int i = 0;
    while(list[i] != MXVAL)
        i = i + 1;
    return i-1;
}

int getLeftChild(int list[], int len, int ndx){
    int ele;
    if((2*ndx+1) > len)
        return len;          /* to be elaborated */
    return (2*ndx+1);
}
```

```
int getRightChild(int list[], int len, int ndx){
      /* students to be complete */
}
```

Building a Binary Tree

Now, the routine to building a binary tree from its array represention can be implemented as -

```
tree buildTree(int list[], int len, int ndx){
    tree temp = NULL;
    if(list[ndx] != NDT){
        temp = (tree) malloc(sizeof(tNode));
        temp->lchild = buildTree(list,len, getLeftChild(list, len, ndx));
        temp->data = list[ndx];
        temp->rchild = buildTree(list,len, getRightChild(list, len, ndx));
    }
    return temp;
}
```

Traversals

A rooted binary tree can be traversed the following most common ways –

- the inorder traversal [leftChild – root – rightChild]
- the postorder traversal [leftChild – rightChild – root] and
- the preorder traversal [root – leftChild – rightChild]

Recursive definitions for inOrder traversal can be implemented as -

```
void inOrderT(tree root){
    if(root != NULL) {
        inOrderT(root->lchild);
        printf("%4d", root->data);
        inOrderT(root->rchild);
    }
}
```

```
void postOrderT(tree root){
    /* students to be complete */
}
```

```
void preOrderT(tree root){
   /* students to be complete */
}
```

BINARY TREE – CREATION & TRAVERSAL

```
/* Write a routine to take tree as input into an array. */
```

Test Cases:

1. Left-skewed rooted binary tree with depth 4.
2. Right-skewed rooted binary tree with depth 3.
3. Combination of left-skewed and right-skewed binary tree of depth 3.
4. A regular full binary tree of depth 3.