Trees
Session 4

BINARY TREE CREATION

Construct a complete binary tree from given array in level order fashion

Given an array of elements, our task is to construct a complete binary tree from this array in level order fashion. That is, elements from left in the array will be filled in the tree level wise starting from level 0.

Examples:

Input : arr[] = {1, 2, 3, 4, 5, 6}
Output : Root of the following tree

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

Approach

If we observe carefully we can see that if parent node is at index i in the array then the left child of that node is at index (2*i + 1) and right child is at index (2*i + 2) in the array.
Using this concept, we can easily insert the left and right nodes by choosing its parent node. We will insert the first element present in the array as the root node at level 0 in the tree and start traversing the array and for every node i we will insert its both child's left and right in the tree.

Code

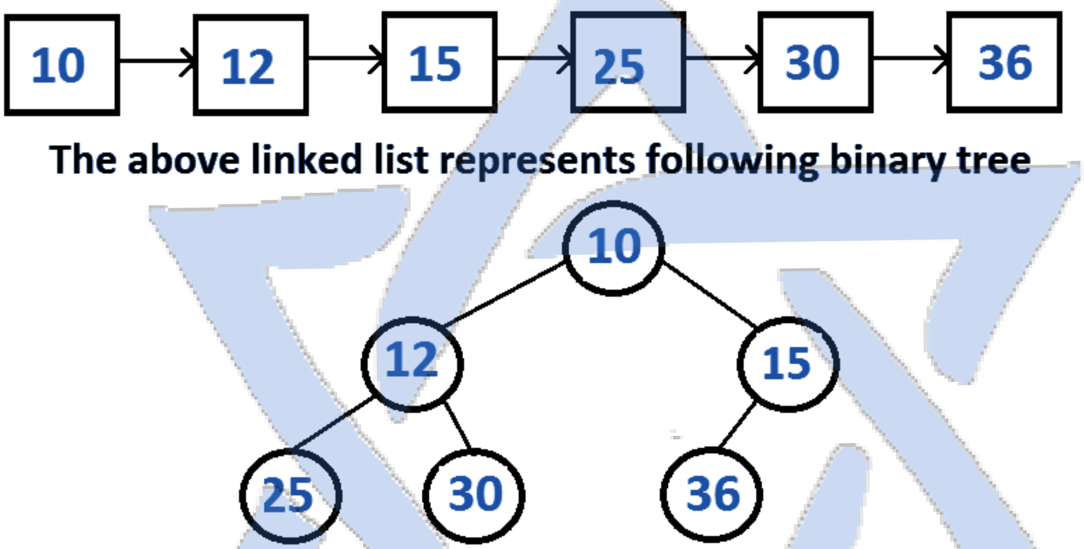
```
 Node* newNode(int data)
{
   Node* node = (Node*)malloc(sizeof(Node));
   node->data = data;
   node->left = node->right = NULL;
   return (node);
}
// Function to insert nodes in level order
Node* insertLevelOrder(int arr[], Node* root , int i, int n)
{
   if (i < n)
   {
      Node* temp = newNode(arr[i]);
      root = temp;

      // insert left child
      root->left = insertLevelOrder(arr,  root->left, 2 * i + 1, n);
      // insert right child
      root->right = insertLevelOrder(arr, root->right, 2 * i + 2, n);
   }
   return root;
}
```

Time Complexity: O(n), where n is the total number of nodes in the tree.

Construct Complete Binary Tree from its Linked List Representation

Given a Linked List Representation of Complete Binary Tree. The task is to construct the Binary tree.

Note : The complete binary tree is represented as a linked list in a way where if root node is stored at position i, its left, and right children are stored at position 2*i+1, 2*i+2 respectively.



Example 1:

Input:
N = 5
K = 1->2->3->4->5
Output: 1 2 3 4 5
Explanation: The tree would look like

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

Now, the level order traversal ofthe above tree is 1 2 3 4 5.

Approach

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.

Catalan Number

From a given number of '**n**' nodes we can create Catalan Number of trees.

Catalan Number, $T(n) = \mathbf{(\ {}^{2n}C_{n} / (n+1)\ )}$

Also, $T(n) = \sum_{i=1}^{n} (T(i-1) * T(n-i))$

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.It depends on what traversals are given. If one of the traversal methods is Inorder then the tree can be constructed, otherwise not.

Therefore, following combination can uniquely identify a tree.

Inorder and Preorder.

Inorder and Postorder.

Inorder and Level-order.

And the following do not.

Postorder and Preorder.

Preorder and Level-order.

Postorder and Level-order.

Construct Tree from given Inorder and Preorder traversals.

Given 2 Arrays of Inorder and preorder traversal. Construct a tree and print the Postorder traversal.

Example -1

```
Input:
N = 4
inorder[] = {1 6 8 7}
preorder[] = {1 6 7 8}
Output: 8 7 6 1
```

Example - 2

```
Input:
N = 6
inorder[] = {3 1 4 0 5 2}
preorder[] = {0 1 3 4 2 5}
Output: 3 4 1 5 2 0
Explanation: The tree will look like
     0
   /   \
  1     2
 /  \  /
3   4 5
```

Approach

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, the leftmost element is the root of the tree. So we know 'A' is the root for given sequences. By searching 'A' in Inorder sequence, we can find out all elements on the left side of 'A' are in the left subtree and elements on the right are in the right subtree. So we know the structure below now.

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

We recursively follow the above steps and get the following tree.

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

Algorithm: buildTree()
1) Pick an element from Preorder. Increment a Preorder Index Variable (preIndex in below code) to pick the next element in the next recursive call.

2) Create a new tree node tNode with the data as a 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.

Code

```
 Node* preorder(int in[], int pre[], int n, int k, int j)
{
   if(n>k)
   {
      int i;
      for(i=k; in[i]!=pre[j] && i<n; i++);
      Node* p=new Node(pre[j]);
      p->left=preorder(in, pre, i, k, j+1);
      p->right=preorder(in, pre, n, i+1, j+i-k+1);
      return p;
   }
   else
      return NULL;
}
Node* buildTree(int in[],int pre[], int n)
{
    return preorder(in,pre,n,0,0);
}
```

Time Complexity: O(n^2). Worst case occurs when the tree is left skewed.

Print Postorder traversal from given Inorder and Preorder traversals

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.

```
int search(int in[] , int x , int n)
{
  for(int i = 0 ;i<n;i++)
  {
    if(x == in[i])
      return i;
  }
  return -1;
}
```

```
void printPostOrder(int in[], int pre[], int n)
{
  int root = search(in , pre[0] , n);

  if(root!= 0)
  printPostOrder(in , pre + 1 , root);

  if(root!=n-1)
  printPostOrder(in + root + 1 , pre + root + 1 , n - root - 1);

  cout << pre[0]<< " ";
//Your code here
}
```

The above solution can be optimized using hashing. We use a HashMap to store elements and their indexes so that we can quickly find index of an element.

```
int preIndex = 0;
void printPost(int in[], int pre[], int inStrt,
               int inEnd, map<int, int> hm)
{
  if (inStrt > inEnd)
      return;

  // Find index of next item in preorder traversal in
  // inorder.
  int inIndex = hm[pre[preIndex++]];

```

```
   // traverse left tree
   printPost(in, pre, inStrt, inIndex - 1, hm);

   // traverse right tree
   printPost(in, pre, inIndex + 1, inEnd, hm);

   // print root node at the end of traversal
   cout << in[inIndex] << " ";
}

void printPostMain(int in[], int pre[],int n)
{
   map<int,int> hm ;
   for (int i = 0; i < n; i++)
   hm[in[i]] = i;

   printPost(in, pre, 0, n - 1, hm);
}
```