17.Tree Traversals

Depth First Traversals:

(a) Inorder

(b) Preorder

(c) Postorder

Breadth First or Level Order Traversal.

Inorder Traversal:

```
Algorithm Inorder(tree)

   1. Traverse the left subtree, i.e., call Inorder(left-subtree)
   2. Visit the root.
   3. Traverse the right subtree, i.e., call Inorder(right-subtree)
```

Uses of Inorder

In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder itraversal s reversed, can be used.

Example: Inorder traversal for the above given figure is 4 2 5 1 3.

Preorder Traversal:

```
Algorithm Preorder(tree)
   1. Visit the root.
   2. Traverse the left subtree, i.e., call Preorder(left-subtree)
   3. Traverse the right subtree, i.e., call Preorder(right-subtree)
```

Uses of Preorder

Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree.

Example: Preorder traversal for the above given figure is 1 2 4 5 3.

Postorder Traversal:

```
Algorithm Postorder(tree)
   1. Traverse the left subtree, i.e., call Postorder(left-subtree)
   2. Traverse the right subtree, i.e., call Postorder(right-subtree)
   3. Visit the root.
```

Uses of Postorder

Postorder traversal is used to delete the tree. Postorder traversal is also useful to get the postfix expression of an expression tree.

Example: Postorder traversal for the above given figure is 4 5 2 3 1

18.Lowest Common Ancestor in a Binary Search Tree.

Given the values of two nodes in a *binary search tree*, write a c program to find the lowest common ancestor. You may assume that both values already exist in the tree.

The function prototype is as follows:

```
 int FindLowestCommonAncestor(node* root, int value1, int value)
```

```
                    20
                    /\
                   8  22
                  /\
                 4 12
                   /\
                  10  14
```

```
I/P : 4 and 14

O/P : 8
(Here the common ancestors of 4 and 14, are {8,20}. Of {8,20}, the
lowest one is 8).
```

Here is the solution

Algorithm:

The main idea of the solution is — While traversing Binary Search Tree from top to bottom, the first node n we encounter with value between n1 and n2, i.e., n1 < n < n2 is the Lowest or Least Common Ancestor(LCA) of n1 and n2 (where n1 < n2). So just traverse the BST in pre-order, if you find a node with value in between n1 and n2 then n is the LCA, if it's value is greater than both n1 and n2 then our LCA lies on left side of the node, if it's value is smaller than both n1 and n2 then LCA lies on right side.

Implementation:

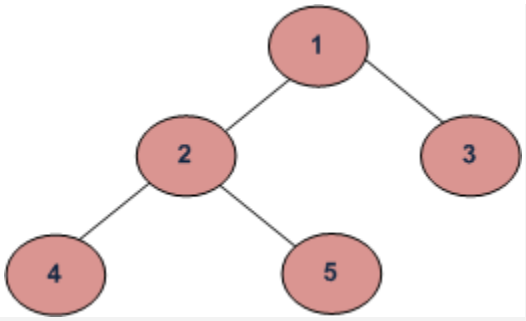
```
/* Function to find least comman ancestor of n1 and n2 */
int leastCommanAncestor(struct node* root, int n1, int n2)
{
  /* If we have reached a leaf node then LCA doesn't exist
     If root->data is equal to any of the inputs then input is
     not valid. For example 20, 22 in the given figure */
  if(root == NULL || root->data == n1 || root->data == n2)
    return -1;

  /* If any of the input nodes is child of the current node
     we have reached the LCA. For example, in the above figure
     if we want to calculate LCA of 12 and 14, recursion should
     terminate when we reach 8*/
  if((root->right != NULL) &&
    (root->right->data == n1 || root->right->data == n2))
    return root->data;
  if((root->left != NULL) &&
    (root->left->data == n1 || root->left->data == n2))
    return root->data;

  if(root->data > n1 && root->data < n2)
    return root->data;
  if(root->data > n1 && root->data > n2)
    return leastCommanAncestor(root->left, n1, n2);
  if(root->data < n1 && root->data < n2)
    return leastCommanAncestor(root->right, n1, n2);
}

}
```

Note that above function assumes that n1 is smaller than n2.

Time complexity: Time complexity is O(Logn) for a balanced BST and O(n) for a skewed BST.

19.Level Order Tree Traversal

Level order traversal of a tree is breadth first traversal for the tree.



Example Tree

Level order traversal of the above tree is 1 2 3 4 5

METHOD 1 (Use function to print a given level)

```
/* Function to print level order traversal a tree*/
void printLevelOrder(struct node* root)
{
  int h = height(root);
  int i;
  for(i=1; i<=h; i++)
    printGivenLevel(root, i);
}

/* Print nodes at a given level */
void printGivenLevel(struct node* root, int level)
{
  if(root == NULL)
    return;
  if(level == 1)
    printf("%d ", root->data);
  else if (level > 1)
  {
    printGivenLevel(root->left, level-1);
    printGivenLevel(root->right, level-1);
  }
}
```

Time Complexity: O(n^2) in worst case. For a skewed tree, printGivenLevel() takes O(n) time where n is the number of nodes in the skewed tree. So time complexity of printLevelOrder() is O(n) + O(n-1) + O(n-2) + .. + O(1) which is O(n^2).

METHOD 2 (Use Queue)

Algorithm:

For each node, first the node is visited and then it's child nodes are put in a FIFO queue.

```
printLevelorder(tree)
1) Create an empty queue q
2) temp_node = root /*start from root*/
3) Loop while temp_node is not NULL
    a) print temp_node->data.
    b) Enqueue temp_node's children (first left then right children)
to q
    c) Dequeue a node from q and assign it's value to temp_node
```

Implementation:

Here is a simple implementation of the above algorithm. Queue is implemented using an array with maximum size of 500. We can implement queue as linked list also.

```
/* frunction prototypes */
struct node** createQueue(int *, int *);
void enQueue(struct node **, int *, struct node *);
struct node *deQueue(struct node **, int *);

/* Given a binary tree, print its nodes in level order
   using array for implementing queue */
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);
```

```
  }
}

/*UTILITY FUNCTIONS*/
struct node** createQueue(int *front, int *rear)
{
  struct node **queue =
   (struct node **)malloc(sizeof(struct node*)*MAX_Q_SIZE);

  *front = *rear = 0;
  return queue;
}

void enQueue(struct node **queue, int *rear, struct node *new_node)
{
  queue[*rear] = new_node;
  (*rear)++;
}

struct node *deQueue(struct node **queue, int *front)
{
  (*front)++;
  return queue[*front - 1];
}
```

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

20.Program to count leaf nodes in a binary tree

A node is a leaf node if both left and right child nodes of it are NULL.

Here is an algorithm to get the leaf node count.

```
getLeafCount(node)
1) If node is NULL then return 0.
2) Else If left and right child nodes are NULL return 1.
3) Else recursively calculate leaf count of the tree using
below formula.
    Leaf count of a tree = Leaf count of left subtree +
                                Leaf count of right
subtree
```

Implementation:

```
/* Function to get the count of leaf nodes in a binary tree*/
unsigned int getLeafCount(struct node* node)
{
  if(node == NULL)
    return 0;
  if(node->left == NULL && node->right==NULL)
    return 1;
  else
    return getLeafCount(node->left)+
           getLeafCount(node->right);
}
```

21.Level order traversal in spiral form

Write a function to print spiral order traversal of a tree.

Algorithm:

This problem is an extension of the level order traversal post.
To print the nodes in spiral order, nodes at different levels should be printed in alternating order. An additional Boolean variable *ltr* is used to change printing order of levels. If *ltr* is 1 then printGivenLevel() prints nodes from left to right else from right to left. Value of *ltr* is flipped in each iteration to change the order.

Function to print level order traversal of tree

```
printLevelorder(tree)
  bool ltr = 0;
  for d = 1 to height(tree)
     printGivenLevel(tree, d, ltr);
     ltr ~= ltr /*flip ltr*/
```

Function to print all nodes at a given level

```
printGivenLevel(tree, level)
if tree is NULL then return;
if level is 1, then
```

```
    print(tree->data);
else if level greater than 1, then
    if(rtl)
        printGivenLevel(tree->right, level-1, ltr);
        printGivenLevel(tree->left, level-1, ltr);
    else
        printGivenLevel(tree->left, level-1, ltr);
        printGivenLevel(tree->right, level-1, ltr);
```

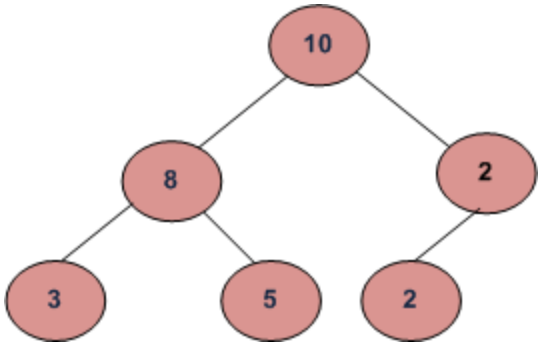
Implementation:

```
/* Function to print spiral traversal of a tree*/
void printLevelOrder(struct node* root)
{
  int h = height(root);
  int i;

  /*ltr -> Left to Right. If this variable is set,
    then the given level is traverseed from left to right. */
  bool ltr = 0;
  for(i=1; i<=h; i++)
  {
    printGivenLevel(root, i, ltr);

    /*Revert ltr to traverse next level in oppposite order*/
    ltr = ~ltr;
  }
}

/* Print nodes at a given level */
void printGivenLevel(struct node* root, int level, int ltr)
{
  if(root == NULL)
    return;
  if(level == 1)
    printf("%d ", root->data);
  else if (level > 1)
  {
    if(ltr)
    {
      printGivenLevel(root->left, level-1, ltr);
      printGivenLevel(root->right, level-1, ltr);
    }
    else
    {
      printGivenLevel(root->right, level-1, ltr);
      printGivenLevel(root->left, level-1, ltr);
    }
  }
}
```

22.Check for Children Sum Property in a Binary Tree.

Given a binary tree, write a function that returns true if the tree satisfies below property.

For every node, data value must be equal to sum of data values in left and right children. Consider data value as 0 for NULL children. Below tree is an example



Algorithm:

Traverse the given binary tree. For each node check (recursively) if the node and both its children satisfy the Children Sum Property, if so then return true else return false.

Implementation:

```
/* returns 1 if children sum property holds for the given
    node and both of its children*/
int isSumProperty(struct node* node)
{
  /* left_data is left child data and right_data is for right child data*/
  int left_data = 0,  right_data = 0;

  /* If node is NULL or it's a leaf node then
     return true */
  if(node == NULL ||
     (node->left == NULL && node->right == NULL))
    return 1;
  else
  {
    /* If left child is not present then 0 is used
       as data of left child */
    if(node->left != NULL)
      left_data = node->left->data;

    /* If right child is not present then 0 is used
      as data of right child */
    if(node->right != NULL)
      right_data = node->right->data;
```

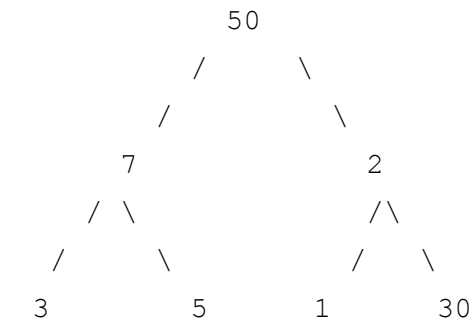
```
    /* if the node and both of its children satisfy the
       property return 1 else 0*/
    if((node->data == left_data + right_data)&&
        isSumProperty(node->left) &&
        isSumProperty(node->right))
      return 1;
    else
      return 0;
  }
}
```

Time Complexity: O(n), we are doing a complete traversal of the tree.

23.Convert an arbitrary Binary Tree to a tree that holds Children Sum Property

Question: Given an arbitrary binary tree, convert it to a binary tree that holds Children Sum Property. You can only increment data values in any node.

For example, the below tree doesn't hold the children sum property, convert it to a tree that holds the property.



Algorithm:

Traverse given tree in post order to convert it, i.e., first change left and right children to hold the children sum property then change the parent node.

Let difference between node's data and children sum be diff.

```
diff = node's children sum - node's data
```

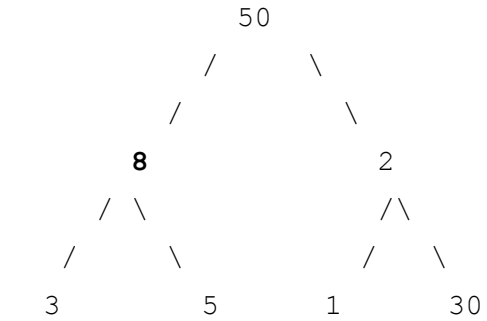
If diff is 0 then nothing needs to be done.

If diff > 0 (node's data is smaller than node's children sum) increment the node's data by diff.

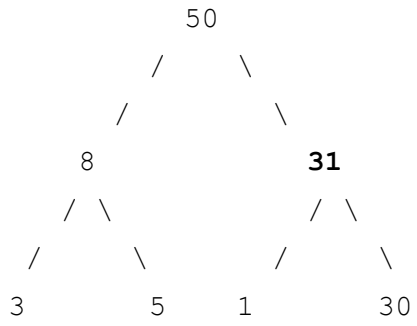
If diff < 0 (node's data is smaller than the node's children sum) then increment one child's data. We can choose to increment either left or right child. Let us always increment the left child. Incrementing a child changes the subtree's children sum property so we need to change left subtree also. Left subtree is fixed by incrementing all the children in left subtree by diff, we have a function increment() for this purpose (see below C code).

Let us run the algorithm for the given example.

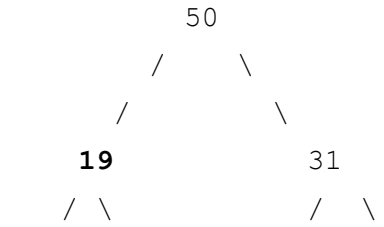
First convert the left subtree (increment 7 to 8).



Then convert the right subtree (increment 2 to 31)



Now convert the root, we have to increment left subtree for converting the root.



```
 /      \        /       \
14       5      1        30
```

Please note the last step – we have incremented 8 to 19, and to fix the subtree we have incremented 3 to 14.

Implementation:

```
/* Program to convert an aribitary binary tree to
   a tree that holds children sum property */

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

struct node
{
  int data;
  struct node* left;
  struct node* right;
};

/* This function is used to increment left subtree */
void increment(struct node* node, int diff);

/* Helper function that allocates a new node
 with the given data and NULL left and right
 pointers. */
struct node* newNode(int data);

/* This function changes a tree to to hold children sum
   property */
void convertTree(struct node* node)
{
  int left_data = 0,  right_data = 0, diff;

  /* If tree is empty or it's a leaf node then
     return true */
  if(node == NULL ||
     (node->left == NULL && node->right == NULL))
    return;
  else
  {
    /* convert left and right subtrees  */
    convertTree(node->left);
    convertTree(node->right);

    /* If left child is not present then 0 is used
       as data of left child */
    if(node->left != NULL)
      left_data = node->left->data;
```

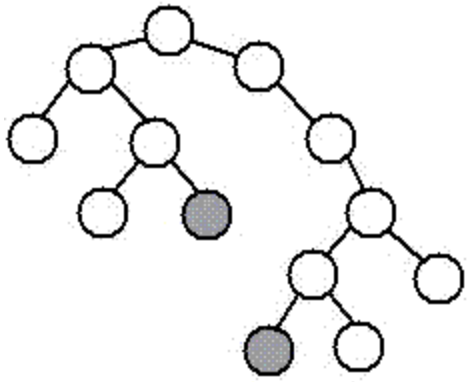
```
    /* If right child is not present then 0 is used
      as data of right child */
    if(node->right != NULL)
      right_data = node->right->data;

    /* get the diff of node's data and children sum */
    diff = left_data + right_data - node->data;

    /* If node's data is smaller than children sum,
       then increment node's data by diff */
    if(diff > 0)
       node->data = node->data + diff;

    /* THIS IS TRICKY --> If node's data is greater than children sum,
      then increment left subtree by diff */
    if(diff < 0)
      increment(node, -diff);
  }
}

/* This function is used to increment left subtree */
void increment(struct node* node, int diff)
{
  /* This if is for the case where left child is NULL */
  if(node->left == NULL)
  {
    node->left = newNode(diff);
    return;
  }

  /* Go in depth, and fix all left children */
  while(node->left != NULL)
  {
    node->left->data = node->left->data + diff;
    node = node->left;
  }
}

/* Given a binary tree, printInorder() prints out its
   inorder traversal*/
void printInorder(struct node* node)
{
  if (node == NULL)
    return;

  /* first recur on left child */
  printInorder(node->left);

  /* then print the data of node */
  printf("%d ", node->data);

  /* now recur on right child */
  printInorder(node->right);
}
```

```
/* Helper function that allocates a new node
 with the given data and NULL left and right
 pointers. */
struct node* newNode(int data)
{
  struct node* node =
      (struct node*)malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;
  return(node);
}

/* Driver program to test above functions */
int main()
{
  struct node *root = newNode(50);
  root->left        = newNode(7);
  root->right       = newNode(2);
  root->left->left  = newNode(3);
  root->left->right = newNode(5);
  root->right->left  = newNode(1);
  root->right->right = newNode(30);

  printf("\n Inorder traversal before conversion ");
  printInorder(root);

  convertTree(root);

  printf("\n Inorder traversal after conversion ");
  printInorder(root);

  getchar();
  return 0;
}
```

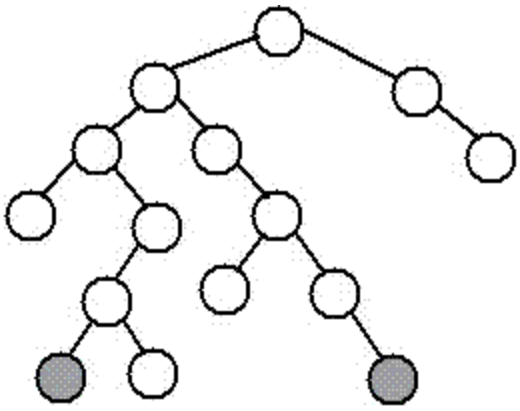
Time Complexity: O(n^2), Worst case complexity is for a skewed tree such that nodes are in decreasing order from root to leaf.

24.Diameter of a Binary 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).



diameter, 9 nodes, through root



diameter, 9 nodes, NOT through root

The diameter of a tree T is the largest of the following quantities:

* the diameter of T's left subtree

* the diameter of T's right subtree

* the longest path between leaves that goes through the root of T (this can be computed from the heights of the subtrees of T)

Implementation:

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

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* function to create a new node of tree and returns pointer */
struct node* newNode(int data);

/* returns max of two integers */
int max(int a, int b);

/* Function to get diameter of a binary tree */
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 = height(tree->left);
  int rheight = height(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));
}

/* UTILITY FUNCTIONS TO TEST diameter() FUNCTION */

/*  The function Compute the "height" of a tree. Height is the
    number f nodes along the longest path from the root node
    down to the farthest leaf node.*/
int height(struct node* node)
{
   /* base case tree is empty */
   if(node == NULL)
       return 0;

   /* If tree is not empty then height = 1 + max of left
      height and right heights */
   return 1 + max(height(node->left), height(node->right));
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                      malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* returns maximum of two integers */
int max(int a, int b)
{
  return (a >= b)? a: b;
}

/* Driver program to test above functions*/
int main()
{

  /* Constructed binary tree is
            1
```

```
          /   \
        2      3
      /  \
    4     5
  */
  struct node *root = newNode(1);
  root->left        = newNode(2);
  root->right       = newNode(3);
  root->left->left  = newNode(4);
  root->left->right = newNode(5);

  printf("Diameter of the given binary tree is %d\n", diameter(root));

  getchar();
  return 0;
}
```

Time Complexity: O(n^2)

Optimized implementation: The above implementation can be optimized by calculating the height in the same recursion rather than calling a height() separately. This optimization reduces time complexity to O(n).

```
/*The second parameter is to store the height of tree.
   Initially, we need to pass a pointer to a location with value
   as 0. So, function should be used as follows:

   int height = 0;
   struct node *root = SomeFunctionToMakeTree();
   int diameter = diameterOpt(root, &height); */
int diameterOpt(struct node *root, int* height)
{
  /* lh --> Height of left subtree
      rh --> Height of right subtree */
  int lh = 0, rh = 0;

  /* ldiameter  --> diameter of left subtree
      rdiameter  --> Diameter of right subtree */
  int ldiameter = 0, rdiameter = 0;

  if(root == NULL)
  {
    *height = 0;
     return 0; /* diameter is also 0 */
  }

  /* Get the heights of left and right subtrees in lh and rh
    And store the returned values in ldiameter and ldiameter */
  ldiameter = diameterOpt(root->left, &lh);
  rdiameter = diameterOpt(root->right, &rh);

  /* Height of current node is max of heights of left and
```

```
     right subtrees plus 1*/
  *height = max(lh, rh) + 1;

  return max(lh + rh + 1, max(ldiameter, rdiameter));
}
```

Time Complexity: O(n)

25.How to determine if a binary tree is height-balanced?

Consider a height-balancing scheme where following conditions should be checked to determine if a binary tree is balanced.

An empty tree is height-balanced. A non-empty binary tree T is balanced if:

1) Left subtree of T is balanced

2) Right subtree of T is balanced

3) The difference between heights of left subtree and right subtree is not more than 1.

The above height-balancing scheme is used in AVL trees

```
/* Returns true if binary tree with root as root is height-balanced */
bool isBalanced(struct node *root)
{
   int lh; /* for height of left subtree */
   int rh; /* for height of right subtree */

   /* If tree is empty then return true */
   if(root == NULL)
    return 1;

   /* Get the height of left and right sub trees */
   lh = height(root->left);
   rh = height(root->right);

   if( abs(lh-rh) <= 1 &&
       isBalanced(root->left) &&
       isBalanced(root->right))
     return 1;

   /* If we reach here then tree is not height-balanced */
   return 0;
}
```

Time Complexity: O(n^2) Worst case occurs in case of skewed tree.

Optimized implementation: Above implementation can be optimized by calculating the height in the same recursion rather than calling a height() function separately.. This optimization reduces time complexity to O(n).

```
/* The function returns true if root is balanced else false
   The second parameter is to store the height of tree.
   Initially, we need to pass a pointer to a location with value
   as 0. We can also write a wrapper over this function */
bool isBalanced(struct node *root, int* height)
{
  /* lh --> Height of left subtree
     rh --> Height of right subtree */
  int lh = 0, rh = 0;

  /* l will be true if left subtree is balanced
    and r will be true if right subtree is balanced */
  int l = 0, r = 0;

  if(root == NULL)
  {
    *height = 0;
     return 1;
  }

  /* Get the heights of left and right subtrees in lh and rh
    And store the returned values in l and r */
  l = isBalanced(root->left, &lh);
  r = isBalanced(root->right,&rh);

  /* Height of current node is max of heights of left and
     right subtrees plus 1*/
  *height = (lh > rh? lh: rh) + 1;

  /* If difference between heights of left and right
     subtrees is more than 2 then this node is not balanced
     so return 0 */
  if((lh - rh >= 2) || (rh - lh >= 2))
    return 0;

  /* If this node is balanced and left and right subtrees
    are balanced then return true */
  else return l&&r;
}
```

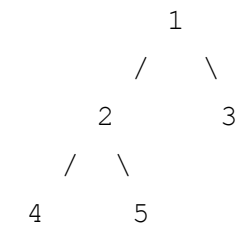
Time Complexity: O(n)

26.Inorder Tree Traversal without Recursion

Using Stack is the obvious way to traverse tree without recursion. Below is an algorithm for traversing binary tree using stack. See this for step wise step execution of the algorithm.

```
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 = current->right
     c) Go to step 3.
5) If current is NULL and stack is empty then we are done.
```

Let us consider the below tree for example




```
Step 1 Creates an empty stack: S = NULL

Step 2 sets current as address of root: current -> 1

Step 3 Pushes the current node and set current = current->left until
current is NULL
     current -> 1
     push 1: Stack S -> 1
     current -> 2
     push 2: Stack S -> 2, 1
     current -> 4
     push 4: Stack S -> 4, 2, 1
     current = NULL
```

Step 4 pops from S

a) Pop 4: Stack S -> 2, 1

b) print "4"

c) current = NULL /*right of 4 */ and go to step 3

Since current is NULL step 3 doesn't do anything.

Step 4 pops again.

a) Pop 2: Stack S -> 1

b) print "2"

c) current -> 5/*right of 2 */ and go to step 3

Step 3 pushes 5 to stack and makes current NULL

Stack S -> 5, 1

current = NULL

Step 4 pops from S

a) Pop 5: Stack S -> 1

b) print "5"

c) current = NULL /*right of 5 */ and go to step 3

Since current is NULL step 3 doesn't do anything

Step 4 pops again.

a) Pop 1: Stack S -> NULL

b) print "1"

c) current -> 3 /*right of 5 */

Step 3 pushes 3 to stack and makes current NULL

Stack S -> 3

current = NULL

Step 4 pops from S

a) Pop 3: Stack S -> NULL

b) print "3"

c) current = NULL /*right of 3 */

Traversal is done now as stack S is empty and current is NULL.

Implementation:

```
/* A binary tree tNode has data, pointer to left child
   and a pointer to right child */
struct tNode
{
   int data;
   struct tNode* left;
   struct tNode* right;
};

/* Structure of a stack node. Linked List implementation is used for
   stack. A stack node contains a pointer to tree node and a pointer to
   next stack node */
struct sNode
{
  struct tNode *t;
  struct sNode *next;
};

/* Stack related functions */
void push(struct sNode** top_ref, struct tNode *t);
struct tNode *pop(struct sNode** top_ref);
bool isEmpty(struct sNode *top);

/* Iterative function for inorder tree traversal */
void inOrder(struct tNode *root)
{
  /* set current to root of binary tree */
  struct tNode *current = root;
  struct sNode *s = NULL;  /* Initialize stack s */
  bool done = 0;

  while (!done)
  {
    /* Reach the left most tNode of the current tNode */
    if(current !=  NULL)
    {
      /* place pointer to a tree node on the stack before traversing
        the node's left subtree */
      push(&s, current);
      current = current->left;
    }

    /* backtrack from the empty subtree and visit the tNode
       at the top of the stack; however, if the stack is empty,
      you are done */
    else
    {
      if (!isEmpty(s))
      {
        current = pop(&s);
```

```
        printf("%d ", current->data);

        /* we have visited the node and its left subtree.
          Now, it's right subtree's turn */
        current = current->right;
      }
      else
        done = 1;
    }
  } /* end of while */
}

/* UTILITY FUNCTIONS */
/* Function to push an item to sNode*/
void push(struct sNode** top_ref, struct tNode *t)
{
  /* allocate tNode */
  struct sNode* new_tNode =
            (struct sNode*) malloc(sizeof(struct sNode));

  if(new_tNode == NULL)
  {
     printf("Stack Overflow \n");
     getchar();
     exit(0);
  }

  /* put in the data  */
  new_tNode->t  = t;

  /* link the old list off the new tNode */
  new_tNode->next = (*top_ref);

  /* move the head to point to the new tNode */
  (*top_ref)    = new_tNode;
}

/* The function returns true if stack is empty, otherwise false */
bool isEmpty(struct sNode *top)
{
   return (top == NULL)? 1 : 0;
}

/* Function to pop an item from stack*/
struct tNode *pop(struct sNode** top_ref)
{
  struct tNode *res;
  struct sNode *top;

  /*If sNode is empty then error */
  if(isEmpty(*top_ref))
  {
     printf("Stack Underflow \n");
     getchar();
     exit(0);
```

```
  }
  else
  {
     top = *top_ref;
     res = top->t;
     *top_ref = top->next;
     free(top);
     return res;
  }
}
```

Time Complexity: O(n)

27.Inorder Tree Traversal without recursion and without stack!

Using Morris Traversal, we can traverse the tree without using stack and recursion. The idea of Morris Traversal is based on Threaded Binary Tree. In this traversal, we first create links to Inorder successor and print the data using these links, and finally revert the changes to restore original tree.

```
1. Initialize current as root
2. While current is not NULL
   If current does not have left child
      a) Print current's data
      b) Go to the right, i.e., current = current->right
   Else
      a) Make current as right child of the rightmost node in
current's left subtree
      b) Go to this left child, i.e., current = current->left
```

Although the tree is modified through the traversal, it is reverted back to its original shape after the completion. Unlike Stack based traversal, no extra space is required for this traversal.

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

/* A binary tree tNode has data, pointer to left child
   and a pointer to right child */
struct tNode
{
   int data;
   struct tNode* left;
   struct tNode* right;
};

/* Function to traverse binary tree without recursion and
```

```
   without stack */
void MorrisTraversal(struct tNode *root)
{
  struct tNode *current,*pre;

  if(root == NULL)
     return;

  current = root;
  while(current != NULL)
  {
    if(current->left == NULL)
    {
      printf(" %d ", current->data);
      current = current->right;
    }
    else
    {
      /* Find the inorder predecessor of current */
      pre = current->left;
      while(pre->right != NULL && pre->right != current)
        pre = pre->right;

      /* Make current as right child of its inorder predecessor */
      if(pre->right == NULL)
      {
        pre->right = current;
        current = current->left;
      }

      /* Revert the changes made in if part to restore the original
        tree i.e., fix the right child of predecssor */
      else
      {
        pre->right = NULL;
        printf(" %d ",current->data);
        current = current->right;
      } /* End of if condition pre->right == NULL */
    } /* End of if condition current->left == NULL*/
  } /* End of while */
}

/* UTILITY FUNCTIONS */
/* Helper function that allocates a new tNode with the
   given data and NULL left and right pointers. */
struct tNode* newtNode(int data)
{
  struct tNode* tNode = (struct tNode*)
                        malloc(sizeof(struct tNode));
  tNode->data = data;
  tNode->left = NULL;
  tNode->right = NULL;

  return(tNode);
}
```

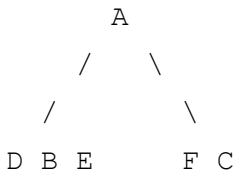
```
/* Driver program to test above functions*/
int main()
{

  /* Constructed binary tree is
            1
          /   \
        2      3
      /  \
    4     5
  */
  struct tNode *root = newtNode(1);
  root->left        = newtNode(2);
  root->right       = newtNode(3);
  root->left->left  = newtNode(4);
  root->left->right = newtNode(5);

  MorrisTraversal(root);

  getchar();
  return 0;
}
```

28.Construct Tree from given Inorder and Preorder traversals

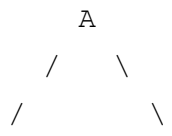
Let us consider the below traversals:

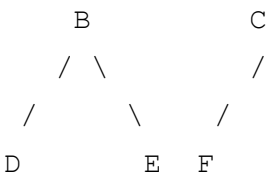
Inorder sequence: D B E A F C
Preorder sequence: A B D E C F

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



We recursively follow above steps and get the following tree.





Algorithm: buildTree()

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

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

```
/* program to construct tree using inorder and preorder traversals */
#include<stdio.h>
#include<stdlib.h>

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
  char data;
  struct node* left;
  struct node* right;
};

/* Prototypes for utility functions */
int search(char arr[], int strt, int end, char value);
struct node* newNode(char data);

/* Recursive function to construct binary of size len from
   Inorder traversal in[] and Preorder traversal pre[].  Initial values
   of inStrt and inEnd should be 0 and len -1.  The function doesn't
   do any error checking for cases where inorder and preorder
   do not form a tree */
struct node* buildTree(char in[], char pre[], int inStrt, int inEnd)
{
  static int preIndex = 0;

  if(inStrt > inEnd)
     return NULL;

  /* Pick current node from Preorder traversal using preIndex
    and increment preIndex */
  struct node *tNode = newNode(pre[preIndex++]);

  /* If this node has no children then return */
```

```
  if(inStrt == inEnd)
    return tNode;

  /* Else find the index of this node in Inorder traversal */
  int inIndex = search(in, inStrt, inEnd, tNode->data);

  /* Using index in Inorder traversal, construct left and
     right subtress */
  tNode->left = buildTree(in, pre, inStrt, inIndex-1);
  tNode->right = buildTree(in, pre, inIndex+1, inEnd);

  return tNode;
}

/* UTILITY FUNCTIONS */
/* Function to find index of value in arr[start...end]
   The function assumes that value is present in in[] */
int search(char arr[], int strt, int end, char value)
{
  int i;
  for(i = strt; i <= end; i++)
  {
    if(arr[i] == value)
      return i;
  }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(char data)
{
  struct node* node = (struct node*)malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* This funtcion is here just to test buildTree() */
void printInorder(struct node* node)
{
  if (node == NULL)
     return;

  /* first recur on left child */
  printInorder(node->left);

  /* then print the data of node */
  printf("%c ", node->data);

  /* now recur on right child */
  printInorder(node->right);
}
```

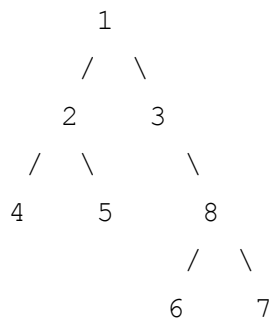
```
/* Driver program to test above functions */
int main()
{
  char in[] = {'D', 'B', 'E', 'A', 'F', 'C'};
  char pre[] = {'A', 'B', 'D', 'E', 'C', 'F'};
  int len = sizeof(in)/sizeof(in[0]);
  struct node *root = buildTree(in, pre, 0, len - 1);

  /* Let us test the built tree by printing Insorder traversal */
  printf("\n Inorder traversal of the constructed tree is \n");
  printInorder(root);
  getchar();
}
```

Time Complexity: O(n^2). Worst case occurs when tree is left skewed. Example Preorder and Inorder traversals for worst case are {A, B, C, D} and {D, C, B, A}.

29.Maximum width of a binary tree

Given a binary tree, write a function to get the maximum width of the given tree. Width of a tree is maximum of widths of all levels.

Let us consider the below example tree.



For the above tree,
width of level 1 is 1,
width of level 2 is 2,
width of level 3 is 3
width of level 4 is 2.

So the maximum width of the tree is 3.

Algortihm:

There are basically two functions. One is to count nodes at a given level (getWidth), and other is to get the maximum width of the tree(getMaxWidth). getMaxWidth() makes use of getWidth() to get the width of all levels starting from root.

```
/*Function to print level order traversal of tree*/
getMaxWidth(tree)
maxWdth = 0
for i = 1 to height(tree)
  width =   getWidth(tree, i);
  if(width > maxWdth)
      maxWdth  = width
return width
/*Function to get width of a given level */
getWidth(tree, level)
if tree is NULL then return 0;
if level is 1, then return 1;
else if level greater than 1, then
    return getWidth(tree->left, level-1) +
    getWidth(tree->right, level-1);
```

Implementation:

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

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/*Function protoypes*/
int getWidth(struct node* root, int level);
int height(struct node* node);
struct node* newNode(int data);
```

```
/* Function to get the maximum width of a binary tree*/
int getMaxWidth(struct node* root)
{
  int maxWidth = 0;
  int width;
  int h = height(root);
  int i;

  /* Get width of each level and compare
     the width with maximum width so far */
  for(i=1; i<=h; i++)
  {
    width = getWidth(root, i);
    if(width > maxWidth)
      maxWidth = width;
  }

  return maxWidth;
}

/* Get width of a given level */
int getWidth(struct node* root, int level)
{

  if(root == NULL)
    return 0;

  if(level == 1)
    return 1;

  else if (level > 1)
    return getWidth(root->left, level-1) +
              getWidth(root->right, 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);
   }
}
/* Helper function that allocates a new node with the
```

```
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                       malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;
  return(node);
}
/* Driver program to test above functions*/
int main()
{
  struct node *root = newNode(1);
  root->left        = newNode(2);
  root->right       = newNode(3);
  root->left->left  = newNode(4);
  root->left->right = newNode(5);
  root->right->right = newNode(8);
  root->right->right->left  = newNode(6);
  root->right->right->right  = newNode(7);

  /*
   Constructed bunary tree is:
          1
        /  \
       2    3
     /  \     \
    4   5     8
              /  \
             6   7
  */
  printf("Maximum width is %d \n", getMaxWidth(root));
  getchar();
  return 0;
}
```

30.Foldable Binary Trees

Question: Given a binary tree, find out if the tree can be folded or not.

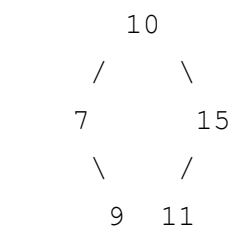
A tree can be folded if left and right subtrees of the tree are structure wise mirror image of each other. An empty tree is considered as foldable.

Consider the below trees:

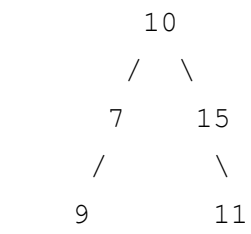
(a) and (b) can be folded.

(c) and (d) cannot be folded.

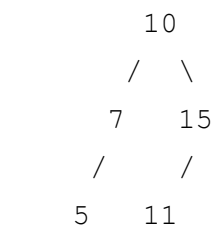
(a)



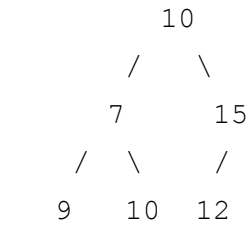
(b)



(c)



(d)



Method 1 (Change Left subtree to its Mirror and compare it with Right subtree)

Algorithm: isFoldable(root)

```
1) If tree is empty, then return true.
2) Convert the left subtree to its mirror image
    mirror(root->left); /* See this post */
3) Check if the structure of left subtree and right subtree is same
```

and store the result.

```
res = isStructSame(root->left, root->right); /*isStructSame()
    recursively compares structures of two subtrees and returns
    true if structures are same */
```

4) Revert the changes made in step (2) to get the original tree.

```
mirror(root->left);
```

5) Return result res stored in step 2.

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

/* You would want to remove below 3 lines if your compiler
   supports bool, true and false */
#define bool int
#define true 1
#define false 0

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
  int data;
  struct node* left;
  struct node* right;
};

/* converts a tree to its mrror image */
void mirror(struct node* node);

/* returns true if structure of two trees a and b is same
   Only structure is considered for comparison, not data! */
bool isStructSame(struct node *a, struct node *b);

/* Returns true if the given tree is foldable */
bool isFoldable(struct node *root)
{
  bool res;

  /* base case */
  if(root == NULL)
    return true;

  /* convert left subtree to its mirror */
  mirror(root->left);

  /* Compare the structures of the right subtree and mirrored
    left subtree */
  res = isStructSame(root->left, root->right);

  /* Get the originial tree back */
```

```
  mirror(root->left);

  return res;
}

bool isStructSame(struct node *a, struct node *b)
{
  if (a == NULL && b == NULL)
  {  return true; }
  if ( a != NULL && b != NULL &&
       isStructSame(a->left, b->left) &&
       isStructSame(a->right, b->right)
     )
  {  return true; }

  return false;
}

/* UTILITY FUNCTIONS */
/* Change a tree so that the roles of the  left and
    right pointers are swapped at every node.
    See http://geeksforgeeks.org/?p=662 for details */
void mirror(struct node* node)
{
  if (node==NULL)
    return;
  else
  {
    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;
  }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                      malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* Driver program to test mirror() */
```

```
int main(void)
{
  /* The constructed binary tree is
         1
       /   \
      2     3
      \    /
       4  5
  */
  struct node *root = newNode(1);
  root->left        = newNode(2);
  root->right       = newNode(3);
  root->right->left  = newNode(4);
  root->left->right = newNode(5);

  if(isFoldable(root) == 1)
  { printf("\n tree is foldable"); }
  else
  { printf("\n tree is not foldable"); }

  getchar();
  return 0;
}
```

Time complexity: O(n)

Method 2 (Check if Left and Right subtrees are Mirror)

There are mainly two functions:

// Checks if tree can be folded or not

```
IsFoldable(root)
1) If tree is empty then return true
2) Else check if left and right subtrees are structure wise mirrors of
    each other. Use utility function IsFoldableUtil(root->left,
    root->right) for this.
```

// Checks if n1 and n2 are mirror of each other.

```
IsFoldableUtil(n1, n2)
1) If both trees are empty then return true.
2) If one of them is empty and other is not then return false.
3) Return true if following conditions are met
   a) n1->left is mirror of n2->right
   b) n1->right is mirror of n2->left
```

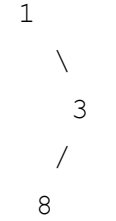
```
#include<stdio.h>
#include<stdlib.h>

/* You would want to remove below 3 lines if your compiler
   supports bool, true and false */
#define bool int
#define true 1
#define false 0

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
  int data;
  struct node* left;
  struct node* right;
};

/* A utility function that checks if trees with roots as n1 and n2
  are mirror of each other */
bool IsFoldableUtil(struct node *n1, struct node *n2);

/* Returns true if the given tree can be folded */
bool IsFoldable(struct node *root)
{
     if (root == NULL)
     {  return true;  }

     return IsFoldableUtil(root->left, root->right);
}

/* A utility function that checks if trees with roots as n1 and n2
  are mirror of each other */
bool IsFoldableUtil(struct node *n1, struct node *n2)
{
    /* If both left and right subtrees are NULL,
      then return true */
    if (n1 == NULL && n2 == NULL)
    {  return true;  }

    /* If one of the trees is NULL and other is not,
      then return false */
    if (n1 == NULL || n2 == NULL)
    {  return false; }

    /* Otherwise check if left and right subtrees are mirrors of
       their counterparts */
    return IsFoldableUtil(n1->left, n2->right) &&
           IsFoldableUtil(n1->right, n2->left);
}

/*UTILITY FUNCTIONS */
/* Helper function that allocates a new node with the
```

```
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                       malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* Driver program to test mirror() */
int main(void)
{
  /* The constructed binary tree is
         1
       /   \
      2     3
      \    /
       4  5
  */
  struct node *root = newNode(1);
  root->left        = newNode(2);
  root->right       = newNode(3);
  root->left->right  = newNode(4);
  root->right->left = newNode(5);

  if(IsFoldable(root) == true)
  { printf("\n tree is foldable"); }
  else
  { printf("\n tree is not foldable"); }

  getchar();
  return 0;
}
```

31.Print nodes at k distance from root

Given a root of a tree, and an integer k. Print all the nodes which are at k distance from root.

For example, in the below tree, 4, 5 & 8 are at distance 2 from root.



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

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
   int data;
   struct node* left;
   struct node* right;
};

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 ) ;
   }
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                       malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* Driver program to test above functions*/
int main()
{

  /* Constructed binary tree is
            1
          /   \
        2      3
      /  \    /
    4     5  8
  */
  struct node *root = newNode(1);
  root->left        = newNode(2);
```

```
  root->right        = newNode(3);
  root->left->left   = newNode(4);
  root->left->right  = newNode(5);
  root->right->left  = newNode(8);

  printKDistant(root, 2);

  getchar();
  return 0;
}
```

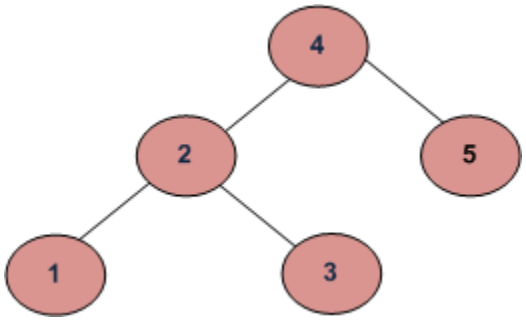
The above program prints 4, 5 and 8.

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

32.Sorted order printing of a given array that represents a BST

Given an array that stores a complete Binary Search Tree, write a function that efficiently prints the given array in ascending order.

For example, given an array [4, 2, 5, 1, 3], the function should print 1, 2, 3, 4, 5



Solution:

Inorder traversal of BST prints it in ascending order. The only trick is to modify recursion termination condition in standard Inorder Tree Traversal.

Implementation:

```
#include<stdio.h>

void printSorted(int arr[], int start, int end)
{
  if(start > end)
    return;
```

```
  // print left subtree
  printSorted(arr, start*2 + 1, end);

  // print root
  printf("%d  ", arr[start]);

  // print right subtree
  printSorted(arr, start*2 + 2, end);
}

int main()
{
  int arr[] = {4, 2, 5, 1, 3};
  int arr_size = sizeof(arr)/sizeof(int);
  printSorted(arr, 0, arr_size-1);
  getchar();
  return 0;
}
```

Time Complexity: O(n)

33.Applications of tree data structure

Difficulty Level: Rookie

Why Tree?

Unlike Array and Linked List, which are linear data structures, tree is hierarchical (or non-linear) data structure.

1) One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

file system

———

```
     /   <-- root
  /      \
...        home
      /          \
   ugrad        course
    /          /    |    \
  ...        cs101 cs112 cs113
```

2) If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays). Self-balancing search trees like AVLand Red-Black trees guarantee an upper bound of O(Logn) for search.

3) We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists). Self-balancing search trees like AVL and Red-Black trees guarantee an upper bound of O(Logn) for insertion/deletion.

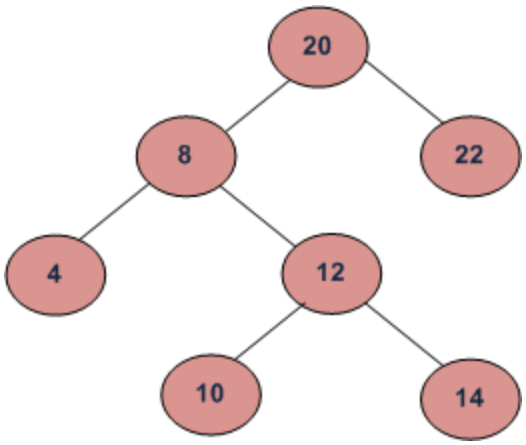
4) Like Linked Lists and unlike Arrays, Pointer implementation of trees don't have an upper limit on number of nodes as nodes are linked using pointers.

As per Wikipedia, following are the common uses of tree.
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

References:
http://www.cs.bu.edu/teaching/c/tree/binary/
http://en.wikipedia.org/wiki/Tree_%28data_structure%29#Common_uses

34.Inorder Successor in Binary Search Tree

In Binary Tree, Inorder successor of a node is the next node in Inorder traversal of the Binary Tree. Inorder Successor is NULL for the last node in Inoorder traversal.
In Binary Search Tree, Inorder Successor of an input node can also be defined as the node with the smallest key greater than the key of input node. So, it is sometimes important to find next node in sorted order.



In the above diagram, inorder successor of **8** is **10**, inorder successor of **10** is **12** and inorder successor of **14** is **20**.

Algorithm to find Inorder Successor

Algorithm is divided into two cases on the basis of right subtree of the input node being empty or not.

Input: *node, root* // *node* is the node whose Inorder successor is needed.

output: *succ* // *succ* is Inorder successor of *node*.

1) If right subtree of *node* is not *NULL*, then *succ* lies in right subtree. Do following.

Go to right subtree and return the node with minimum key value in right subtree.

2) If right sbtree of *node* is NULL, then *succ* is one of the ancestors. Do following.

Travel up using the parent pointer until you see a node which is left child of it's parent. The parent of such a node is the *succ*.

Implementation

Note that the function to find InOrder Successor is highlighted (with gray background) in below code.

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

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
    struct node* parent;
};
```

```
struct node * minValue(struct node* node);

struct node * inOrderSuccessor(struct node *root, struct node *n)
{
  // step 1 of the above algorithm
  if( n->right != NULL )
    return minValue(n->right);

  // step 2 of the above algorithm
  struct node *p = n->parent;
  while(p != NULL && n == p->right)
  {
     n = p;
     p = p->parent;
  }
  return p;
}

/* Given a non-empty binary search tree, return the minimum data
    value found in that tree. Note that the entire tree does not need
    to be searched. */
struct node * minValue(struct node* node) {
  struct node* current = node;

  /* loop down to find the leftmost leaf */
  while (current->left != NULL) {
    current = current->left;
  }
  return current;
}

/* Helper function that allocates a new node with the given data and
    NULL left and right pointers. */
struct node* newNode(int data)
{
  struct node* node = (struct node*)
                      malloc(sizeof(struct node));
  node->data   = data;
  node->left   = NULL;
  node->right  = NULL;
  node->parent = NULL;

  return(node);
}

/* Give a binary search tree and a number, inserts a new node with
    the given number in the correct place in the tree. Returns the new
    root pointer which the caller should then use (the standard trick to
    avoid using reference parameters). */
struct node* insert(struct node* node, int data)
{
  /* 1. If the tree is empty, return a new,
      single node */
  if (node == NULL)
```

```
    return(newNode(data));
  else
  {
    struct node *temp;

    /* 2. Otherwise, recur down the tree */
    if (data <= node->data)
    {
         temp = insert(node->left, data);
         node->left  = temp;
         temp->parent= node;
    }
    else
    {
        temp = insert(node->right, data);
        node->right = temp;
        temp->parent = node;
    }

    /* return the (unchanged) node pointer */
    return node;
  }
}

/* Driver program to test above functions*/
int main()
{
  struct node* root = NULL, *temp, *succ, *min;

  //creating the tree given in the above diagram
  root = insert(root, 20);
  root = insert(root, 8);
  root = insert(root, 22);
  root = insert(root, 4);
  root = insert(root, 12);
  root = insert(root, 10);
  root = insert(root, 14);
  temp = root->left->right->right;

  succ =  inOrderSuccessor(root, temp);
  if(succ !=  NULL)
    printf("\n Inorder Successor of %d is %d ", temp->data, succ->data);
  getchar();
  return 0;
}
```

Output of the above program:

Inorder Successor of 14 is 20

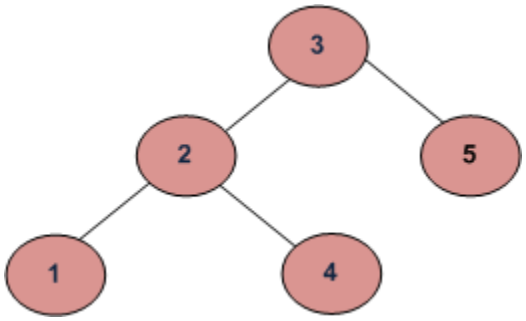
References:

http://net.pku.edu.cn/~course/cs101/2007/resource/Intro2Algorithm/book6/chap13.htm

35.Get Level of a node in a Binary Tree

Given a Binary Tree and a key, write a function that returns level of the key.

For example, consider the following tree. If the input key is 3, then your function should return 1. If the input key is 4, then your function should return 3. And for key which is not present in key, then your function should return 0.



The idea is to start from the root and level as 1. If the key matches with root's data, return level. Else recursively call for left and right subtrees with level as level + 1.

```
#include<stdio.h>

/* A tree node structure */
struct node
{
  int data;
  struct node *left;
  struct node *right;
};

/*
   Helper function for getLevel().  It returns level of the data if data
is
   present in tree, otherwise returns 0.
*/
int getLevelUtil(struct node *node, int data, int level)
{
  if ( node == NULL )
    return 0;

  if ( node->data == data )
    return level;

  return getLevelUtil ( node->left, data, level+1) |
         getLevelUtil ( node->right, data, level+1);
```

```
}

/* Returns level of given data value */
int getLevel(struct node *node, int data)
{
  return getLevelUtil(node,data,1);
}

/* Utility function to create a new Binary Tree node */
struct node* newNode(int data)
{
  struct node *temp = new struct node;
  temp->data = data;
  temp->left = NULL;
  temp->right = NULL;

  return temp;
}

/* Driver function to test above functions */
int main()
{
  struct node *root = new struct node;
  int x;

  /* Constructing tree given in the above figure */
  root = newNode(3);
  root->left = newNode(2);
  root->right = newNode(5);
  root->left->left = newNode(1);
  root->left->right = newNode(4);

  x = 3;
  printf(" Level of %d is %d", x, getLevel(root, x));

  x = 6;
  printf("\n Level of %d is %d", x, getLevel(root, x));

  getchar();
  return 0;
}
```

Output:

Level of 3 is 1

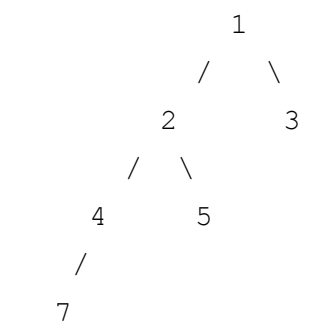
Level of 6 is 0

Time Complexity: O(n) where n is the number of nodes in the given Binary Tree.

36.Print Ancestors of a given node in Binary Tree

Given a Binary Tree and a key, write a function that prints all the ancestors of the key in the given binary tree.

For example, if the given tree is following Binary Tree and key is 7, then your function should print 4, 2 and 1.




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

using namespace std;

/* A binary tree node has data, pointer to left child
   and a pointer to right child */
struct node
{
   int data;
   struct node* left;
   struct node* right;
};

/* If target is present in tree, then prints the ancestors
   and returns true, otherwise returns false. */
bool printAncestors(struct node *root, int target)
{
  /* base cases */
  if ( root == NULL )
     return false;

  if ( root->data == target )
     return true;

  /* If target is present in either left or right subtree of this node,
     then print this node */
  if ( printAncestors(root->left, target) ||
```

```
        printAncestors(root->right, target) )
  {
    cout<<root->data<<" ";
    return true;
  }

  /* Else return false */
  return false;
}

/* Helper function that allocates a new node with the
   given data and NULL left and right pointers. */
struct node* newnode(int data)
{
  struct node* node = (struct node*)
                      malloc(sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;

  return(node);
}

/* Driver program to test above functions*/
int main()
{

  /* Construct the following binary tree
              1
            /   \
          2      3
        /  \
      4     5
     /
    7
  */
  struct node *root = newnode(1);
  root->left        = newnode(2);
  root->right       = newnode(3);
  root->left->left  = newnode(4);
  root->left->right = newnode(5);
  root->left->left->left  = newnode(7);

  printAncestors(root, 7);

  getchar();
  return 0;
}
```

Output:

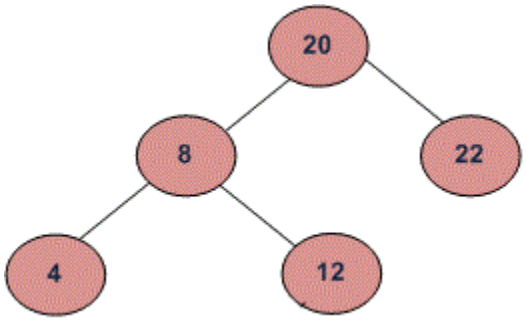
4 2 1

Time Complexity: O(n) where n is the number of nodes in the given Binary Tree.

37.Print BST keys in the given range

Given two values k1 and k2 (where k1 < k2) and a root pointer to a Binary Search Tree. Print all the keys of tree in range k1 to k2. i.e. print all x such that k1<=x<=k2 and x is a key of given BST. Print all the keys in increasing order.

For example, if k1 = 10 and k2 = 22, then your function should print 12, 20 and 22.



Algorithm:

1) If value of root's key is greater than k1, then recursively call in left subtree.

2) If value of root's key is in range, then print the root's key.

3) If value of root's key is smaller than k2, then recursively call in right subtree.

Implementation:

```
#include<stdio.h>

/* A tree node structure */
struct node
{
  int data;
  struct node *left;
  struct node *right;
};

/* The functions prints all the keys which in the given range [k1..k2].
    The function assumes than k1 < k2 */
void Print(struct node *root, int k1, int k2)
{
   /* base case */
   if ( NULL == root )
      return;

   /* Since the desired o/p is sorted, recurse for left subtree first
      If root->data is greater than k1, then only we can get o/p keys
      in left subtree */
```

```
   if ( k1 < root->data )
     Print(root->left, k1, k2);

   /* if root's data lies in range, then prints root's data */
   if ( k1 <= root->data && k2 >= root->data )
     printf("%d ", root->data );

  /* If root->data is smaller than k2, then only we can get o/p keys
      in right subtree */
   if ( k2 > root->data )
     Print(root->right, k1, k2);
}

/* Utility function to create a new Binary Tree node */
struct node* newNode(int data)
{
  struct node *temp = new struct node;
  temp->data = data;
  temp->left = NULL;
  temp->right = NULL;

  return temp;
}

/* Driver function to test above functions */
int main()
{
  struct node *root = new struct node;
  int k1 = 10, k2 = 25;

  /* Constructing tree given in the above figure */
  root = newNode(20);
  root->left = newNode(8);
  root->right = newNode(22);
  root->left->left = newNode(4);
  root->left->right = newNode(12);

  Print(root, k1, k2);

  getchar();
  return 0;
}
```

Output:

12 20 22

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

38.Tournament Tree (Winner Tree) and Binary Heap

Given a team of N players. How many minimum games are required to find second best player?

We can use adversary arguments based on tournament tree (Binary Heap).

Tournament tree is a form of min (max) heap which is a complete binary tree. Every external node represents a player and internal node represents winner. In a tournament tree every internal node contains winner and every leaf node contains one player.

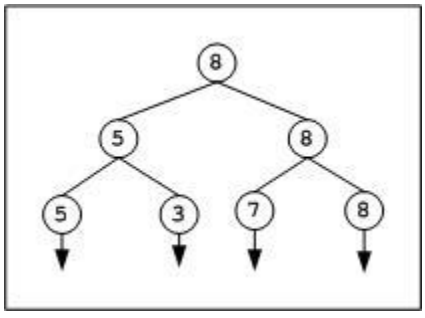
There will be N – 1 internal nodes in a binary tree with N leaf (external) nodes. For details see this post(put n = 2 in equation given in the post).

It is obvious that to select the best player among N players, (N – 1) players to be eliminated, i.e. we need minimum of (N – 1) games (comparisons). Mathematically we can prove it. In a binary tree I = E – 1, where I is number of internal nodes and E is number of external nodes. It means to find maximum or minimum element of an array, we need N – 1 (internal nodes) comparisons.

Second Best Player

The information explored during best player selection can be used to minimize the number of comparisons in tracing the next best players. For example, we can pick second best player in **$(N + \log_2 N - 2)$** comparisons. For details read this comment.

The following diagram displays a tournament tree (*winner tree*) as a max heap. Note that the concept of *loser tree* is different.



The above tree contains 4 leaf nodes that represent players and have 3 levels 0, 1 and 2. Initially 2 games are conducted at level 2, one between 5 and 3 and another one between 7 and 8. In the next move, one more game is conducted between 5 and 8 to conclude the final winner. Overall we need 3

comparisons. For second best player we need to trace the candidates participated with final winner, that leads to 7 as second best.

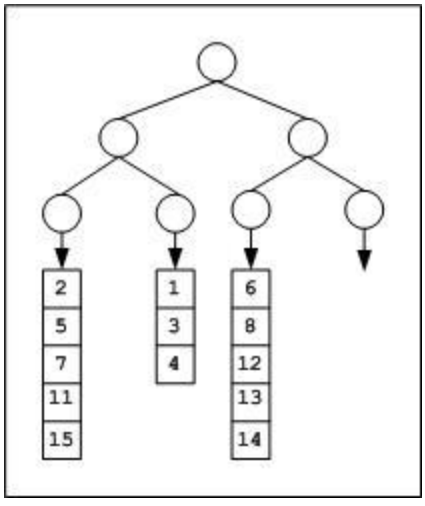
Median of Sorted Arrays

Tournament tree can effectively be used to find median of sorted arrays. Assume, given M sorted arrays of equal size L (for simplicity). We can attach all these sorted arrays to the tournament tree, one array per leaf. We need a tree of height **CEIL (log_2M)** to have atleast M external nodes.

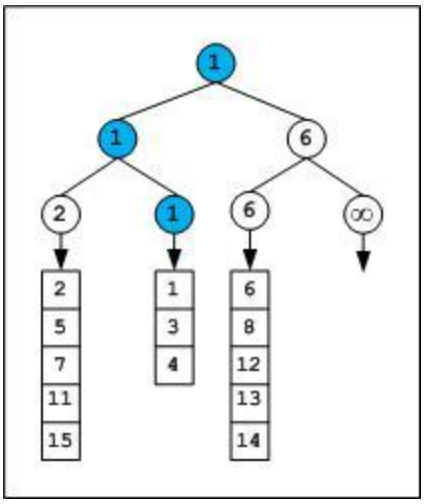
Consider an example. Given 3 (M = 3) sorted integer arrays of maximum size 5 elements.

```
{ 2, 5, 7, 11, 15 } ---- Array1
{1, 3, 4} ---- Array2
{6, 8, 12, 13, 14} ---- Array3
```

What should be the height of tournament tree? We need to construct a tournament tree of height $log_2$3 .= 1.585 = 2 rounded to next integer. A binary tree of height 2 will have 4 leaves to which we can attach the arrays as shown in the below figure.



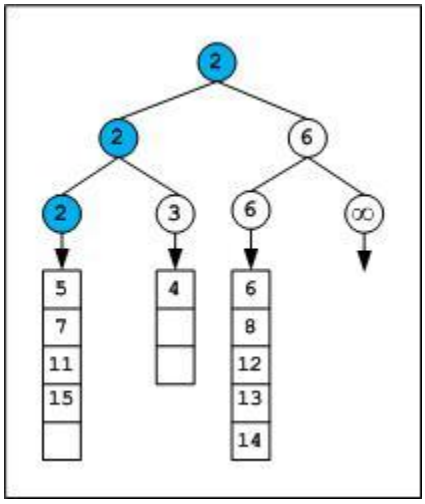
After the first tournament, the tree appears as below,



We can observe that the winner is from Array2. Hence the next element from Array2 will dive-in and games will be played along the winner path of previous tournament.

Note that infinity is used as sentinel element. Based on data being hold in nodes we can select the sentinel character. For example we usually store the pointers in nodes rather than keys, so NULL can serve as sentinel. If any of the array exhausts we will fill the corresponding leaf and upcoming internal nodes with sentinel.

After the second tournament, the tree appears as below,



The next winner is from Array1, so next element of Array1 array which is 5 will dive-in to the next round, and next tournament played along the path of 2.

The tournaments can be continued till we get median element which is (5+3+5)/2 = 7th element. Note that there are even better algorithms for finding median of union of sorted arrays, for details see the related links given below.

In general with M sorted lists of size L_1, L_2 … L_m requires time complexity of ***O((L₁ + L₂ + … + Lₘ) * logM)*** to merge all the arrays, and ***O(m*logM)*** time to find median, where ***m*** is median position.

Select smallest one million elements from one billion unsorted elements: Read the Source.

As a simple solution, we can sort the billion numbers and select first one million.

On a limited memory system sorting billion elements and picking the first one million seems to be impractical. We can use tournament tree approach. At any time only elements of tree to be in memory.

Split the large array (perhaps stored on disk) into smaller size arrays of size one million each (or even smaller that can be sorted by the machine). Sort these 1000 small size arrays and store them on disk as individual files. Construct a tournament tree which can have atleast 1000 leaf nodes (tree to be of height 10 since $2^9 < 1000 < 2^{10}$, if the individual file size is even smaller we will need more leaf nodes). Every leaf node will have an engine that picks next element from the sorted file stored on disk. We can play the tournament tree game to extract first one million elements.

Total cost = sorting 1000 lists of one million each + tree construction + tournaments

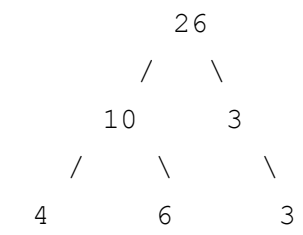
Implementation

We need to build the tree (heap) in bottom-up manner. All the leaf nodes filled first. Start at the left extreme of tree and fill along the breadth (i.e. from 2^{k-1} to $2^k – 1$ where k is depth of tree) and play the game. After practicing with few examples it will be easy to write code. We will have code in an upcoming article.

39.Check if a given Binary Tree is SumTree

Write a function that returns true if the given Binary Tree is SumTree else false. A SumTree is a Binary Tree where the value of a node is equal to sum of the nodes present in its left subtree and right subtree. An empty tree is SumTree and sum of an empty tree can be considered as 0. A leaf node is also considered as SumTree.

Following is an example of SumTree.



Method 1 (Simple)

Get the sum of nodes in left subtree and right subtree. Check if the sum calculated is equal to root's data. Also, recursively check if the left and right subtrees are SumTrees.

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

/* A binary tree node has data, left child and right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* A utility function to get the sum of values in tree with root
  as root */
int sum(struct node *root)
{
   if(root == NULL)
     return 0;
   return sum(root->left) + root->data + sum(root->right);
}

/* returns 1 if sum property holds for the given
    node and both of its children */
int isSumTree(struct node* node)
{
    int ls, rs;

    /* If node is NULL or it's a leaf node then
       return true */
    if(node == NULL ||
            (node->left == NULL && node->right == NULL))
        return 1;

   /* Get sum of nodes in left and right subtrees */
   ls = sum(node->left);
   rs = sum(node->right);

   /* if the node and both of its children satisfy the
       property return 1 else 0*/
    if((node->data == ls + rs)&&
            isSumTree(node->left) &&
```

```
            isSumTree(node->right))
        return 1;

   return 0;
}

/*
 Helper function that allocates a new node
 with the given data and NULL left and right
 pointers.
*/
struct node* newNode(int data)
{
    struct node* node =
        (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

/* Driver program to test above function */
int main()
{
    struct node *root  = newNode(26);
    root->left         = newNode(10);
    root->right        = newNode(3);
    root->left->left   = newNode(4);
    root->left->right  = newNode(6);
    root->right->right = newNode(3);
    if(isSumTree(root))
        printf("The given tree is a SumTree ");
    else
        printf("The given tree is not a SumTree ");

    getchar();
    return 0;
}
```

Time Complexity: O(n^2) in worst case. Worst case occurs for a skewed tree.

Method 2 (Tricky)

The Method 1 uses sum() to get the sum of nodes in left and right subtrees. The method 2 uses following rules to get the sum directly.

1) If the node is a leaf node then sum of subtree rooted with this node is equal to value of this node.

2) If the node is not a leaf node then sum of subtree rooted with this node is twice the value of this node (Assuming that the tree rooted with this node is SumTree).

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

```
/* A binary tree node has data, left child and right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* Utillity function to check if the given node is leaf or not */
int isLeaf(struct node *node)
{
    if(node == NULL)
        return 0;
    if(node->left == NULL && node->right == NULL)
        return 1;
    return 0;
}

/* returns 1 if SumTree property holds for the given
    tree */
int isSumTree(struct node* node)
{
    int ls; // for sum of nodes in left subtree
    int rs; // for sum of nodes in right subtree

    /* If node is NULL or it's a leaf node then
       return true */
    if(node == NULL || isLeaf(node))
        return 1;

    if( isSumTree(node->left) && isSumTree(node->right))
    {
        // Get the sum of nodes in left subtree
        if(node->left == NULL)
            ls = 0;
        else if(isLeaf(node->left))
            ls = node->left->data;
        else
            ls = 2*(node->left->data);

        // Get the sum of nodes in right subtree
        if(node->right == NULL)
            rs = 0;
        else if(isLeaf(node->right))
            rs = node->right->data;
        else
            rs = 2*(node->right->data);

        /* If root's data is equal to sum of nodes in left
           and right subtrees then return 1 else return 0*/
        return(node->data == ls + rs);
    }

    return 0;
}
```

```
/* Helper function that allocates a new node
 with the given data and NULL left and right
 pointers.
*/
struct node* newNode(int data)
{
    struct node* node =
        (struct node*)malloc(sizeof(struct node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;
    return(node);
}

/* Driver program to test above function */
int main()
{
    struct node *root  = newNode(26);
    root->left         = newNode(10);
    root->right        = newNode(3);
    root->left->left   = newNode(4);
    root->left->right  = newNode(6);
    root->right->right = newNode(3);
    if(isSumTree(root))
        printf("The given tree is a SumTree ");
    else
        printf("The given tree is not a SumTree ");

    getchar();
    return 0;
}
```

40.Decision Trees – Fake (Counterfeit) Coin Puzzle

Let us solve the “fake coin” puzzle using decision trees. There are two variants of the puzzle. One such variant is given below (try to solve on your own, assume N = 8), another variant is given as next problem.

*We are provided a two pan fair balance and N identically looking coins, out of which only one coin **may be** defective. How can we trace which coin, if any, is odd one, and also determine whether it is lighter or heavier in minimum number of trials in the worst case?*

Let us start with relatively simple examples. After reading every problem try to solve on your own.

Problem 1: (Easy)

*Given 5 coins out of which one coin is **lighter**. In the worst case, how many minimum number of weighing are required to figure out the odd coin?*

Name the coins as 1, 2, 3, 4 and 5. We know that one coin is lighter. Considering best out come of balance, we can group the coins in two different ways, [(1, 2), (3, 4) and (5)], or [(12), (34) and (5)]. We can easily rule out groups like [(123) and (45)], as we will get obvious answer. Any other combination will fall into one of these two groups, like [(2)(45) and (13)], etc.

Consider the first group, pairs (1, 2) and (3, 4). We can check (1, 2), if they are equal we go ahead with (3, 4). We need two weighing in worst case. The same analogy can be applied when the coin in heavier.

With the second group, weigh (12) and (34). If they balance (5) is defective one, otherwise pick the lighter pair, and we need one more weighing to find odd one.

Both the combinations need two weighing in case of 5 coins with prior information of one coin is lighter.

Analysis: In general, if we know that the coin is heavy or light, we can trace the coin in $\log_3(N)$ trials (rounded to next integer). If we represent the outcome of balance as ternary tree, every leaf represent an outcome. Since any coin among N coins can be defective, we need to get a 3-ary tree having minimum of N leaves. A 3-ary tree at k-th level will have 3^k leaves and hence we need $3^k >= N$.

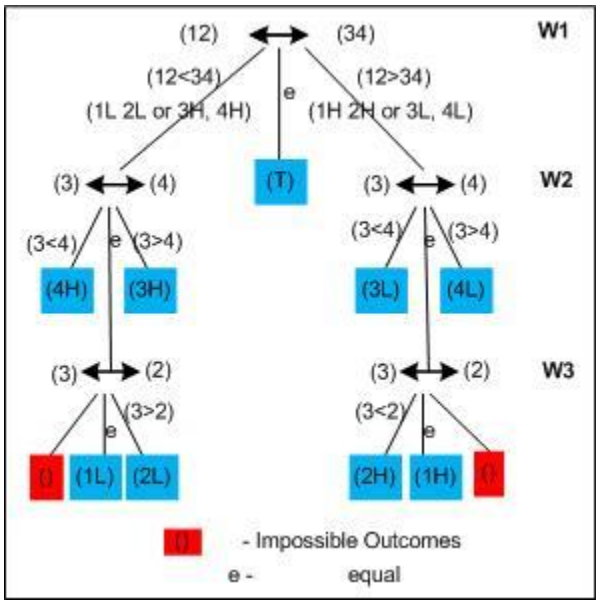
In other-words, in ***k*** trials we can examine upto $\boldsymbol{3^k}$ coins, if we know whether the defective coin is heavier or lighter. Given that a coin is heavier, verify that 3 trials are sufficient to find the odd coin among 12 coins, because $3^2 < 12 < 3^3$.

Problem 2: (Difficult)

*We are given 4 coins, out of which only one coin **may be** defective. We don't know, whether all coins are genuine or any defective one is present. How many number of weighing are required in worst case to figure out the odd coin, if present? We also need to tell whether it is heavier or lighter.*

From the above analysis we may think that k = 2 trials are sufficient, since a two level 3-ary tree yields 9 leaves which is greater than N = 4 (read the problem once again). Note that it is impossible to solve above 4 coins problem in two weighing. The decision tree confirms the fact (try to draw).

We can group the coins in two different ways, [(12, 34)] or [(1, 2) and (3, 4)]. Let us consider the combination (12, 34), the corresponding decision tree is given below. Blue leaves are valid outcomes, and red leaves are impossible cases. We arrived at impossible cases due to the assumptions made earlier on the path.



The outcome can be (12) < (34) i.e. we go on to left subtree or (12) > (34) i.e. we go on to right subtree.

The left subtree is possible in two ways,

- A) Either 1 or 2 can be lighter OR
- B) Either 3 or 4 can be heavier.

Further on the left subtree, as second trial, we weigh (1, 2) or (3, 4). Let us consider (3, 4) as the analogy for (1, 2) is similar. The outcome of second trail can be three ways

- A) (3) < (4) yielding 4 as defective heavier coin, OR
- B) (3) > (4) yielding 3 as defective heavier coin OR
- C) (3) = (4), yielding ambiguity. Here we need one more weighing to check a genuine coin against 1 or 2. In the figure I took (3, 2) where 3 is confirmed as genuine. We can get (3) > (2) in which 2 is lighter, or (3) = (2) in which 1 is lighter. Note that it impossible to get (3) < (2), it contradicts our assumption leaned to left side.

Similarly we can analyze the right subtree. We need two more weighings on right subtree as well.

Overall we need 3 weighings to trace the odd coin. Note that we are unable to utilize two outcomes of 3-ary trees. Also, the tree is not full tree, middle branch terminated after first weighing. Infact, we can get 27 leaves of 3 level full 3-ary tree, but only we got 11 leaves including impossible cases.

Analysis: Given N coins, all may be genuine or only one coin is defective. We need a decision tree with atleast (2N + 1) leaves correspond to the outputs. Because there can be N leaves to be lighter, or N leaves to be heavier or one genuine case, on total (2N + 1) leaves.

As explained earlier ternary tree at level k, can have utmost 3^k leaves and we need a tree with leaves of $3^k > (2N + 1)$.

In other words, we need atleast $k > log_3(2N + 1)$ weighing to find the defective one.

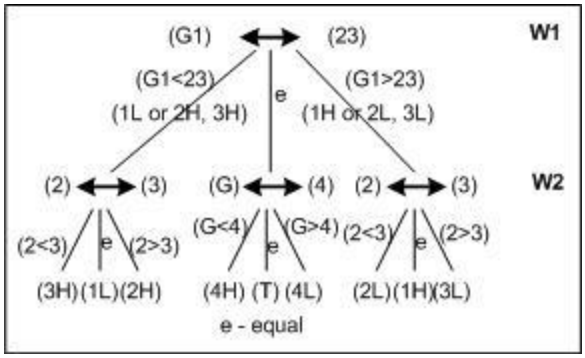
Observe the above figure that not all the branches are generating leaves, i.e. we are missing valid outputs under some branches that leading to more number of trials. When possible, we should group the coins in such a way that every branch is yielding valid output (in simple terms generate full 3-ary tree). Problem 4 describes this approach of 12 coins.

Problem 3: (Special case of two pan balance)

*We are given 5 coins, a group of 4 coins out of which one coin is defective (we **don't know** whether it is heavier or lighter), and one coin is genuine. How many weighing are required in worst case to figure out the odd coin whether it is heavier or lighter?*

Label the coins as 1, 2, 3, 4 and G (genuine). We now have some information on coin purity. We need to make use that in the groupings.

We can best group them as [(G1, 23) and (4)]. Any other group can't generate full 3-ary tree, try yourself. The following diagram explains the procedure.



The middle case (G1) = (23) is self explanatory, i.e. 1, 2, 3 are genuine and 4th coin can be figured out lighter or heavier in one more trial.

The left side of tree corresponds to the case (G1) < (23). This is possible in two ways, either 1 should be lighter or either of (2, 3) should be heavier. The former instance is obvious when next weighing (2, 3) is balanced, yielding 1 as lighter. The later instance can be (2) < (3) yielding 3 as heavier or (2) > (3) yielding 2 as heavier. The leaf nodes on left branch are named to reflect these outcomes.

The right side of tree corresponds to the case (G1) > (23). This is possible in two ways, either 1 is heavier or either of (2, 3) should be lighter. The former instance is obvious when the next weighing (2, 3) is balanced, yielding 1 as heavier. The later case can be (2) < (3) yielding 2 as lighter coin, or (2) > (3) yielding 3 as lighter.

In the above problem, under any possibility we need only two weighing. We are able to use all outcomes of two level full 3-ary tree. We started with (N + 1) = 5 coins where N = 4, we end up with (2N + 1) = 9 leaves. *Infact we should have 11 outcomes since we stared with 5 coins, where are other 2 outcomes? These two outcomes can be declared at the root of tree itself (prior to first weighing), can you figure out?*

If we observe the figure, after the first weighing the problem reduced to “we know three coins, either one can be lighter (heavier) or one among other two can be heavier (lighter)”. This can be solved in one weighing (read Problem 1).

Analysis: Given (N + 1) coins, one is genuine and the rest N can be genuine or only one coin is defective. The required decision tree should result in minimum of (2N + 1) leaves. Since the total possible outcomes are (2(N + 1) + 1), number of weighing (trials) are given by the height of ternary tree, $k >= \log_3[2(N + 1) + 1]$. *Note the equality sign*.

Rearranging k and N, *we can weigh maximum of* $N <= (3^k – 3)/2$ *coins in k trials*.

Problem 4: (The classic 12 coin puzzle)

You are given two pan fair balance. You have 12 identically looking coins out of which one coin may be lighter or heavier. How can you find odd coin, if any, in minimum trials, also determine whether defective coin is lighter or heavier, in the worst case?

How do you want to group them? Bi-set or tri-set? Clearly we can discard the option of dividing into two equal groups. It can’t lead to best tree. Remember to group coins such that the first weighing reveals atleast one genuine coin.

Let us name the coins as 1, 2, … 8, A, B, C and D. We can combine the coins into 3 groups, namely (1234), (5678) and (ABCD). Weigh (1234) and (5678). You are encouraged to draw decision tree while reading the procedure. The outcome can be three ways,

1. (1234) = (5678), both groups are equal. Defective coin may be in (ABCD) group.
2. (1234) < (5678), i.e. first group is less in weight than second group.
3. (1234) > (5678), i.e. first group is more in weight than second group.

The output (1) can be solved in two more weighing as special case of two pan balance given in Problem 3. We know that groups (1234) and (5678) are genuine and defective coin may be in (ABCD). Pick one genuine coin from any of weighed groups, and proceed with (ABCD) as explained in Problem 3.

Outcomes (2) and (3) are special. In both the cases, we know that (ABCD) is genuine. And also, we know a set of coins being lighter and a set of coins being heavier. We need to shuffle the weighed two groups in such a way that we end up with smaller height decision tree.

Consider the second outcome where (1234) < (5678). It is possible when any coin among (1, 2, 3, 4) is lighter or any coin among (5, 6, 7, 8) is heavier. We revealed lighter or heavier possibility after first weighing. If we proceed as in Problem 1, we will not generate best decision tree. Let us shuffle coins as (1235) and (4BCD) as new groups (there are different shuffles possible, they also lead to minimum weighing, can you try?). If we weigh these two groups again the outcome can be three ways, i) (1235) < (4BCD) yielding one among 1, 2, 3 is lighter which is similar to Problem 1 explained above, we need one more weighing, ii) (1235) = (4BCD) yielding one among 6, 7, 8 is heavier which is similar to Problem 1 explained above, we need one more weighing iii) (1235) > (4BCD) yielding either 5 as heavier coin or 4 as lighter coin, at the expense of one more weighing.

Similar way we can also solve the right subtree (third outcome where (1234) > (5678)) in two more weighing.

We are able to solve the 12 coin puzzle in 3 weighing worst case.

Few Interesting Puzzles:

1. Solve Problem 4 with N = 8 and N = 13, How many minimum trials are required in each case?
2. Given a function *int weigh(A[], B[])* where A and B are arrays (need not be equal size). The function returns -1, 0 or 1. It returns 0 if sum of all elements in A and B are equal, -1 if A < B and 1 if A > B. Given an array of 12 elements, all elements are equal except one. The odd element can be as that of

others, smaller or greater than other. Write a program to find the odd element (if any) using *weigh()*minimum times.

3. You might have seen 3-pan balance in science labs during school days. Given a 3-pan balance (4 outcomes) and N coins, how many minimum trials are need to figure out odd coin?

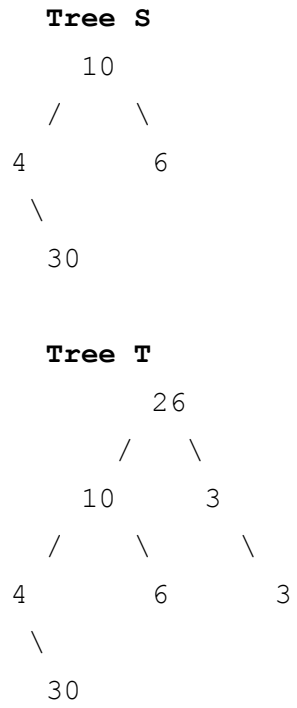
References:

Similar problem was provided in one of the exercises of the book ”Introduction to Algorithms by Levitin”. Specifically read section 5.5 and section 11.2 including exercises.

41.Check if a binary tree is subtree of another binary tree

Given two binary trees, check if the first tree is subtree of the second one. A subtree of a tree T is a tree S consisting of a node in T and all of its descendants in T. The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree.

For example, in the following case, tree S is a subtree of tree T.



Solution: Traverse the tree T in inorder fashion. For every visited node in the traversal, see if the subtree rooted with this node is identical to S.

Following is C implementation for this.

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

/* A binary tree node has data, left child and right child */
struct node
{
    int data;
    struct node* left;
    struct node* right;
};

/* A utility function to check whether trees with roots as root1 and
   root2 are identical or not */
bool areIdentical(struct node * root1, struct node *root2)
{
    /* base cases */
    if(root1 == NULL && root2 == NULL)
        return true;

    if(root1 == NULL || root2 == NULL)
        return false;

    /* Check if the data of both roots is same and data of left and right
       subtrees are also same */
    return (root1->data == root2->data   &&
            areIdentical(root1->left, root2->left) &&
            areIdentical(root1->right, root2->right) );
}

/* This function returns true if S is a subtree of T, otherwise false */
bool isSubtree(struct node *T, struct node *S)
{
    /* base cases */
    if (S == NULL)
        return true;

    if (T == NULL)
        return false;

    /* Check the tree with root as current node */
    if (areIdentical(T, S))
        return true;

    /* If the tree with root as current node doesn't match then
       try left and right subtrees one by one */
    return isSubtree(T->left, S) ||
           isSubtree(T->right, S);
}
```

```
/* Helper function that allocates a new node with the given data
   and NULL left and right pointers. */
struct node* newNode(int data)
{
    struct node* node =
        (struct node*)malloc(sizeof(struct node));
    node->data  = data;
    node->left  = NULL;
    node->right = NULL;
    return(node);
}

/* Driver program to test above function */
int main()
{
    /* Construct the following tree
              26
            /   \
          10     3
        /    \     \
      4      6      3
       \
        30
    */
    struct node *T        = newNode(26);
    T->right              = newNode(3);
    T->right->right       = newNode(3);
    T->left               = newNode(10);
    T->left->left         = newNode(4);
    T->left->left->right  = newNode(30);
    T->left->right        = newNode(6);

    /* Construct the following tree
          10
        /    \
      4      6
       \
        30
    */
    struct node *S    = newNode(10);
    S->right          = newNode(6);
    S->left           = newNode(4);
    S->left->right    = newNode(30);

    if( isSubtree(T, S) )
        printf("Tree S is subtree of tree T");
    else
        printf("Tree S is not a subtree of tree T");

    getchar();
    return 0;
}
```