1.

```cpp
#include <bits/stdc++.h>
using namespace std;

struct Edge{
    int from, to, length;
};

struct Graph{
    int V, E;  // V-> Number of vertices, E-> Number of edges
    struct Edge *edge;
};

struct Graph *createGraph(int V, int E){
    struct Graph *graph = new Graph;
    graph->V = V;
    graph->E = E;
    graph->edge = new Edge[E];
    return graph;
}

void printArr(int dist[], int n){
    cout << "Vertex distance from start: " << endl;
    for (int i = 0; i < n; i++){
        cout << "No: " << i << "\t" << dist[i] << endl;
    }
}

void BellmanFord(struct Graph *graph, int start){
    int V = graph->V;
    int E = graph->E;
    int dist[V];

    for (int i = 0; i < V; i++){
        dist[i] = INT_MAX;
    }
    dist[start] = 0;
```

```
    // shortest path from start to any other vertex can have
    // at-most |V| - 1 edges
    for (int i = 1; i <= V - 1; i++){
        for (int j = 0; j < E; j++){
            int u = graph->edge[j].from;
            int v = graph->edge[j].to;
            int w = graph->edge[j].length;
            if (dist[u] != INT_MAX && dist[u] + w < dist[v])
                dist[v] = dist[u] + w;
        }
    }

    // check for negative-weight cycles. The above
    for (int i = 0; i < E; i++){
        int u = graph->edge[i].from;
        int v = graph->edge[i].to;
        int weight = graph->edge[i].length;
        if (dist[u] != INT_MAX && dist[u] + weight < dist[v]){
            cout << "Graph contains negative length cycle";
            return;       // If negative cycle is detected, simply
return
        }
    }
    printArr(dist, V);
    return;
}

int main(){
    int V = 5;
    int E = 6;
    struct Graph *graph = createGraph(V, E);

    graph->edge[0].from = 0;
    graph->edge[0].to = 1;
    graph->edge[0].length = 4;

    graph->edge[1].from = 1;
    graph->edge[1].to = 2;
```

```
    graph->edge[1].length = 3;

    graph->edge[2].from = 1;
    graph->edge[2].to = 3;
    graph->edge[2].length = 2;

    graph->edge[3].from = 1;
    graph->edge[3].to = 4;
    graph->edge[3].length = 5;

    graph->edge[4].from = 3;
    graph->edge[4].to = 4;
    graph->edge[4].length = 1;

    graph->edge[5].from = 4;
    graph->edge[5].to = 2;
    graph->edge[5].length = 1;

    BellmanFord(graph, 0);
    return 0;
}
```

```
Vertex distance from start:
No: 0    0
No: 1    4
No: 2    7
No: 3    6
No: 4    7
```

2.

End of the first for-loop

30, 20, 28, 12, 18, 16, 4, 10, 2, 6, 8

Second for-loop

Iteration 1

28, 20, 16, 12, 18, 8, 4, 10, 2, 6, 30

Iteration 2

20, 18, 16, 12, 6, 8, 4, 10, 2, 28, 30

Iteration 3

18, 12, 16, 10, 6, 8, 4, 2, 20, 28, 30

Iteration 4

16, 12, 8, 10, 6, 2, 4, 18, 20, 28, 30

Iteration 5

12, 10, 8, 4, 6, 2, 16, 18, 20, 28, 30

Iteration 6

10, 6, 8, 4, 2, 12, 16, 18, 20, 28, 30

Iteration 7

8, 6, 2, 4, 10, 12, 16, 18, 20, 28, 30

Iteration 8

6, 4, 2, 8, 10, 12, 16, 18, 20, 28, 30

Iteration 9

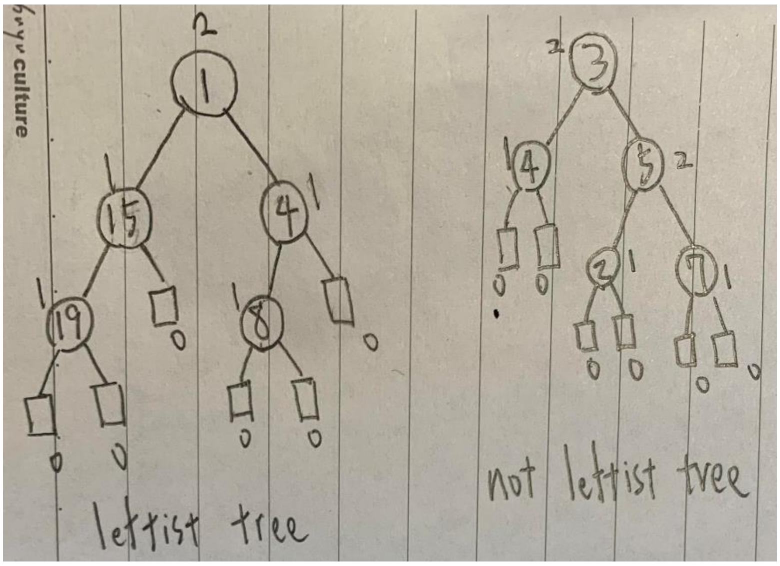
4, 2, 6, 8, 10, 12, 16, 18, 20, 28, 30

Iteration 10

2, 4, 6, 8, 10, 12, 16, 18, 20, 28, 30

#

3.



4.

No.
Date : :

According to Mathematical induction on k. The binomial tree B_0 is the base binomial tree for $k=0$. By definition, B_0 is a single node. Consider a binomial tree B_k, $k \geq 1$. Since B_k is constructed using two copies of B_{k-1}, by the hypothesis, each B_{k-1} has 2^{k-1} nodes. Thus, B_k has $2^{k-1}+2^{k-1}=2^k$ nodes ※

5.

```
#include <bits/stdc++.h>
using namespace std;

struct node {
    int key;
    int element;
    struct node *left, *right;
};

struct node* newNode(int item){
    struct node* temp = new node;
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}
```

```
void inorder(struct node* root){
    if (root != NULL) {
        inorder(root->left);
        cout << root->key << " ";
        inorder(root->right);
    }
}

struct node* insert(struct node* node, int key,int element){
    /* If the tree is empty, return a new node */
    if (node == NULL)
        return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left = insert(node->left, key, element);
    else
        node->right = insert(node->right, key, element);

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

struct node* minValueNode(struct node* node){
    struct node* current = node;
    while (current && current->left != NULL)
        current = current->left;
    return current;
}

struct node* deleteNode(struct node* root, int key){
    if (root == NULL)
        return root;
    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
```

```
    else {
        if (root->left==NULL and root->right==NULL)
            return NULL;
        else if (root->left == NULL) {
            struct node* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL) {
            struct node* temp = root->left;
            free(root);
            return temp;
        }

        // node with two children: Get the inorder successor
        // (smallest in the right subtree)
        struct node* temp = minValueNode(root->right);

        // Copy the inorder successor's content to this node
        root->key = temp->key;

        // Delete the inorder successor
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}

int main(){
    /*   tree to create
            8
         /     \
        3      10
       / \     / \
      1   6   9   14 */
    struct node* root = NULL;
    root = insert(root, 8, 2);
    root = insert(root, 3, 3);
    root = insert(root, 1, 1);
```

```
    root = insert(root, 6, 4);
    root = insert(root, 10, 5);
    root = insert(root, 9, 4);
    root = insert(root, 14, 3);

    cout << "Inorder traversal of the given tree \n";
    inorder(root);

    cout << "\nDelete 1\n";
    root = deleteNode(root, 1);
    cout << "Inorder traversal of the modified tree \n";
    inorder(root);

    cout << "\nDelete 3\n";
    root = deleteNode(root, 3);
    cout << "Inorder traversal of the modified tree \n";
    inorder(root);

    cout << "\nDelete 8\n";
    root = deleteNode(root, 8);
    cout << "Inorder traversal of the modified tree \n";
    inorder(root);
}
```

```
Inorder traversal of the given tree
1 3 6 8 9 10 14
Delete 1
Inorder traversal of the modified tree
3 6 8 9 10 14
Delete 3
Inorder traversal of the modified tree
6 8 9 10 14
Delete 8
Inorder traversal of the modified tree
6 9 10 14
```

time complexity is O(h), where h is for the height.

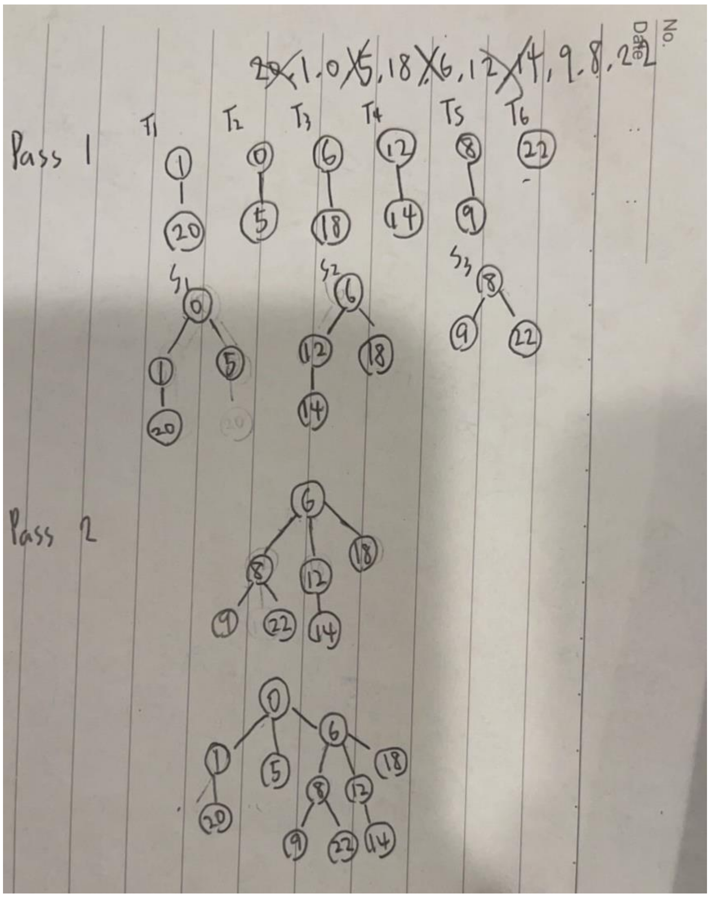
6.

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    int length;
```

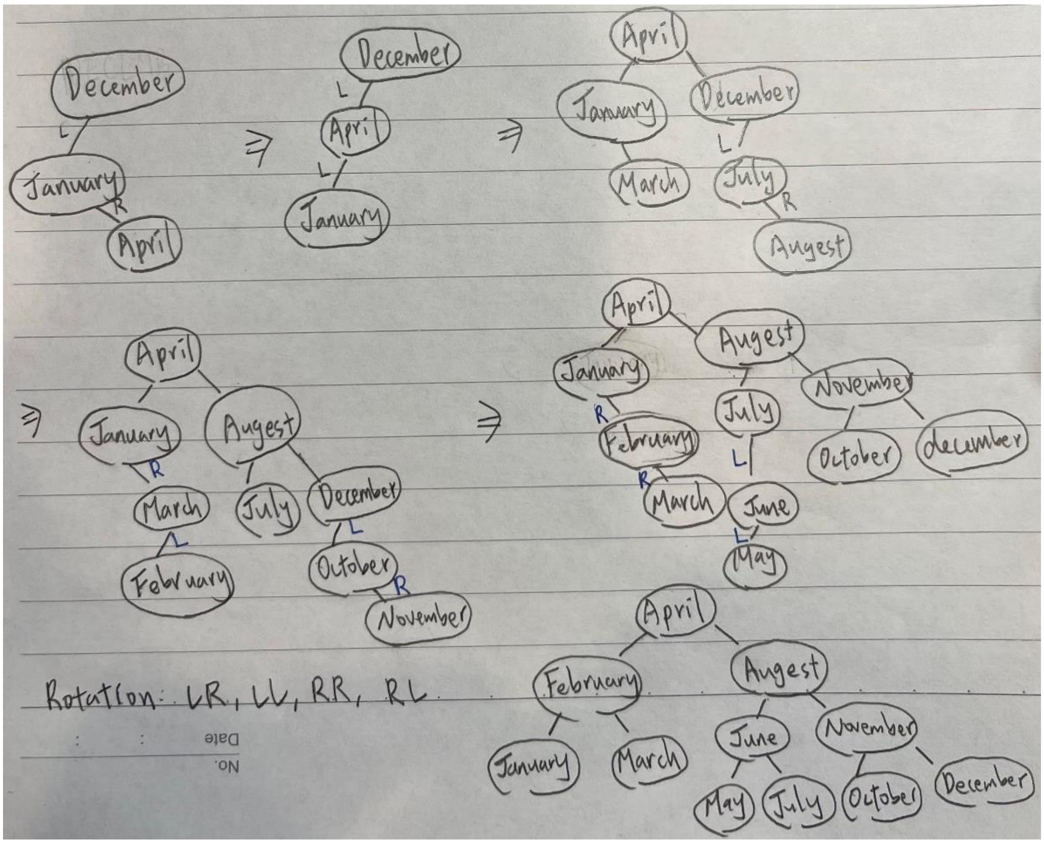
```cpp
    struct Node *left, *right;
};

Node* newNode(int data,int length){
    Node* temp = new Node;
    temp->data = data;
    temp->length = length;
    temp->left = temp->right = NULL;
    return temp;
}

void dfs(struct Node* node){
    if(node){
        if(node->left) node->left->length += node->length;
        if(node->right) node->right->length += node->length;
        cout << node->data << "\t\t" << node->length << endl;
        dfs(node->left);
        dfs(node->right);
    }
}

int main(){
    struct Node* v = newNode(0,0);
    v->left = newNode(1,5);
    v->right = newNode(2,9);
    v->left->left = newNode(3,3);
    v->left->right = newNode(4,2);
    cout << "vertex: " << " " << "shortest path from the root:" << endl;
    dfs(v);

    return 0;
}
```

```
vertex:  shortest path from the root:
0               0
1               5
3               8
4               7
2               9
```

7.



8.



9.

```
#include<bits/stdc++.h>
using namespace std;

class Node{
    public:
    int key;
    Node *left;
    Node *right;
    int height;
};

// A utility function to get maximum
// of two integers
int max(int a, int b);

// A utility function to get the
// height of the tree
int height(Node *N){
    if (N == NULL)
```

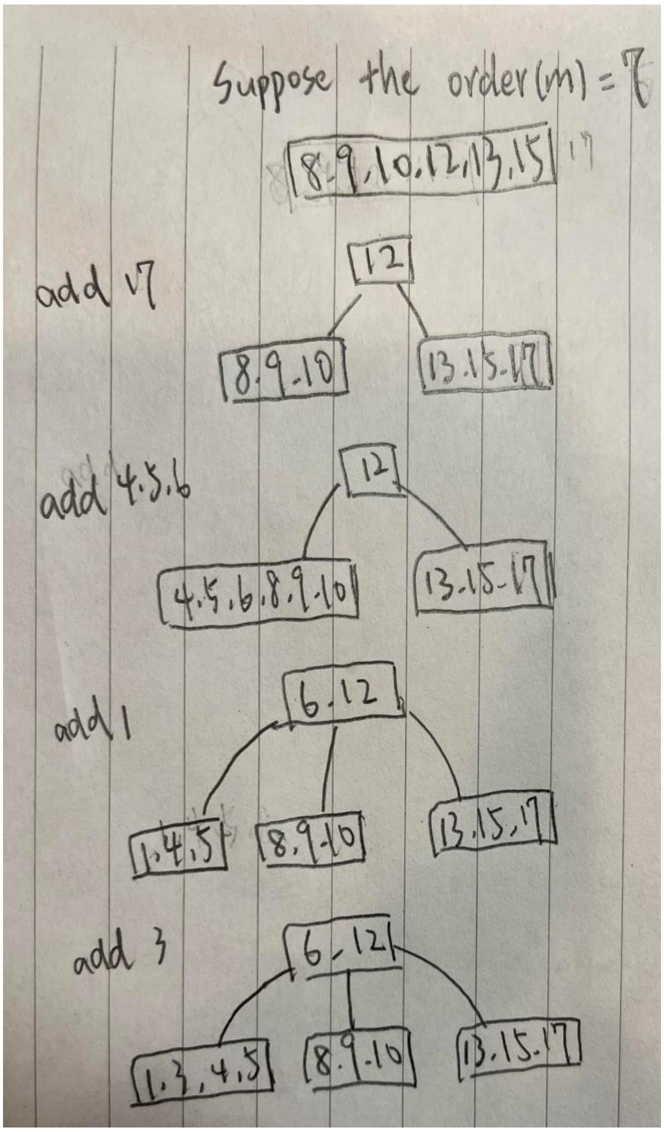
```
        return 0;
    return N->height;
}

// A utility function to get maximum
// of two integers
int max(int a, int b){
    return (a > b)? a : b;
}

Node* newNode(int key){
    Node* node = new Node();
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1;
    return(node);
}

// A utility function to right
// rotate subtree rooted with y
// See the diagram given above.
Node *rightRotate(Node *y){
    Node *x = y->left;
    Node *T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = max(height(y->left),
                    height(y->right)) + 1;
    x->height = max(height(x->left),
                    height(x->right)) + 1;

    // Return new root
    return x;
```

```
}

Node *leftRotate(Node *x){
    Node *y = x->right;
    Node *T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = max(height(x->left),
                    height(x->right)) + 1;
    y->height = max(height(y->left),
                    height(y->right)) + 1;

    // Return new root
    return y;
}

// Get Balance factor of node N
int getBalance(Node *N){
    if (N == NULL)
        return 0;
    return height(N->left) - height(N->right);
}

// Recursive function to insert a key
// in the subtree rooted with node and
// returns the new root of the subtree.
Node* insert(Node* node, int key){
    /* 1. Perform the normal BST insertion */
    if (node == NULL)
        return(newNode(key));

    if (key < node->key)
        node->left = insert(node->left, key);
    else if (key > node->key)
```

```
    node->right = insert(node->right, key);
else // Equal keys are not allowed in BST
    return node;

/* 2. Update height of this ancestor node */
node->height = 1 + max(height(node->left),
                       height(node->right));

/* 3. Get the balance factor of this ancestor
    node to check whether this node became
    unbalanced */
int balance = getBalance(node);

// If this node becomes unbalanced, then
// there are 4 cases

// Left Left Case
if (balance > 1 && key < node->left->key)
    return rightRotate(node);

// Right Right Case
if (balance < -1 && key > node->right->key)
    return leftRotate(node);

// Left Right Case
if (balance > 1 && key > node->left->key){
    node->left = leftRotate(node->left);
    return rightRotate(node);
}

// Right Left Case
if (balance < -1 && key < node->right->key){
    node->right = rightRotate(node->right);
    return leftRotate(node);
}

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

```
}

void InOrder(Node *root){
    if(root != NULL){
        InOrder(root->left);
        cout << root->key << " ";
        InOrder(root->right);
    }
}

int main(){
    Node *root = NULL;

    root = insert(root, 12);
    root = insert(root, 1);
    root = insert(root, 4);
    root = insert(root, 3);
    root = insert(root, 7);
    root = insert(root, 8);
    root = insert(root, 10);
    root = insert(root, 2);
    root = insert(root, 11);
    root = insert(root, 5);
    root = insert(root, 6);

    cout << "Inorder traversal of the constructed AVL tree is: \n";
    InOrder(root);
}
```

10.


Suppose the order(m) = 7
8,9,10,12,13,15
17
add 17
12
8.9.10
13.15.17
add 4.5.6
12
4.5.6.8.9.10
13.15.17
add 1
6.12
1.4.5
8.9.10
13,15,17
add 3
6,12
1,3,4,5
8.9.10
13.15.17