Traversals for tree:

1. IN-Order

a) Recursion

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

b) Stack(Iterative)

```
void inOrder(Node *root)
{
    Node* curr = root;
    stack<Node*> st;

    while(curr!=NULL || !st.empty())
    {
        while(curr!=NULL)
        {
            st.push(curr);
            curr=curr->left;
        }
        curr = st.top();
        st.pop();
        cout<<curr->data<<" ";
        curr=curr->right;
    }
}
```

2. Pre-Order

a. Recursion

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

b. Stack(Iterative)

```
void preOrder(Node *root)
{
     Node* curr = root;
     stack<Node*> st;

     while(curr!=NULL || !st.empty())
     {
         while(curr!=NULL)
         {
                cout<<curr->data<<" ";
                st.push(curr);
                curr=curr->left;
            }
            curr = st.top();
            st.pop();
            curr=curr->right;
        }
   }
```

3. Post-Order

a. Recursion

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

b. Stack(Iterative)

1.1 Without vector

```
void postOrder(Node *root){
    if(root==NULL) return;
    stack<Node*> st;
    while(1){
        while(root!=NULL){
            st.push(root);
            st.push(root);
            root=root->left;
        }
        if(st.empty())
            break;
        root=st.top();
        st.pop();
        if(!st.empty() && st.top()==root)
             root=root->right;
        else{
               cout<<root->data<<" ";
               root=NULL;
        }
    }
  }
```

1.2 With vector

```
void postOrder(Node *root)
{
        if(root==NULL) return;
        vector<int> ans;

        stack<Node*> s;
        s.push(root);

        while(s.size() ) {
              Node* prev = s.top();
              ans.push_back(s.top()->data);
              s.pop();
              if(prev->left) {
                  s.push(prev->left);
              }
              if(prev->right) {
                  s.push(prev->right);
              }
          }
          reverse(ans.begin(), ans.end());
          for(int i : ans)
              cout<<i<<" ";
  }
```

Inorder: ***Left - Root - Right***

Preorder: ***Root - Left - Right***

Postorder: ***Left - Right - Root***

Top View:

```
void topView(Node * root)
{
    Node* temp=root;
    map<int,int> m;
    vector<Node*> v;
    vector<int> vi;
    v.push_back(temp);
    vi.push_back(0);
    while(!v.empty()) {
            Node* x = v[0];
            v.erase(v.begin());
            int d = vi[0];
            vi.erase(vi.begin());

            if(m.find(d)!=m.end())   {}
            else
                m[d]=x->data;
            if(x->left!=NULL)
            {
                v.push_back(x->left);
                vi.push_back(d-1);
            }
            if(x->right!=NULL)
            {
                v.push_back(x->right);
                vi.push_back(d+1);
            }
    }
        map<int,int>:: iterator itr;
        for(itr=m.begin();itr!=m.end();itr++)
        {
            cout<<itr->second<<" ";
        }
        cout<<endl;
   }
```

Level Order

My Approach

```
vector<TreeNode* > q1;
vector<TreeNode* > q2;
q1.push_back(A);

while(!q1.empty() || !q2.empty())
{
        if((!q1.empty()))
        {
            vector<int> v;
               for(int i=0;i<q1.size();i++)
               {
                   v.push_back(q1[i]->val);
               }
               ans.push_back(v);
           }
           while(!q1.empty())
           {
               TreeNode* node = q1[0];
               q1.erase(q1.begin());
               if(node->left)
                   q2.push_back(node->left);
               if(node->right)
                   q2.push_back(node->right);
           }
           if(!q2.empty())
           {
               vector<int> v;
               for(int i=0;i<q2.size();i++)
               {
                   v.push_back(q2[i]->val);
               }
               ans.push_back(v);
           }

```

```
            while(!q2.empty())
            {
                TreeNode* node = q2[0];
                q2.erase(q2.begin());
                if(node->left)
                    q1.push_back(node->left);
                if(node->right)
                    q1.push_back(node->right);
            }

    }
```

Abhinav's Approach

```
queue<TreeNode* > q;
q.push(A);

while(q.size()!=0)
{
    vector<int> v;
    int s=q.size();
    while(s--)
    {
            TreeNode* node=q.front();
            q.pop();
            v.push_back(node->val);

            if(node->left)
                q.push(node->left);

            if(node->right)
                q.push(node->right);
        }
        ans.push_back(v);
    }
```

Shrayans's Approach

```
void buildVector(TreeNode *root, int depth, vector<vector<int> > &ret)
{
        if(root == NULL) return;
        if(ret.size() == depth)
            ret.push_back(vector<int>());

        ret[depth].push_back(root->val);
        buildVector(root->left, depth + 1, ret);
        buildVector(root->right, depth + 1, ret);
}

vector<vector<int> > levelOrder(TreeNode *root)
{
        vector<vector<int> > ret;
        buildVector(root, 0, ret);
        return ret;
}
```

<u>BFS(Breadth First Search)</u> : QUEUE (Iterative)

<u>DFS(Depth First Search)</u> : STACK (Recursion)

Sieve Of Eratosthenes

Time Complexity: *O(n*log(log(n)))*

```
bool* SieveOfEratosthenes(int n)
{
    bool prime[n+1];
    memset(prime, true, sizeof(prime));
    for (int p=2; p*p<=n; p++)
    {
        if (prime[p] == true)
        {
            for (int i=p*p; i<=n; i += p)
                  prime[i] = false;
          }
      }
      return prime;
  }
```

Least Prime Divisor

```
void least_prime_divisor()
{
    int prime[1000009];
    memset(prime,0,sizeof(prime));
    prime[0]=prime[1]=1;
    for(int i=2;i<=1000;i++)
    {
        if(prime[i]==0)
        {
              for(int j=2*i;j<1000009;j+=i)
              {
                  if(prime[j]==0)
                      prime[j]=i;
              }
           }
       }
  }
```

Fast Power

1.Iterative

```
#define ll long long
ll fastpow(ll base, ll exp)
{
    ll res=1;
    while(exp>0)
    {
       if(exp%2==1)
            res=res*base;
       base=base*base;
          exp/=2;
     }
       return res;
  }
```

2. Recursive

```
ll fastpow(ll b,ll e)
{
    if(e==0)
        return 1;
    if(e==1)
        return b;

    ll temp = fastpow(b,e/2);
    if(e%2==0)
          return temp*temp;
     else
          return b*temp*temp;
  }
```

Fenwick Tree

1-based Indexing

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

class FenwickTree  {
   public:
    vector<int> tree;
    int n;
       FenwickTree(int n)
       {
           this->n = n+1;
           tree.assign(n+1,0);
      }
      void update(int idx, int delta)
      {
          idx++;
          // int delta=tree[idx]-value;
          for(;idx<n; idx += idx & -idx)
              tree[idx]+=delta;
      }
      void build(vector<int> &a)
      {
          for(int i=0;i<a.size();i++)
              update(i,a[i]);
      }

      int sum(int idx)
      {
          int ret=0;
          ++idx;

          for(;idx>0; idx -= idx & -idx)
              ret+=tree[idx];
          return ret;
      }
```

```

    int query(int l, int r)
    {
        return sum(r)-sum(l-1);
    }

    void print()
    {
        cout<<"\n\nTree : ";
        for(int i=1;i<n;i++)
        {
            cout<<tree[i]<<" ";
        }
        cout<<"\n\n";
    }

};

int main()
{
    int n;
    cin>>n;
    vector<int> a(n);
    for(int i=0;i<n;i++)
        cin>>a[i];
    FenwickTree f(n);
    f.build(a);
    f.print();
}
```

0-based Indexing

```
struct FenwickTree {
    vector<int> bit;  // binary indexed tree
    int n;

    FenwickTree(int n) {
        this->n = n;
        bit.assign(n, 0);
    }

    FenwickTree(vector<int> a) : FenwickTree(a.size()) {
        for (size_t i = 0; i < a.size(); i++)
            add(i, a[i]);
    }

    int sum(int r) {
        int ret = 0;
        for (; r >= 0; r = (r & (r + 1)) - 1)
            ret += bit[r];
        return ret;
    }

    int sum(int l, int r) {
        return sum(r) - sum(l - 1);
    }

    void add(int idx, int delta) {
        for (; idx < n; idx = idx | (idx + 1))
            bit[idx] += delta;
    }
};
```

Segment Tree

```
#define MAX 400005
int tree[MAX];

void update(int* a, node, start, end, index, value)
  {
    if(start == end)  {
        a[index]=value;
        tree[node]=value;
        return;
    }
    int mid = (start+end)/2;

      if(start<=index && index<=mid)
          update(a,2*node,start,mid,index,value);
      else
          update(a,2*node+1,mid+1,end,index,value);

      tree[node]=tree[2*node] + tree[2*node+1];
  }

 int query(node, start, end, left, right)
  {
      if( right < start || end < left)
          return 0;

      if( left <= start && right >= end)
          return tree[node];

      int mid = (start + end)/2;
      int q_left = query(2*node,start, mid ,left,
min(right, mid));
      int q_right = query(2*node+1, mid+1, end,
max(left,mid+1),right);
      return q_left + q_right;
  }
```

```
void build(int* a,int node,int start, int end)
{
    if(end < start)
        return;
    if(start == end)  {
        tree[node]=a[start];
        return;
    }

    int mid = (start+end)/2;
    build(a,2*node,start,mid);
    build(a,2*node+1,mid+1,end);

    tree[node] = tree[2*node] + tree[2*node +1];
}

void print_tree(int node,int start,int end,int space)
{
    if(tree[node]==-1)
        return;

    int mid=(start+end)/2;
    print_tree(2*node+1,mid+1,end,space+10);
    for(int i=0;i<space;i++)
        cout<<" ";
    cout<<tree[node]<<"["<<start<<":"<<end<<"]\n";
    print_tree(2*node,start,mid,space+10);
}

int main()
{
  memset(tree,-1,MAX);
  //solve
}
```

Lazy Propagation

Trie

```
typedef class TrieNode
{
    public:
        TrieNode* characters[27];
        int end;
        TrieNode()  {
            for(int i=0;i<27;i++)
                characters[i]=NULL;
            end=0;
        }
  }Node;

  Node* root=new Node();
  vector<string> allprefixes;

  void insert(Node* root,string s)
  {
      Node* temp = root;
      for(int i=0;i<s.length();i++)
      {
          int t = s[i]-97;
          if(temp->characters[t]!=NULL)
          {
              temp=temp->characters[t];
          }
          else
          {
              temp->characters[t]=new Node();
              temp=temp->characters[t];
          }
      }
      temp->end=1;
  }
```

```

int search(Node* root,string s)
{
    Node* temp=root;
    for(int i=0;i<s.length();i++)
    {
        int t = s[i]-97;
        if(temp->characters[t]!=NULL)
        {
            temp=temp->characters[t];
        }
        else
        {
            break;
        }
    }
    if(temp->end==1)
        return 1;
    return 0;
}

void newprefix(Node* temp,string prefix)
{
    if(temp->end==1)
    {
        allprefixes.push_back(prefix);
    }

    for(int i=0;i<27;i++)
    {
        if(temp->characters[i]!=NULL)
        {
            char c = i+97;
            string np = prefix+c;
            newprefix(temp->characters[i],np);
        }
    }
}
```

```

vector<string> prefixes(Node* root,string prefix)
{
    Node* temp=root;
    allprefixes.clear();
    for(int i=0;i<prefix.length();i++)
    {
        int t = prefix[i]-97;
        if(temp->characters[t]!=NULL)
        {
            temp=temp->characters[t];
        }
        else
        {
            return allprefixes;
        }
    }

    newprefix(temp,prefix);
    return allprefixes;
}
```

Disjoint Set Union

```
n = N
p=[]
rank=[]

def make_set(x):
    p[x] = x
    rank[x] = 0

def find(x):
    if(x != p[x]):
        p[x] = find(p[x])
    return p[x]

def link(x, y):
    if(rank[x] > rank[y]):
        swap(x,y)

    if(rank[x] == rank[y]):
        rank[y] = rank[y] + 1

    p[x] = y
    return y

def union(x, y):
    link(find(x),find(y))

'''''
    In our analysis, we show that any sequence of m UNION
and FIND operations on n elements take at most
    O((m + n) log * n) steps, where log * n is
    the number of times you must iterate the log 2
function on n before getting a number less than or
equal to 1.
    (So log * 4 = 2, log * 16 = 3, log * 65536 = 4.)
'''
```

SCC