1. Graph Search Algorithm:

For some data structure like linkedlist, each node is only connected one other node, and the connection is in single direction. This allows us the traverse the linkedlist iteratively by keep visiting the next node of a given node till the end of the list. In graph, each node is connected with multiple nodes, for a given node, we need a metric to determine which node to go next. There are two ways to do it Breadth First Search(BFS) and Deep First Search(DFS).

1.1 BFS

a. Intuition

(1) For any node, we would like to visit all its adjacent nodes before we go deeper.

(2) For any node, there are three states.

State1 Unvisited, it means we never reach it. For example, at the beginning of the search, all nodes are unvisited.

State2 Visited, but not each of its adjacent nodes is visited. For example, we start from node0, and node0 has adjacent nodes named node1 and node2. Once we visit node 0, node0 becomes visited, but its adjacent node1 and node2 are unvisited.

State3 Visited, and all of its adjacent nodes are visited.

(3) For all the nodes with state3, we are done

For all the nodes with state2, we need to keep track of it using certain data structure. That means given a node, we need to store all its adjacent nodes which are state2. Then we come back to visit those stored first. Using a queue can meet our needs.

(4) All the nodes with state2 are store in a queue.

(5) For cyclic graph, we can come back to the same node we start with, so also need a data structure to keep track of all the visited nodes. For acyclic graph and if we start a node that connects to all the other node, we do no need additional data structure to store all visited nodes.

b. Algorithm and Implementation

```
class Graph
{
    private int V;   // No. of vertices
    private LinkedList<Integer> adj[]; //Adjacency Lists

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
```

```
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v,int w)
    {
        adj[v].add(w);
    }

    // prints BFS traversal from a given source s
    void BFS(int s)
    {
        // Mark all the vertices as not visited(By default
        // set as false)
        boolean visited[] = new boolean[V];

        // Create a queue for BFS
        LinkedList<Integer> queue = new LinkedList<Integer>();

        // Mark the current node as visited and enqueue it
        visited[s]=true;
        queue.add(s);

        while (queue.size() != 0)
        {
            // Dequeue a vertex from queue and print it
            s = queue.poll();
            System.out.print(s+" ");

            // Get all adjacent vertices of the dequeued vertex s
            // If a adjacent has not been visited, then mark it
            // visited and enqueue it
            Iterator<Integer> i = adj[s].listIterator();
            while (i.hasNext())
            {
                int n = i.next();
                if (!visited[n])
                {
                    visited[n] = true;
                    queue.add(n);
                }
            }
        }
    }
}
```

c. Time complexity O(V + E)

(1) Loop element in the queue

we keep adding and poping element in and out of the queue. As we check whether the element has been visited or not before we enqueue it, each element is enqueued only once. Therefore the time complexity for the queue operation is O(V)

(2) Loop for adjacent elements of each node pop out of the queue. The total number of iterations are 2 x number of edges. The factor 2 comes from duplicated counting.

So, the total time complexity is O(V+E).

1.2 DFS

a. Intuition

(1) In DFS, we explore the graph as deep as possible.

(2) This means for each node, as long as there is an adjacent unvisited node, we would visited that node, and keep doing it until we reach the node where all the adjacent nodes have been visited.

(3) Then we go backwards to the last but two node, keep visiting another unvisited adjacent node. Repeating in (2).

(4) For any node, there are three states. Same as BFS.

b. Implementation

(1) From the intuition above, for each node we keep exploring the edges as deep as possible. Once reach the deepest one, we move back to the last but two node, then exploring the unvisited adjacent node as deep as possible. A natural structure to solve go in and go out pattern is recursion.

```
class Graph
{
    private int V;   // No. of vertices

    // Array  of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];

    // Constructor
    Graph(int v)
    {
        V = v;
        adj = new LinkedList[v];
        for (int i=0; i<v; ++i)
            adj[i] = new LinkedList();
    }
```

```
    //Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        adj[v].add(w);  // Add w to v's list.
    }

    // A function used by DFS
    void DFSUtil(int v,boolean visited[])
    {
        // Mark the current node as visited and print it
        visited[v] = true;
        System.out.print(v+" ");

        // Recur for all the vertices adjacent to this vertex
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext())
        {
            int n = i.next();
            if (!visited[n])
                DFSUtil(n, visited);
        }
    }

    // The function to do DFS traversal. It uses recursive DFSUtil()
    void DFS(int v)
    {
        // Mark all the vertices as not visited(set as
        // false by default in java)
        boolean visited[] = new boolean[V];

        // Call the recursive helper function to print DFS traversal
        DFSUtil(v, visited);
    }
}
```

(2) The above recursion call using a recursion call stack, we can define a explicit stack to do the search.

```
static class Graph
    {
        int V; //Number of Vertices

        LinkedList<Integer>[] adj; // adjacency lists

        //Constructor
        Graph(int V)
        {
```

```
    this.V = V;
    adj = new LinkedList[V];

    for (int i = 0; i < adj.length; i++)
        adj[i] = new LinkedList<Integer>();

}

//To add an edge to graph
void addEdge(int v, int w)
{
    adj[v].add(w); // Add w to v's list.
}

// prints all not yet visited vertices reachable from s
void DFSUtil(int s, Vector<Boolean> visited)
{
    // Create a stack for DFS
    Stack<Integer> stack = new Stack<>();

    // Push the current source node
    stack.push(s);

    while(stack.empty() == false)
    {
        // Pop a vertex from stack and print it
        s = stack.peek();
        stack.pop();

        // Stack may contain same vertex twice. So
        // we need to print the popped item only
        // if it is not visited.
        if(visited.get(s) == false)
        {
            System.out.print(s + " ");
            visited.set(s, true);
        }

        // Get all adjacent vertices of the popped vertex s
        // If a adjacent has not been visited, then push it
        // to the stack.
        Iterator<Integer> itr = adj[s].iterator();

        while (itr.hasNext())
        {
            int v = itr.next();
            if(!visited.get(v))
                stack.push(v);
        }
```

```
            }
        }

        // prints all vertices in DFS manner
        void DFS()
        {
            Vector<Boolean> visited = new Vector<Boolean>(V);
            // Mark all the vertices as not visited
            for (int i = 0; i < V; i++)
                visited.add(false);

            for (int i = 0; i < V; i++)
                if (!visited.get(i))
                    DFSUtil(i, visited);
        }
    }
```

c.Time complexity O(V+E)
For the iterative version, analysis is similar to BFS. The size of the stack is O(V) for each node, we loop the adjacent nodes, the total adjacent nodes is O(E), Therefore, the time complexity O(V+E).
For the recursive version, the DFS recursion is called V times, and in each DFS, we loop the adjacent nodes of the current nodes, the total number of iteration is O(E). Therefore, it is O(V+E).

1.3. Topological sorting
For a direct acyclic graph, we can use it to represent the order of a sequence of events. Topological sort transfers a graph to a linear array/list while keep the order the sequence.
a. Why topological?
Topology is a subject in math which studies the properties of an geometry object under continuous deformations. For example, the donut and the cup has one hole and one can continuously deform a donut to a cup. The topological sort transfer a graph to linear list without changing its topology property. Topological sort requires a graph has no cycle, so that we can transform it to a list which does not contain any cycle either. And the acyclic graph and the list have the same topological property.
b. Implementation

```
class Graph
{
    private int V;   // No. of vertices
    private LinkedList<Integer> adj[]; // Adjacency List
```

```
//Constructor
Graph(int v)
{
    V = v;
    adj = new LinkedList[v];
    for (int i=0; i<v; ++i)
        adj[i] = new LinkedList();
}

// Function to add an edge into the graph
void addEdge(int v,int w) { adj[v].add(w); }

// A recursive function used by topologicalSort
void topologicalSortUtil(int v, boolean visited[],
                         Stack stack)
{
    // Mark the current node as visited.
    visited[v] = true;
    Integer i;

    // Recur for all the vertices adjacent to this
    // vertex
    Iterator<Integer> it = adj[v].iterator();
    while (it.hasNext())
    {
        i = it.next();
        if (!visited[i])
            topologicalSortUtil(i, visited, stack);
    }

    // Push current vertex to stack which stores result
    stack.push(new Integer(v));
}

// The function to do Topological Sort. It uses
// recursive topologicalSortUtil()
void topologicalSort()
{
    Stack stack = new Stack();

    // Mark all the vertices as not visited
    boolean visited[] = new boolean[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // Call the recursive helper function to store
    // Topological Sort starting from all vertices
    // one by one
```

```
        for (int i = 0; i < V; i++)
            if (visited[i] == false)
                topologicalSortUtil(i, visited, stack);

        // Print contents of stack
        while (stack.empty()==false)
            System.out.print(stack.pop() + " ");
    }
```

2. Binary Tree traversal algorithm

2.1 Recursively traverse a tree (same as DFS)

a. pseudocode

```
traverse(Node node)
{
   if (node == null) return
   traverse(node.left);
   traverse(node.right);
}
```

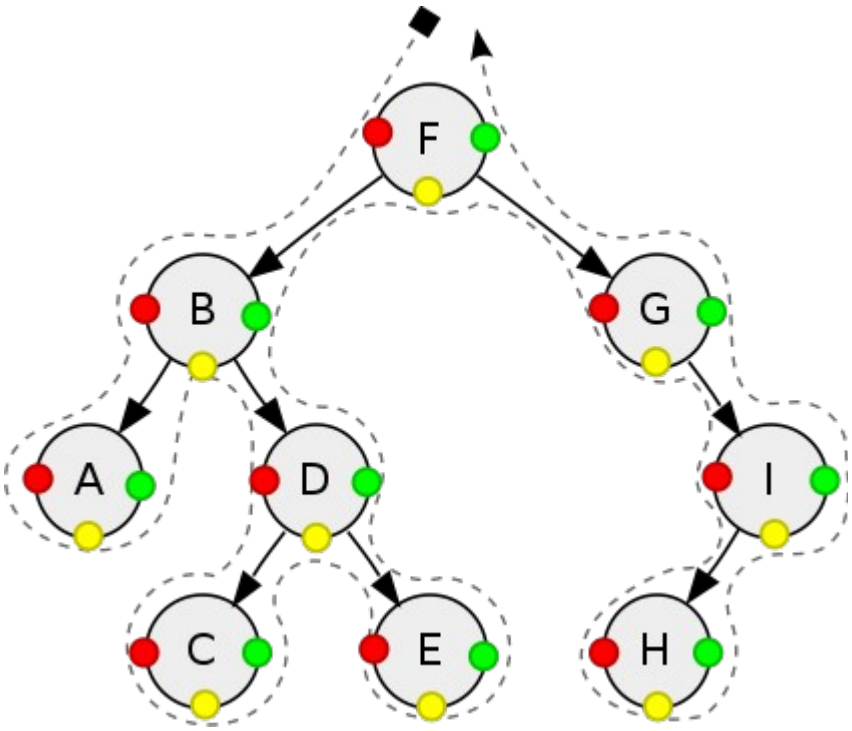
For each node, the traverse function is entered three times

1) before traverse its left node
2) after traverse its left node & before traverse the right node
3) after traverse its right node

b. visiting trace is a loop from the root node to the root node

for example the tree below, the visiting trace is

F B A B D C D E D B F G I H I G F



we see each node is touched 3 times. If we want to convert the tree to a linear data structure like array or list such that each node is picked only once, depending on how we pick up the node, we can end up 3 different ways of traversal

(1) If we each elements for the first time of its appearance, such an order is called preorder traversal，which is F B A D C E G I H. This is also a topological sort of the tree, as for each node, all of its children come after itself.

(2) If we each elements for the first time of its appearance, such an order is called inorder traversal, which is A B C D E F G H I

(3) If we each elements for the first time of its appearance, such an order is called postorder traversal, which is A C E D B H I G F

c. Complexity

(1) Time complexity. O(N)

This is because one has to visit each node of the tree.

(2) Space complexity

Since it is a recursion, the space complexity is the depth of recursion. For balanced tree it is O(logN), for unbalanced, the worst case is O(N).

d. Build a tree from its traversal result

following combination can uniquely identify a tree.

Inorder and Preorder.

Inorder and Postorder.

Inorder and Level-order.

And following do not.

Postorder and Preorder.

Preorder and Level-order.

Postorder and Level-order.

2.1 Iteratively traverse a tree.

Iterative traversal can be used as level order traversal(it is the same as BFS)

3. Binary Search Tree

3.1 Property

(1) left< node < right

(2) its in order traversal is a sorted array.