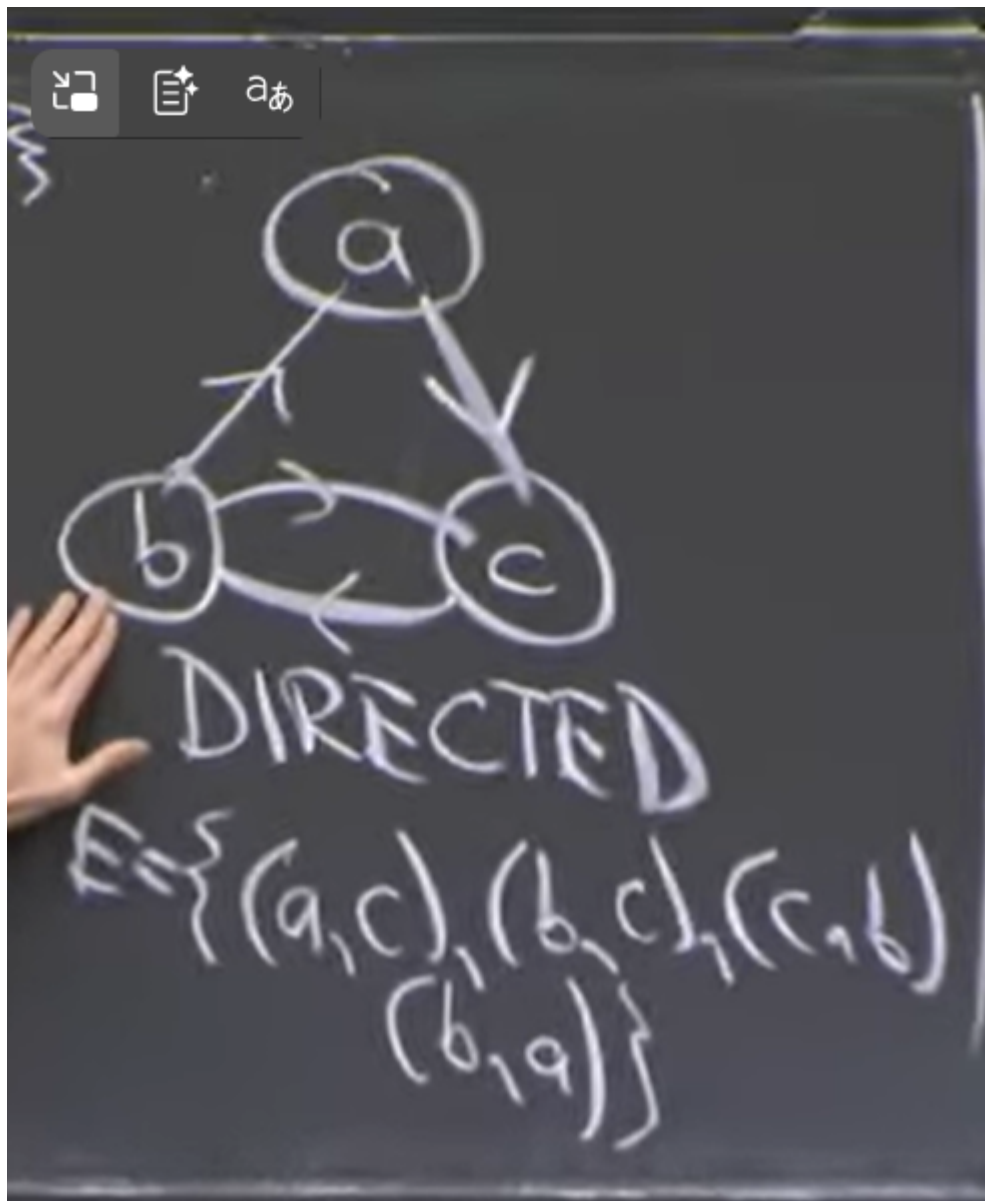
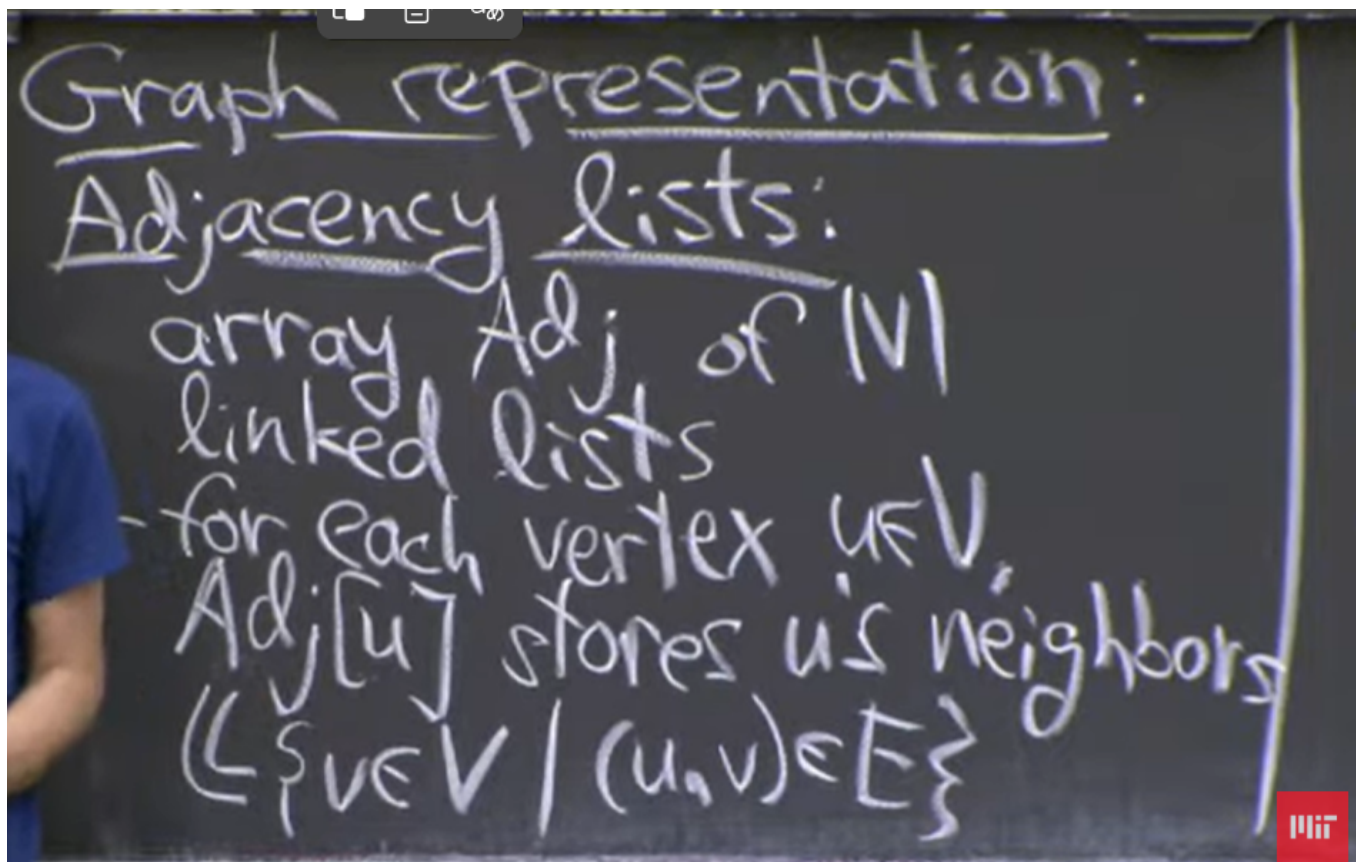


Breadth First Search



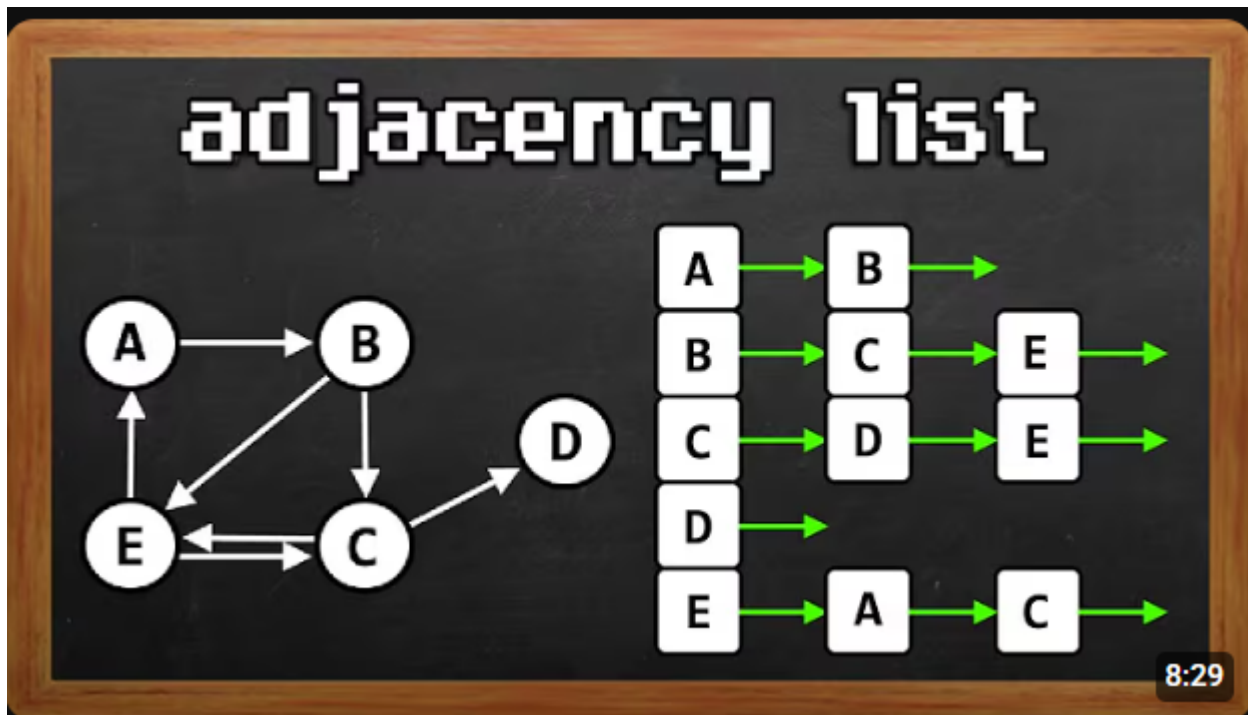


The adjacency list tells us when you are at a specific node, what are the places you can go to, so it is stored in the form of a linked list

Adj[b] = {a, c}
 Adj[a] = {c}
 Adj[c] = {b}

Adj:

b	→	a	→	c
a	→	c	→	
c	→	b	→	



implementation of the directed graph in java

```
1
2 public class graph
3 {
4     int[][] matrix;
5     graph(int size){
6         matrix=new int[size][size];
7     }
8     public static void addNode(Node node){
9
10    }
11    public void addEdge(int src, int dst){
12        matrix[src][dst]=1;
13    }
14    public boolean checkEdge(int src, int dst){
15        if(matrix[src][dst]==1){
16            return true;
17        }
18        else{
19            return false;
20        }
21    }
22    public void print(){
23        for(int i=0;i<matrix.length;i++){
24            for(int j=0;j< matrix[j].length;j++){
25                System.out.print(matrix[i][j]+" ");
26            }
27            System.out.println();
28        }
29    }
30 }
```

```
STOP import java.util.*;
2
3 public class Main{
4     public static void main(String[] args){
5         graph graph1=new graph(5);
6         graph1.addNode(new Node('A')); //index:0
7         graph1.addNode(new Node('B')); //index:1
8         graph1.addNode(new Node('C')); //index:2
9         graph1.addNode(new Node('D')); //index:3
10        graph1.addNode(new Node('E')); //index:4
11
12        graph1.addEdge(0,1);
13        graph1.addEdge(1,2);
14        graph1.addEdge(2,3);
15        graph1.addEdge(2,4);
16        graph1.addEdge(4,2);
17        graph1.addEdge(4,0);
18    }
19 }
20
```

implementing undirected graph(cpp)

Adjacency List (store neighbors for $\forall v$)

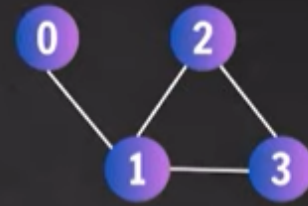
`list<int> [V]`

$0 \rightarrow [1] \rightarrow \text{list<int>}$

$1 \rightarrow [0, 2, 3] \rightarrow \text{list<int>}$

$2 \rightarrow [1, 3] \rightarrow \text{list<int>}$

$3 \rightarrow [1, 2] \rightarrow \text{list<int>}$



$V=4$

```
#include <iostream>
#include <vector>
#include <list>
using namespace std;

class Graph{
    int V;
    list<int> *l; // list *arr
public:
    Graph(int V){
        this->V=V;
        l=new list<int> [V];
    };
    void addEdge(int u,int v){ //unidrected edge
        l[u].push_back(v);
        l[v].push_back(u);
    }
    void printAdjList(){
        for(int i=0;i<=V;i++){
```

```

        cout<<i<<" : ";
        for(int neigh: l[i]){
            cout<<neigh<<" ";
        }
        cout<<endl;
    }
}

};

int main(){
    Graph g(5);

    g.addEdge(0,1);
    g.addEdge( 1,2);
    g.addEdge(1,3);
    g.addEdge(2,4);

    g.printAdjList();
    return 0;
}

```

in the above code what we are saying is:

```
list<int> *l=new list<int> [V];
```

Some applications of breadth first search

1. Web crawling
2. social networking
3. network broadcast
4. garbage collection
5. Checking the model of your chip
6. Checking mathematical conjectures

Model checking is-- you have some finite model of either a piece of code, or a circuit, or chip, whatever, and you want to prove that it actually does what you think it does.

And so you've drawn a graph.

The graph is all the possible states that your circuit or your computer program could reach, or that it could possibly have.

You start in some initial state, and you want to know among all the states that you can reach, does it have some property.

And so you need to visit all the vertices that are reachable from a particular place.

And usually people do that using breadth-first search.

I use breadth-first search a lot, myself, to check mathematical conjectures.

So if you're a mathematician, and you think something is true.

Like maybe-- It's hard to give an example of that.

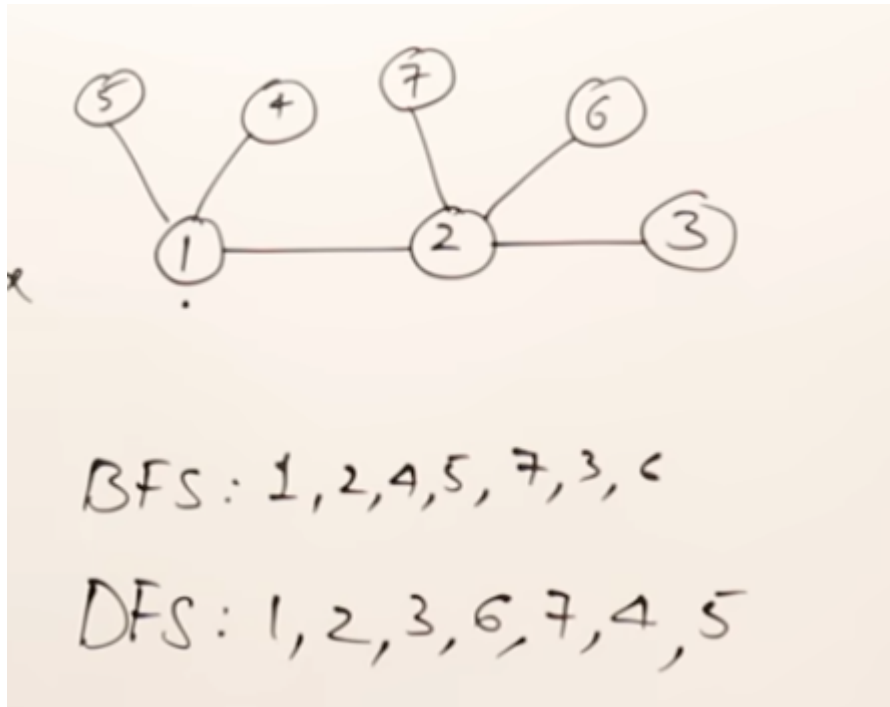
But you can imagine some graph of all the possible inputs to that theorem, and you need to check them for every possible input-- If this is true-- the typical way to do that is breadth-first searching through that entire graph of states.

Usually, we're testing finite, special cases of a general conjecture, but if we find a counter-example, we're done.

Don't have to work on it anymore.

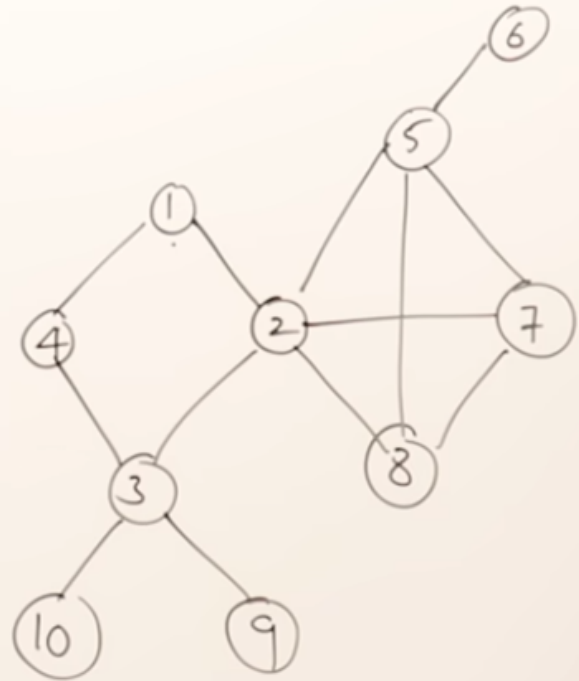
If we don't find a counter-example, usually then we have to do the mathematics.

It doesn't solve everything, but it's helpful.



BFS & DFS

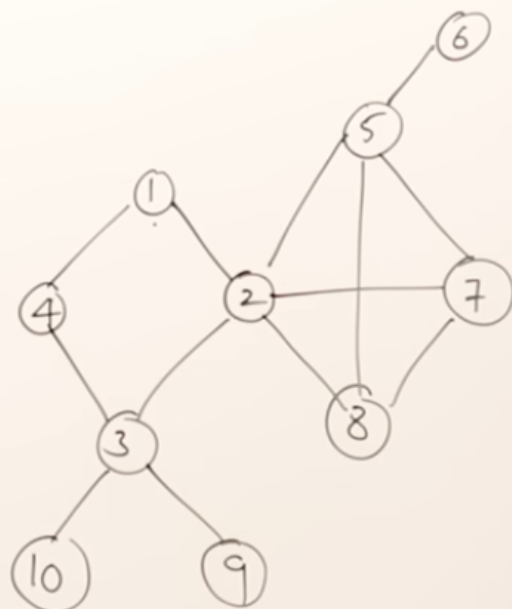
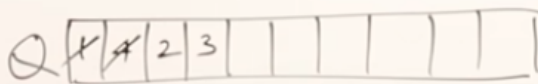
BFS: 1, 4, 2



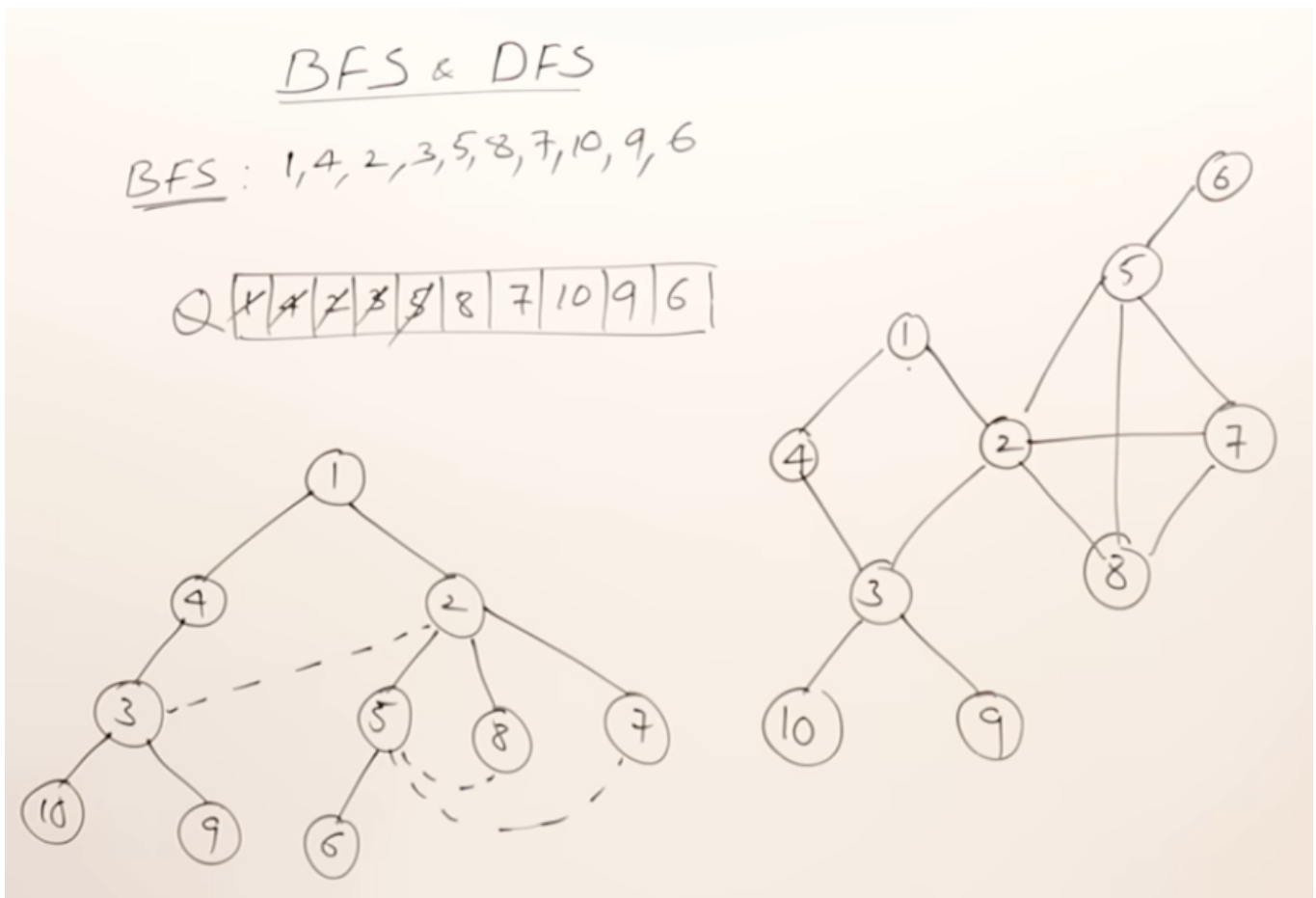
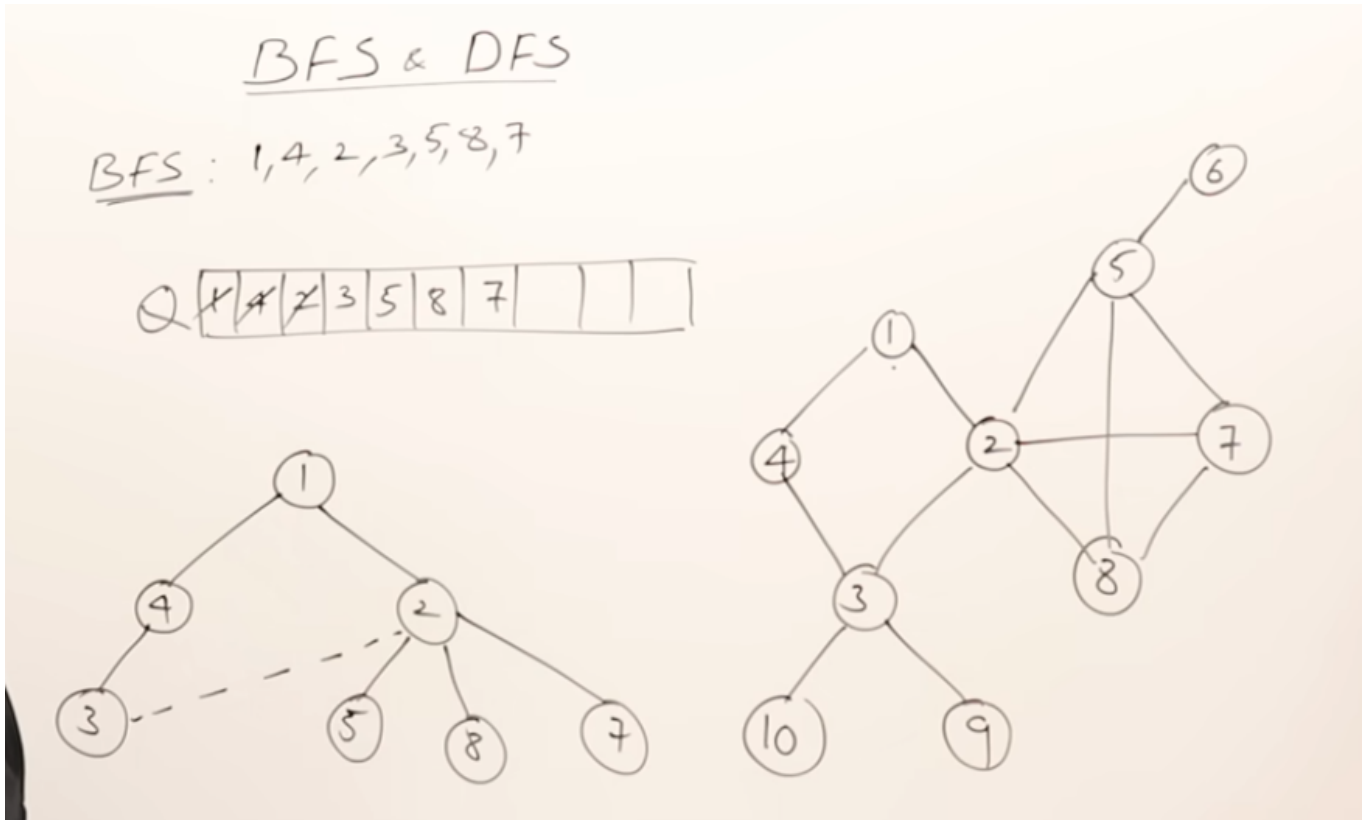
4 is the first node we have pushed on to the queue, we will now explore from node 4 i.e. see if there are any nodes connected to node 4, if there are push that to the queue (node 3 is pushed), and then pop node 4 off the queue

BFS & DFS

BFS: 1, 4, 2, 3



We first push the visited nodes onto the queue, we start from node 1, then push node 1 to the queue, next we push node 4 and node 2 to the queue, notice that



Graph data structure interesting points

The relative advantage of graph retrieval grows with the complexity of a query. For example, one might want to know "that movie about submarines with the actor who was in that movie with that other actor that played the lead in *Gone With the Wind*". *This first requires the system to find the actors in _Gone With the Wind*, find all the movies they were in, find all the actors in all of those movies who were not the lead in *Gone With the Wind*, and then find all of the movies they were in, finally filtering that list to those with descriptions containing "submarine". In a relational database, this would require several separate searches through the movies and actors tables, doing another search on submarine movies, finding all the actors in those movies, and then comparing the (large) collected results. In contrast, the graph database would walk from *Gone With the Wind* to Clark Gable), gather the links to the movies he has been in, gather the links out of those movies to other actors, and then follow the links out of those actors back to the list of movies. The resulting list of movies can then be searched for "submarine". All of this can be done via one search.

Implementation of BFS

The goal is to represent the graph in abdul bari's notes for better understanding

Based on his words

1. First create a boolean array of size 10, initialize all the elements to 0
2. Create a 10*10 adjacency matrix
3. Initialize the graph, i.e. fill the matrix
4. push the elements onto the queue
5. After looping through the whole adjacency matrix,