8

$G = (\{A, B, C\}, \{\ \})$
$V \to 3$
$E \to 0$

disconnected graph

connected component

graph → undirected
→ dis connected

If a a subgraph of a graph is which forms a tree

Is this a connected graph??

$A \to C$
$C \to A$
$A \to B$
$B \to A$??

Graph representations

① Adjacency matrix

n no nodes
n × n space

	A	B	C
A	0	1	0
B	1		1
C	0	1	0

symmetrical matrix

$A \to B$

For weighted graph

	A	B	C
A	0	1	0
B	1		1
C	0	1	0

$A \to B$

pt weight value indicated by 0 or 1.

Pros → easy to implement

Cons → space.

n nodes
n^2 space

For sparse graph → 2n nodes & 3 edges

② Adj array list.

In each node, there is a list of its immediate neighbours.

Hashing

<key, value>
↓
node, neighbor < edge >
value

0 → [ ]
1 → [3, 2]
2 → [1, 3]
3 → [1, 2]

<key, value>
"in hashtable < int >
class Node {
int edge;
int weight
}

Pros : time space

8. Given a graph, tell one random of neighbour for a node??

(Q6)

8. Given two nodes, check if they are directly connected??

(matrix)

Graph traversals

Tree do not have cycles

0 → [ ] → 1 → 0 . . .
(infinite loop?)

visited ?

Tree traversal techniques

① Depth First search

(Pre order traversal)

visited

0	1	2	3	4	5	6
T	T	T	T	T	T	T

	0	1	2	3	4	5	6
0	0	1	0	1	0	0	0
1	1	0	1	0	1	0	0
2	0	1	0	1	0	1	0
3	1	0	1	0	1	0	1
4	0	1	0	1	0	1	0
5	0	0	1	0	1	0	1
6	0	0	0	1	0	1	0

TC → O(V + E)

Recursive Implementation

```
bool visit[n];
void DFS (int i) {
    print i;
    for int j in adjlist[i]
        if ( ! visit[j] ) {
            visit[j] = 1;
            DFS(j);
        }
    }
}
main() {
    visit[0] = 1;
    DFS(0);
    for i (range(n))
        if visit[i] = 0:
            print i
    }
```

visiting nodes

Iterative Implementation

visiting stack

8. Given an undirected graph, tell whether the graph is connected or not??

0	1	2	3	4	5	6
T	T	T	T	T	T	T

① Take any random node
② Perform DFS from there
③ check visited [ ] of any node to find, then graph is disconnected

Breadth First traversal

(level vs levels)

1. Append any node as root (L0)
2. Nodes adjacent to root, random then visitated as L1.
3. adj to adj → L2.

L0 → 0
L1 → 1, 2, 3
L2 → 4, 5, 6
L3 → 7

BFS → 0, 1, 2, 3, 4, 5, 6, 7

L0 → 0 → 1 edge only from root node

also BFS

0, 2, 1, 3, 4, 5, 6, 7

TC → O(V + E)

```
pseudocode :
    visit = [ ]
    Q := queue.
    q.enqueue(s);
    mark s as visited visit[s] = 1;
    while( Q is not empty)
        v = q.dequeue();
        if(probably all neighbours of v
            if v is not visited
                q.enqueue(v);
                mark v as visited;
        }
    }
```

BFS & can be used to find the shortest path b/w two nodes in an unweighted graph.

Recursion :

```
void BFS(q, visit) {
    while ( q != empty())
        int v = q.dequeue();
        print v;
        for int u = adjlist[v]
            if ( !visit[u])
                visit[u] = 1;
                q.add(u);
        }
    }
    BFS (q, visit);
}
```

Graph Implementation

directed / undirected
weighted

Hashmap < String, Vertices >

neighbours
Hashmap < String, Integer >

add vertex (A)

string value>