YASH PRADHAN

Traversals

- **Graphs:**

The graph data structures consists of nodes and edges. nodes are vertices and edges are the connecting links/arcs between pair of vertices.

- **Tree:**

Tree is a special type of graph, a minimally connected graph.

- **Binary Tree:**

Each nodes can have at max 2 children. The root has no parent, and the nodes having no children are called leaf nodes.

- **Breadth First Search:**

In BFS, we traverse the data structure level by level. i.e. breadth first manner.

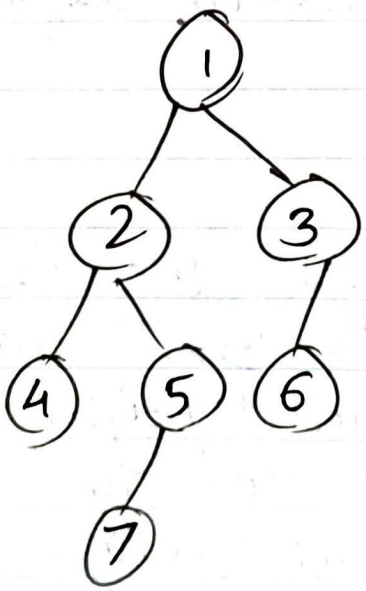
In other words, all neighbours of a node u will be visited and then their respective children (children of neighbours) would be visited. We use the Queue data structure for BFS.

```
class Node{
    int val
    Node left, right
}
```

- Depth First Search:

As the name suggest, in DFS we traverse in depth first manner.
DFS uses stack ie recursion.
In this when we are visiting a neighbour v of node u, we would recursively invoke DFS on v, later when recursion unwinds for DFS(v), the other neighbours of u would be visited.

Tree - DFS



Algorithm

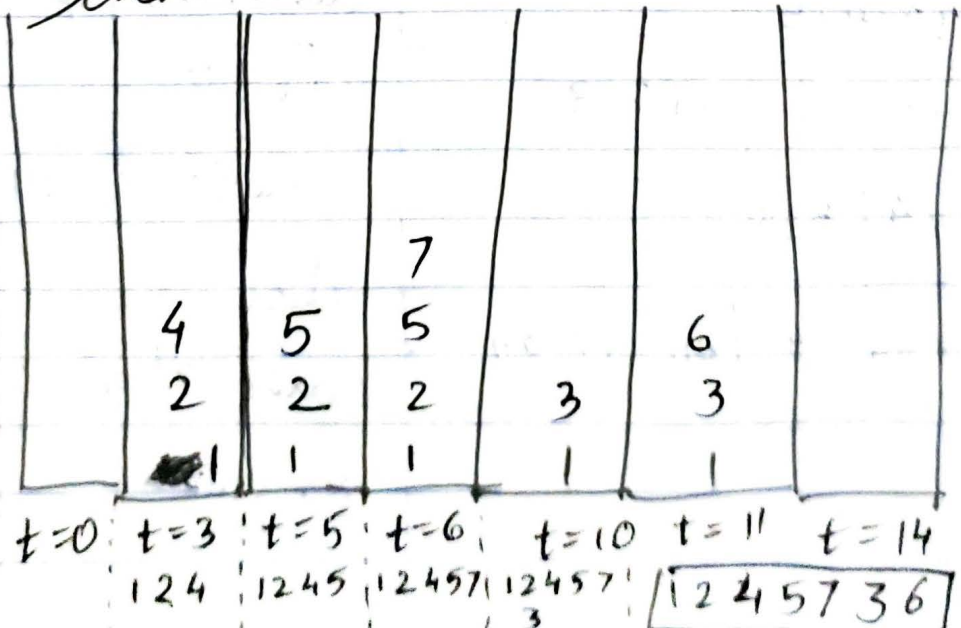
```
DFS(Node u)

    if u == null
        return
    visit(u)
    DFS(u.left)
    DFS(u.right)
```

Stack Trace:

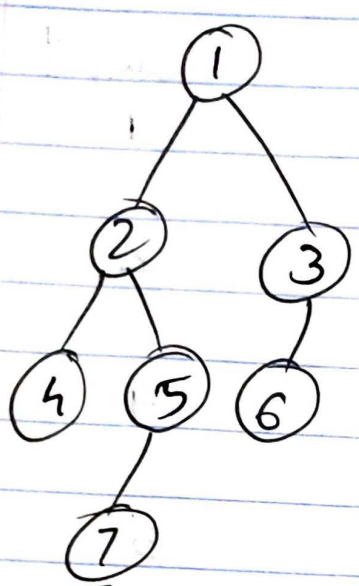
```
DFS(1)                                  Output:
    visit(1)                              1
    DFS(2)
        visit(2)                          2
        DFS(4)
            visit(4)                      4
        DFS(5)
            visit(5)                      5
            DFS(7)
                visit(7)                  7
    DFS(3)
        visit(3)                          3
        DFS(6)
            visit(6)                      6
```

—terminate

Schematics:



Tree : BFS.



Trace

q [1]

current = 1 | op: 1

q [2|3]

current = 2 | op: 1 2

q [3|4|5]

current = 3 | op: 1 2 3

q [4|5|6]

current = 4 | op: 1 2 3 4

q [5|6]

current = 5 | op: 1 2 3 4 5

q [6|7]

current = 6 | op: 1 2 3 4 5 6

q [7]

current = 7 | op: 1 2 3 4 5 6 7

terminate

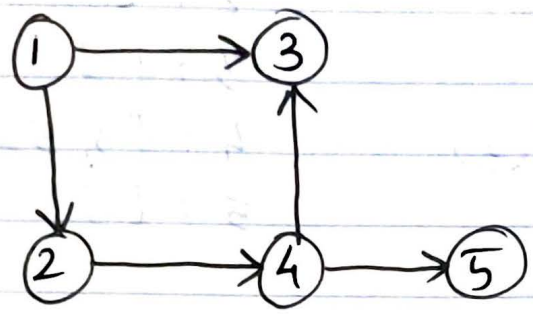
BFS Algo.

```
BFS(root)
    src ← root
    if src == null:
        return.
    q ← new Queue()
    q.add(src)

    while (!q.isEmpty)
        u ← q.remove()
        visit(u)
        if u.left != null
            q.add(u.left)
        if u.right != null
            q.add(u.right)
```

— terminate.

Graph Traversal:

DFS



Representation:
Adjacency List
HashMap<Integer, List<Integer>>

1 : [2, 3]
2 : [4]
3 : [ ]
4 : [3, 5]
5 : [ ]

DFS traversal.

visited = FFFFF, op: –

2 : 4
1 : 2 3

visited: TTFFF
op: 1 2

3 : [ ]
4 : 3 5
2 : 4
1 : 2 3

~~5 already visited~~
visited: TTTTF
op: 1 2 4 3

5 : [ ]
4 : 3 5
2 : 4
1 : 2 3

visited: TTTTT
op: 1 2 4 3 5

1 : 2 3

3 already visited
terminate

Algorithm:

⇒ DFS(V, E)

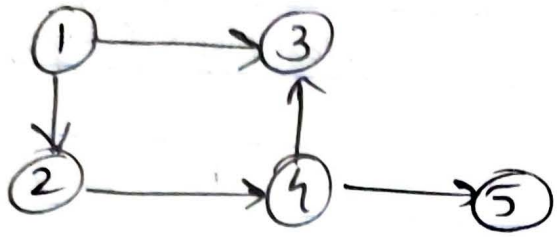
```
graph ← create(V,E)
visited ← new int[V]
for u ∈ V
    if ! visited[u]
        DFS-VISIT(u)
```

⇒ DFS-VISIT(Node u)

```
visited[u] ← True
for v ∈ u.adjList():
    if ! visited[v]
        DFS-VISIT(v)
```

– terminate.

BFS:



graph

1 : [2, 3]
2 : [4]
3 : []
4 : [3, 5]
5 : []

BFS traversal:

o/p :- visited FFFFF

q [1]

---

visited TFFFF

o/p : 1

q [2] [3]

---

visited : TTFFF

o/p : 12

q [3] [4]

---

visited : TTTFF

o/p : 123

q [4]

---

visited : TTTTF

o/p : 1234

q [5]

---

visited : TTTTT

o/p : 12345

terminate

BFS algorithm

```
BFS(V, E):
    graph ← createGraph(V, E)
    visited ← new int[V]
    src ← getSource(V).
    q ← new Queue()
    q.add(src)
    while(!q.isEmpty())
        u ← q.remove()
        if visited[u] == True
            continue
        visited[u] ← True
        for v ∈ u.adjList()
            if !visited[v]
                q.add(v)
```

terminate