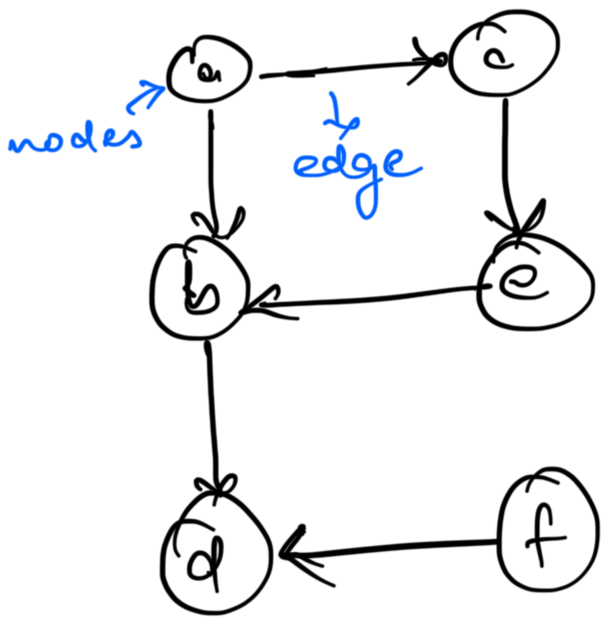
** Graphs: nodes + edges

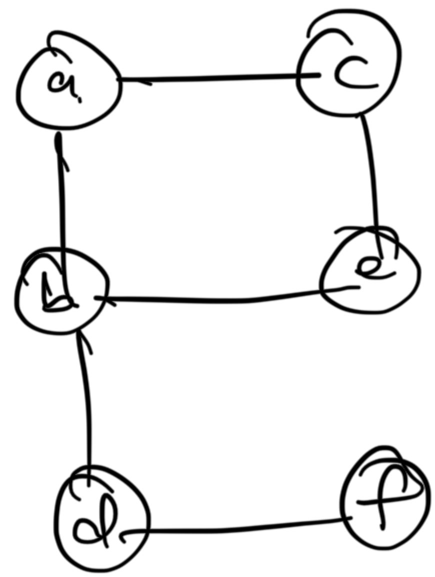
directed graphs



"obey the direction of arrowheads here"

"for a → b & c are neighbor nodes ..."

undirected graphs



"two way street"

In program, you write it as,

we use some hashmap data structure to represent an adjacency list"

Suppose for above example, we'll write it as:

adjancquncy list

```
{
  a: [b, c],
  b: [d],
  c: [e],
  d: [],
  e: [b],
  f: [d]
}
```

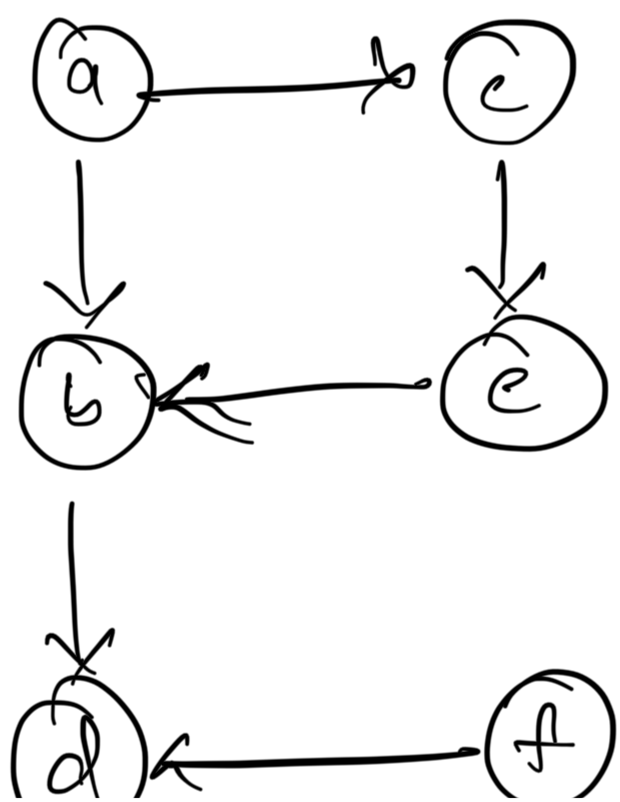
→ **Algorithms:**

__Depth first traversal__: It travels in a way to explore all possible nodes in one direction first and then move to next direction.

__Breadth first traversal__:

It travels in every direction possible together.

__Ex:-__



In depth first, the traversal would be:
- a, b, d
- a, c, e, b, d

In breadth first,

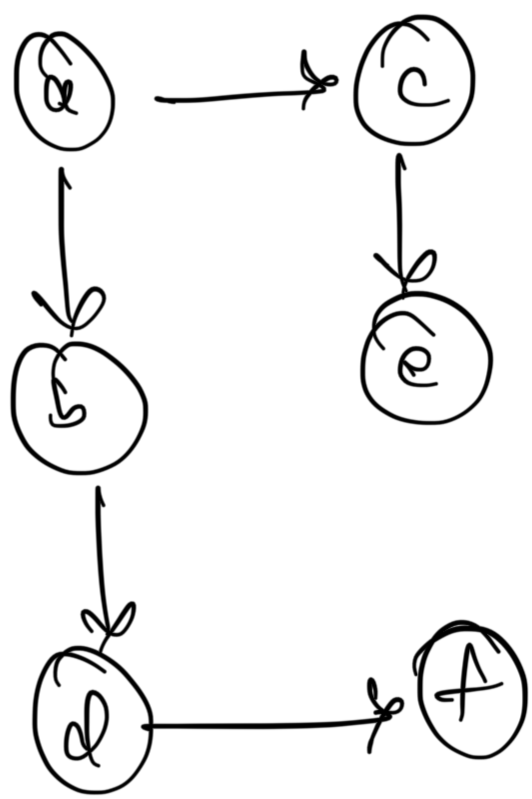
a, b, c . . . — .

→ Depth first: uses Stack
Breadth first: uses queue

- Stack is something where you add to the top and remove from the top.
- Queue is something where you

add to the back and remove from the front

These gives two different orderings and that's only difference between these two algorithms.

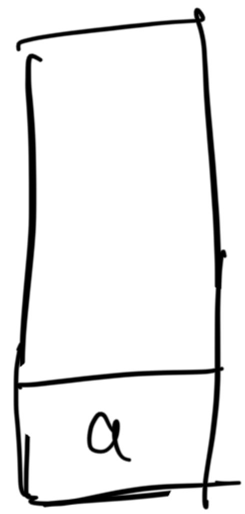
→ 

→ Depth-first search :-

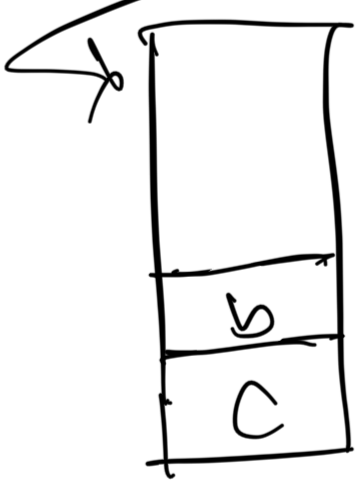
For the above graph, let's start

from choosing a as the starting point.

a → current

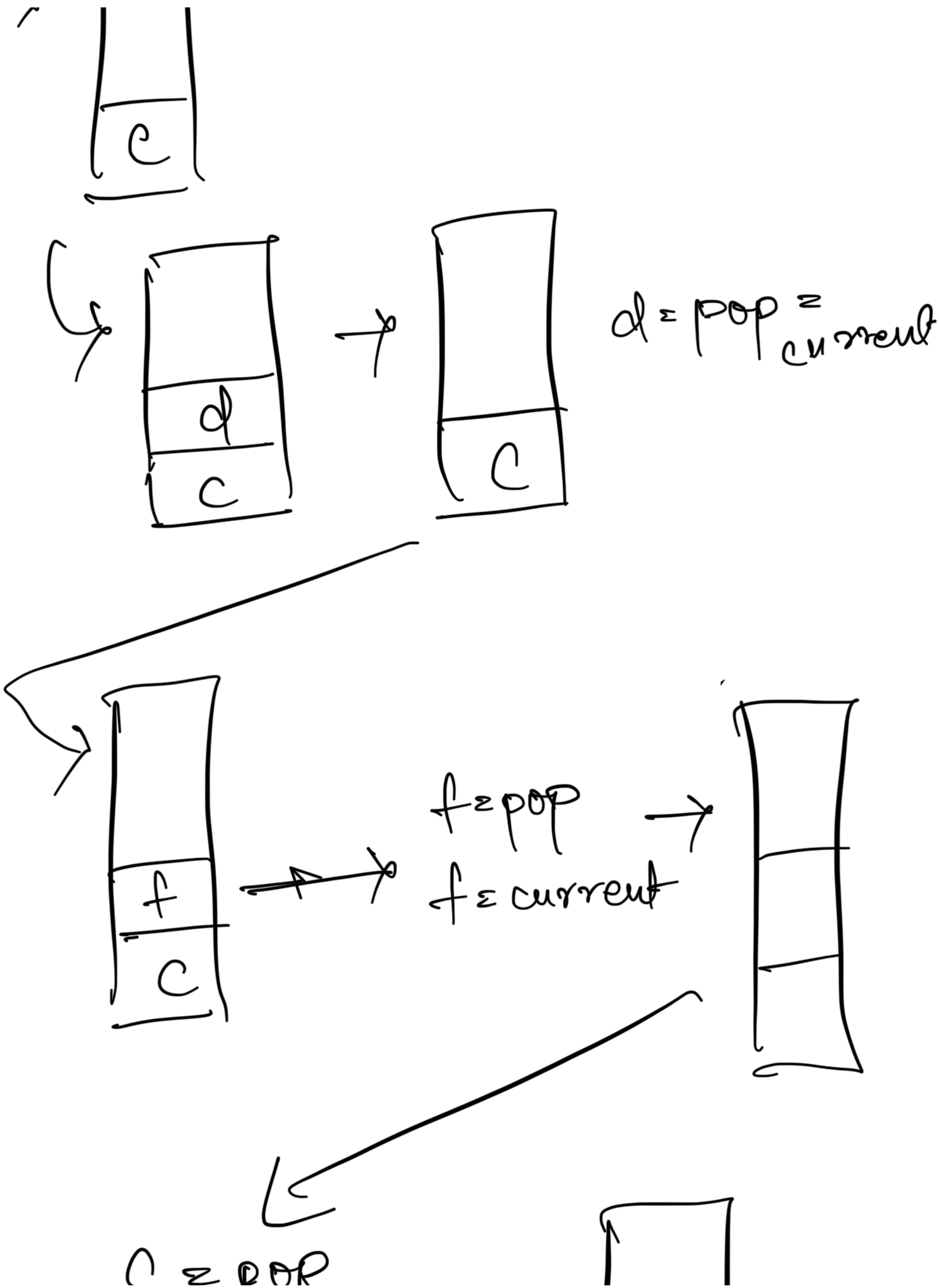


then pop



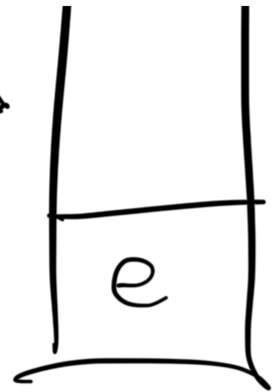
Now, pop what's on the top of the stack

b = current

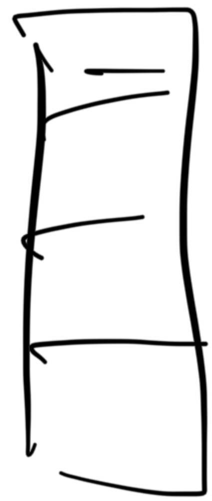


c
d
c
c
d = pop = current
f
c
f = pop
f = current
c = pop

c = current →

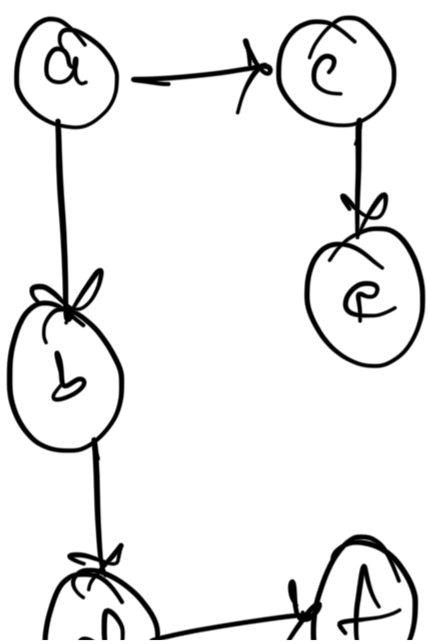


e = current = pop



if stack empty

→ Breadth First



Queue: First in first Out

a → initialize with

then c

first b

c b → a "current" ⇒

Now b ⇒ current

d c → b "current"

↳ e d → c ⇒ current

↳ f e → ⇒ d = current

$\hookrightarrow \xrightarrow{f} \Rightarrow e = \text{current}$

$\hookrightarrow \longrightarrow \Rightarrow f = \text{current}$

Queue empty

→ has path problem

Let's imagine a adjacency list ⇒ { f : [g, i],

g : [h],

h : [ ]

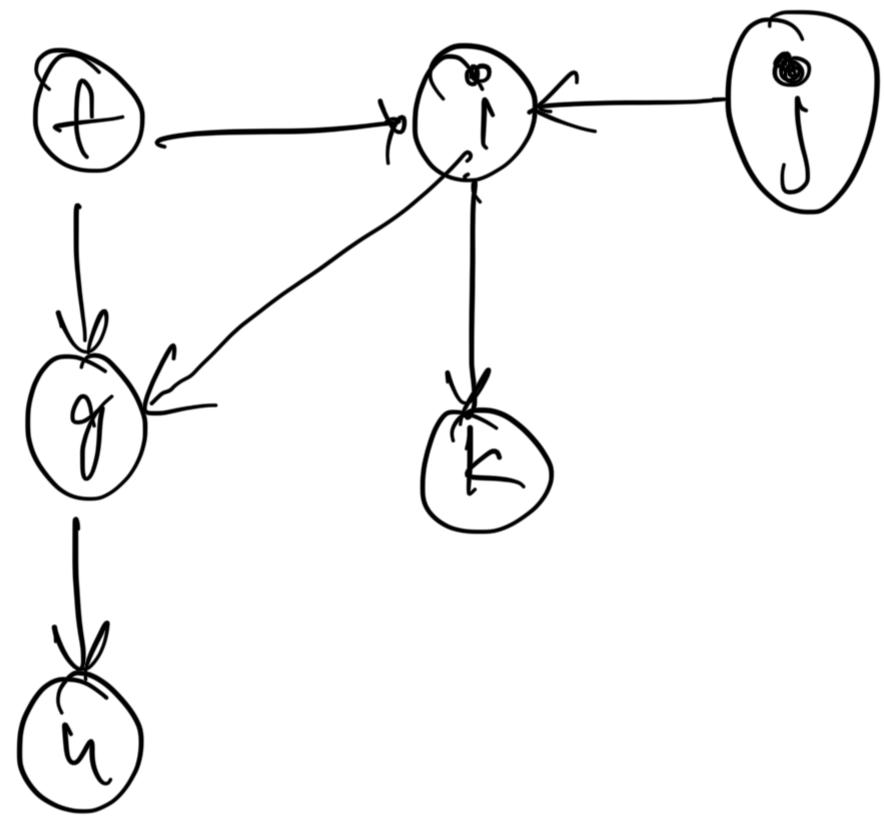
i : [g, k],

j : [i],

k : [ ] }

k: [ ]

Visualize the above:



This is an acyclic graph (no cycles)

→ Here, we want to take in not only the graph information but also a source and destination node.

We want to return True or false indicating whether or not we can travel from the source node to the destination node.

For this problem:

Source: f destination: k

Here, we can use both bfs or dfs approach.

Let's go with dfs:

$f \to g \to h$

$f \to i \to g \to h$

$f \to i \to k$ return True

Time complexity :-

let's say :- $n = \#$ nodes
$e = \#$ edges

Time = $O(e)$, we have to travel every edge of our graph.

Here, the Space Complexity depends on the # of nodes.

Undirected graph problem :-

let's consider:

edges : [
[i, j],
[k, i],
[m, k],

[k, l],
[o, n]
]

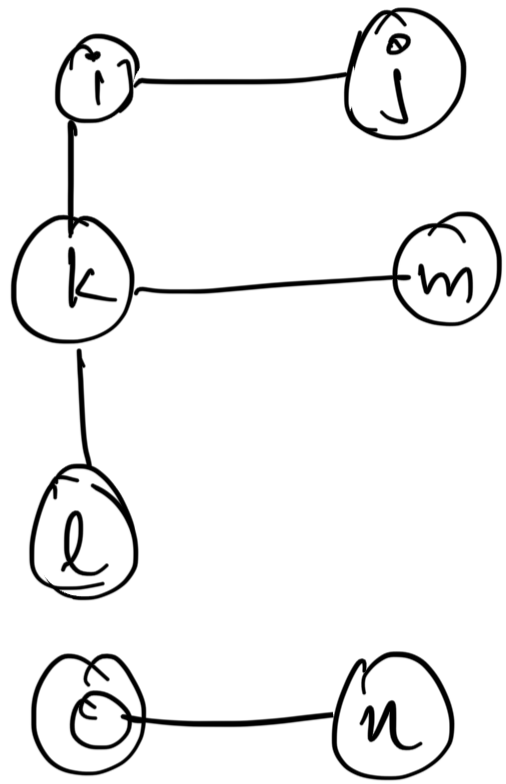
edge_list ↑

Let's convert edge list to adjacency list.

```
edges: [
  [i, j],
  [k, i],
  [m, k],
  [k, l],
  [o, n]
]
```

$\Rightarrow$

```
graph: {
  i: [j, k]
  j: [i]
  k: [i, m, l]
  m: [k]
  l: [k]
  o: [n]
  n: [o]
```



- Connected components count problem:-
- largest component problem:
- Shortest path:-
- island count problem:-
- minimum island size:-

Grid Graph Problem