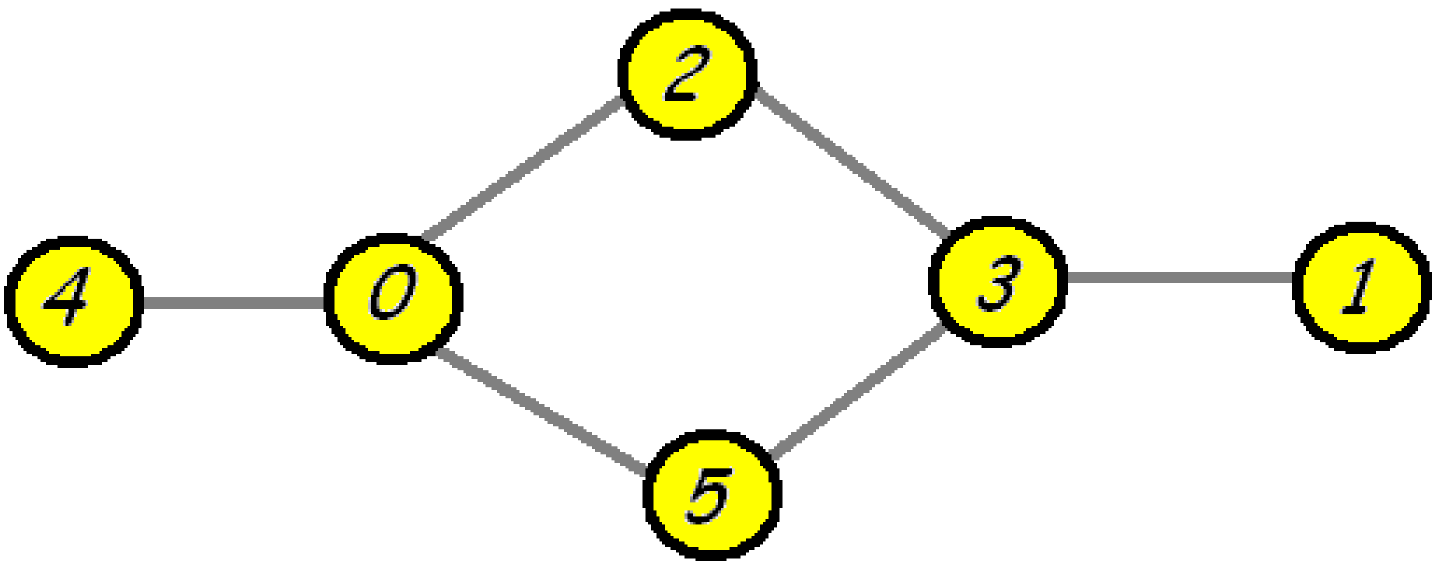
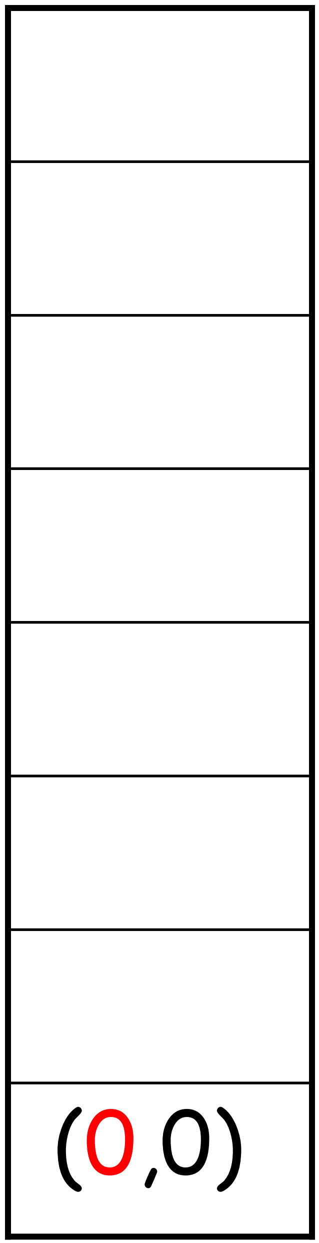
Start at vertex 0.

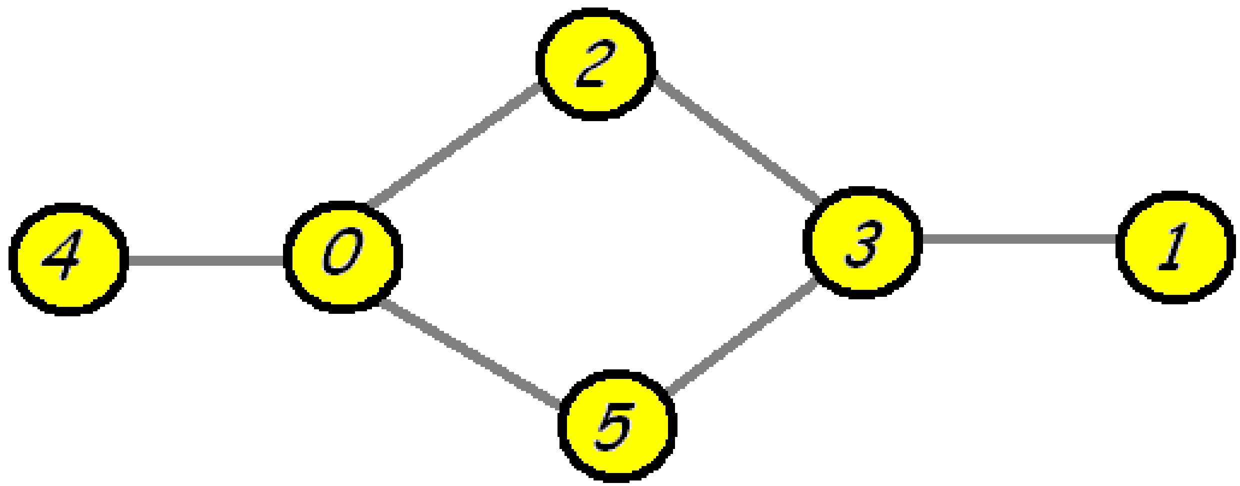
1. Find the parent and DFI arrays resulting from a DFS. Show the stack at each step of the computation.

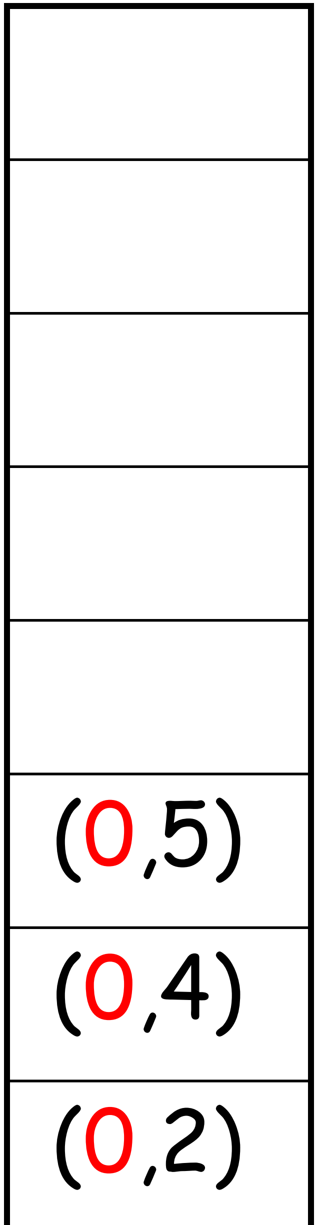


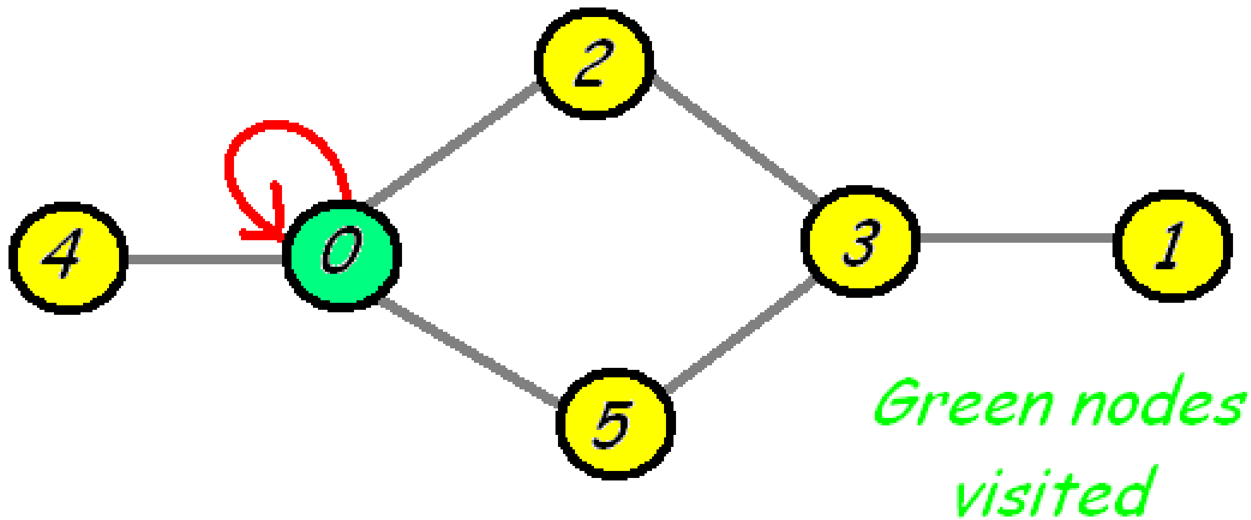
2. Apply BFS showing the queue, parent, BFI and level information.

Depth First Search (DFS)







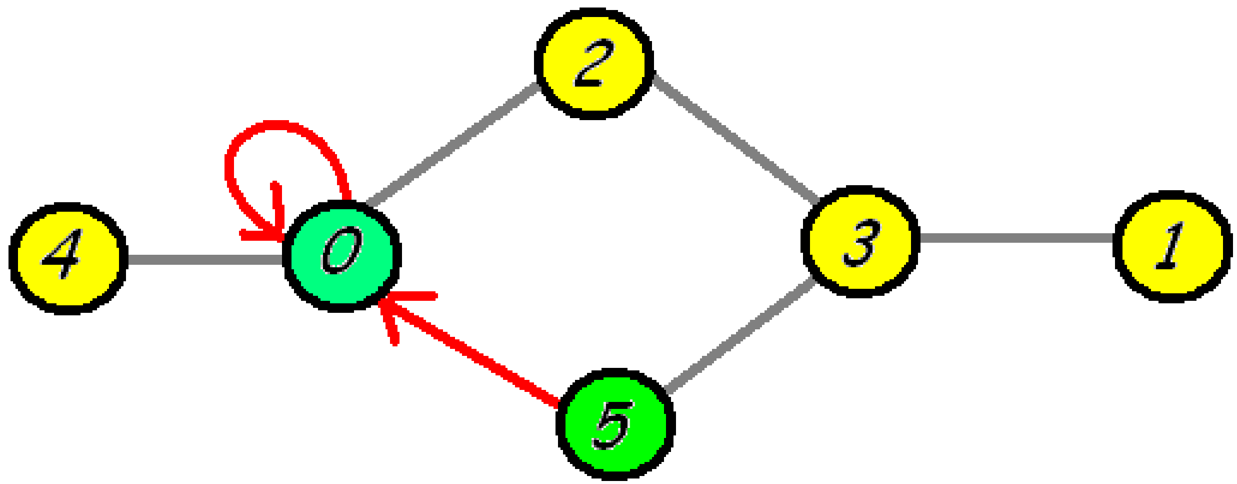


0. Pop (0,0)

Neighbours of 0 are 2, 4, 5.

Push (0, 2), (0, 4), (0,5).

(5,3)
(0,4)
(0,2)

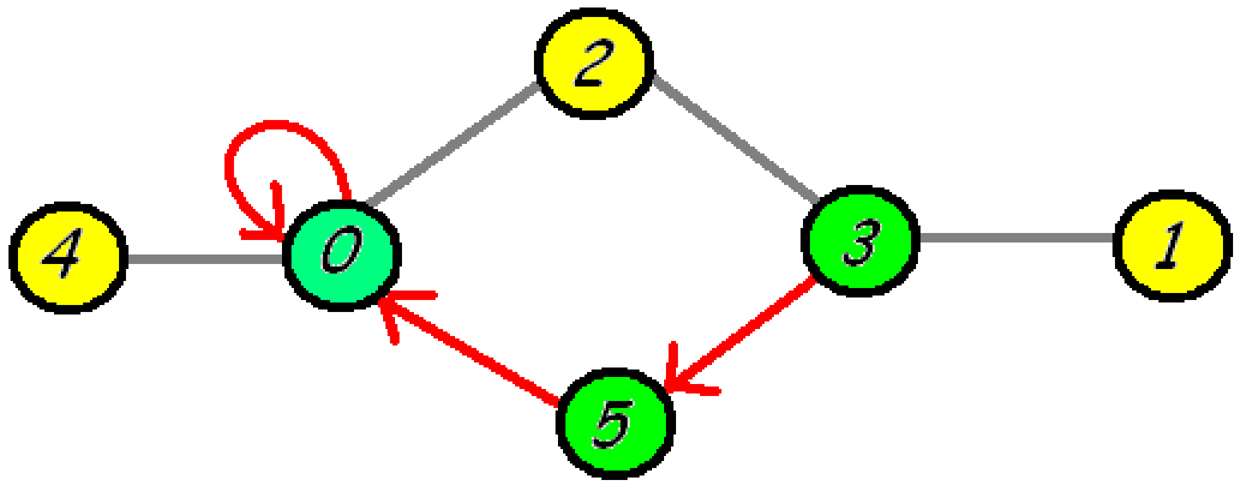


1. Pop (0,5)

Neighbours of 5 are 0, 3.

Push (5,3).

(3,2)
(3,1)
(0,4)
(0,2)

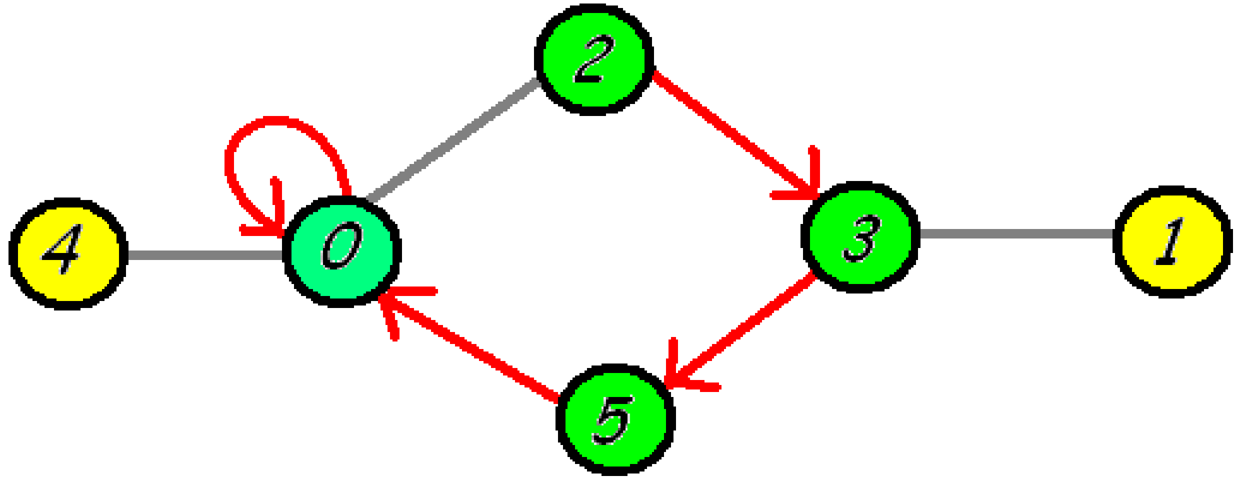


2. Pop (5,3)

Neighbours of 3 are 1, 2, 5.

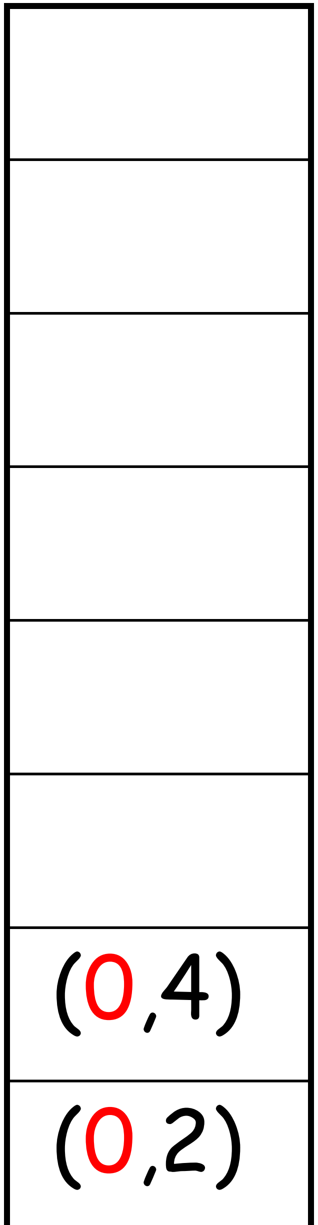
Push (3,1), (3,2).

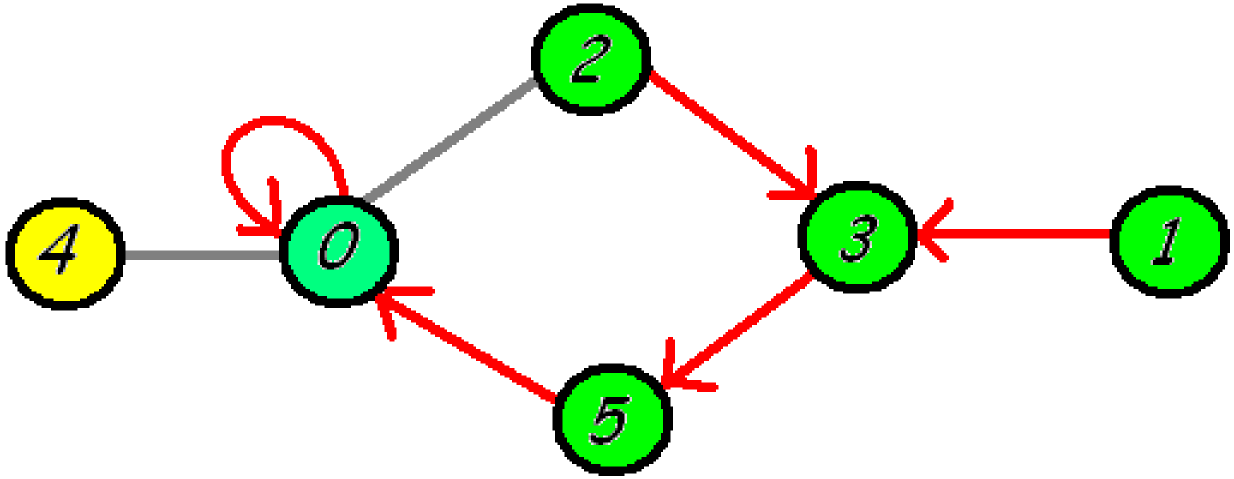
(3,1)
(0,4)
(0,2)



3. Pop (3,2)

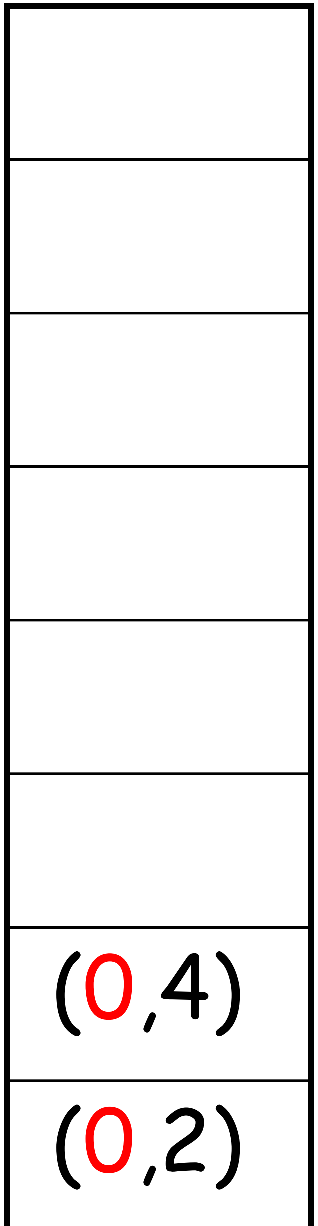
Neighbours of 2 are 0, 3.

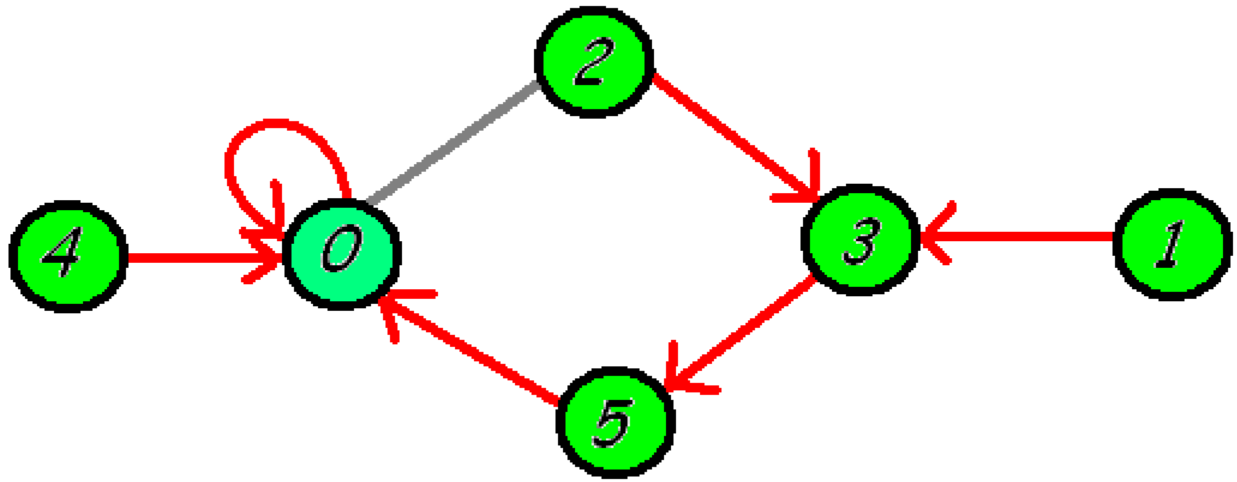




4. Pop (3,1)

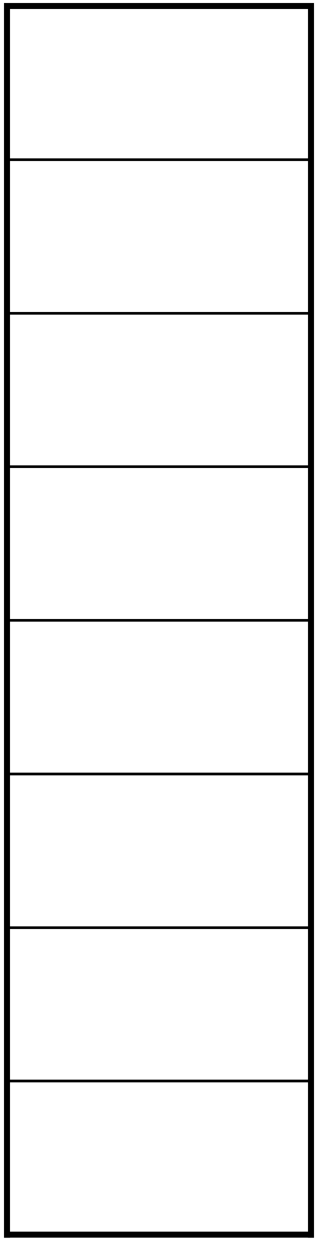
Neighbour of 1 is 3.

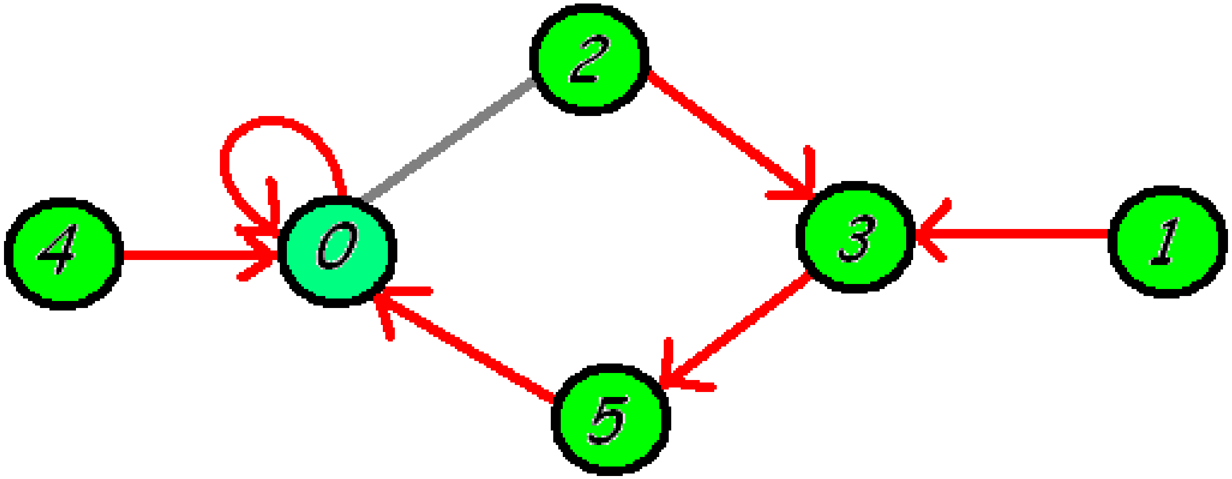




5. Pop (0,4)

Neighbour of 4 is 0.

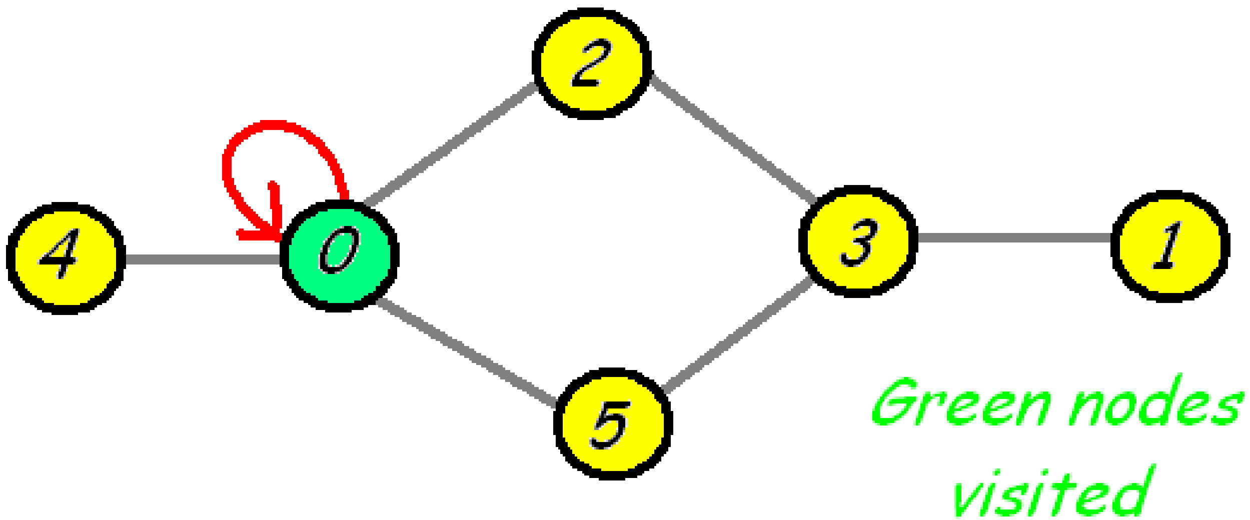




6. Pop (0,2)

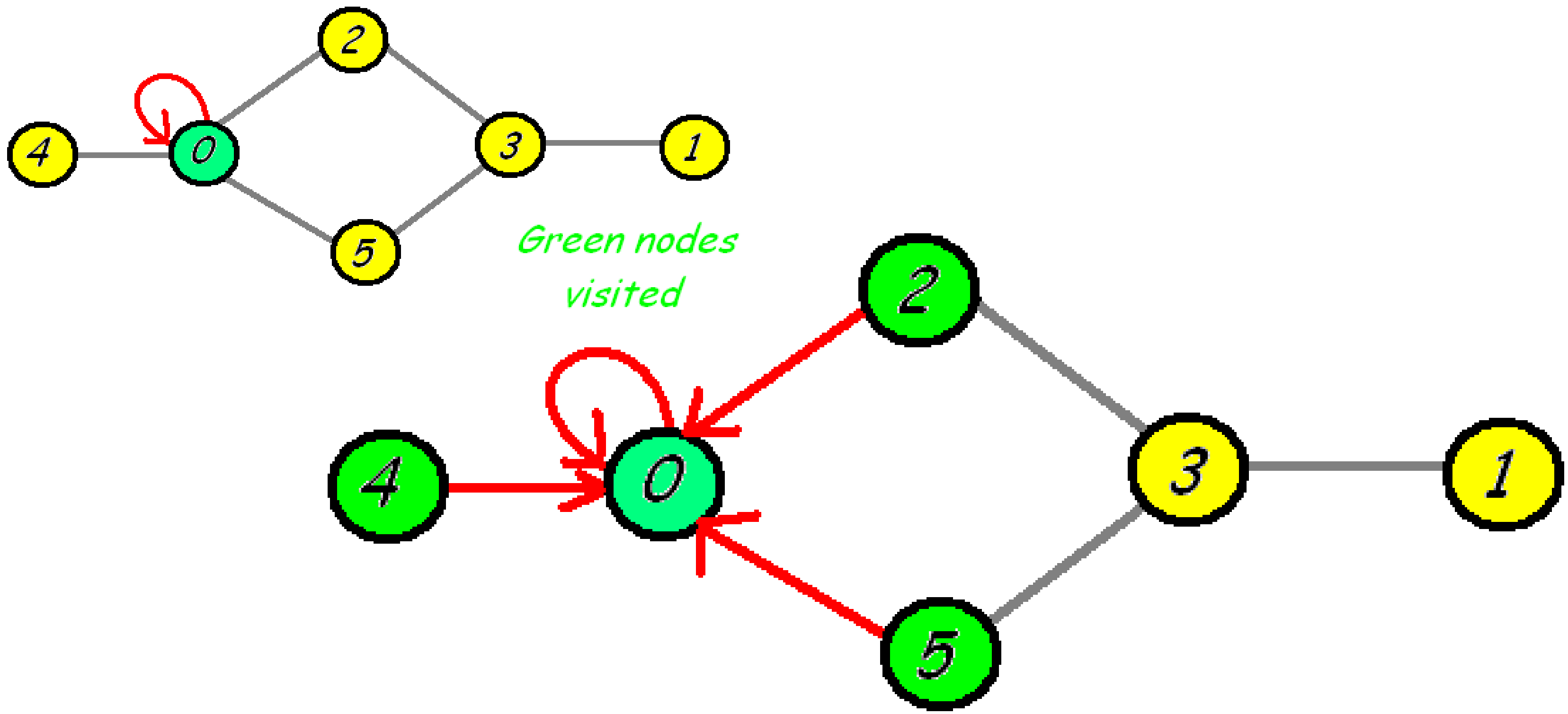
Ignore because 2 has already been visited.

BFS: Start with vertex 0 in queue.



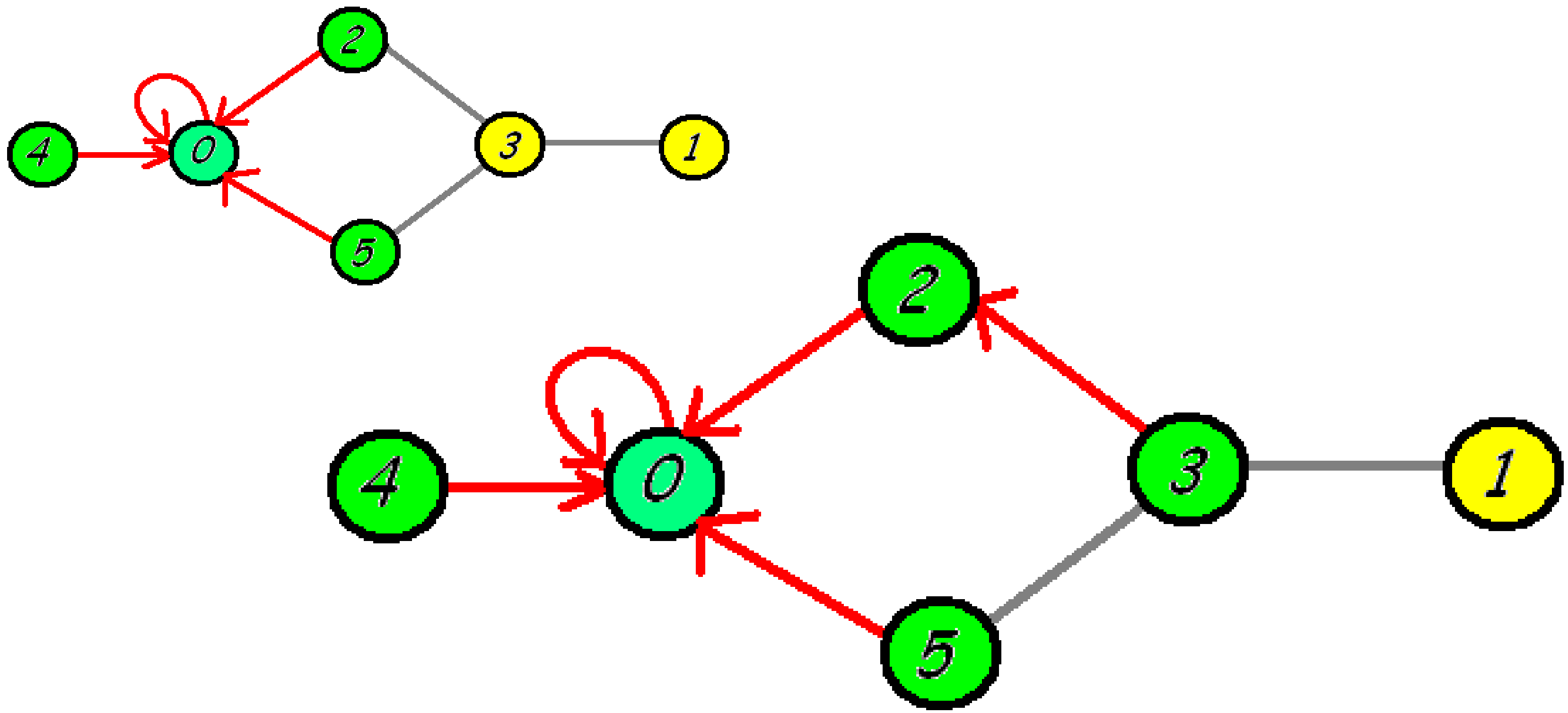
i	0	1	2	3	4	5
queue[i]	0					

BFS: Traverse neighbours of 0: 2, 4, 5



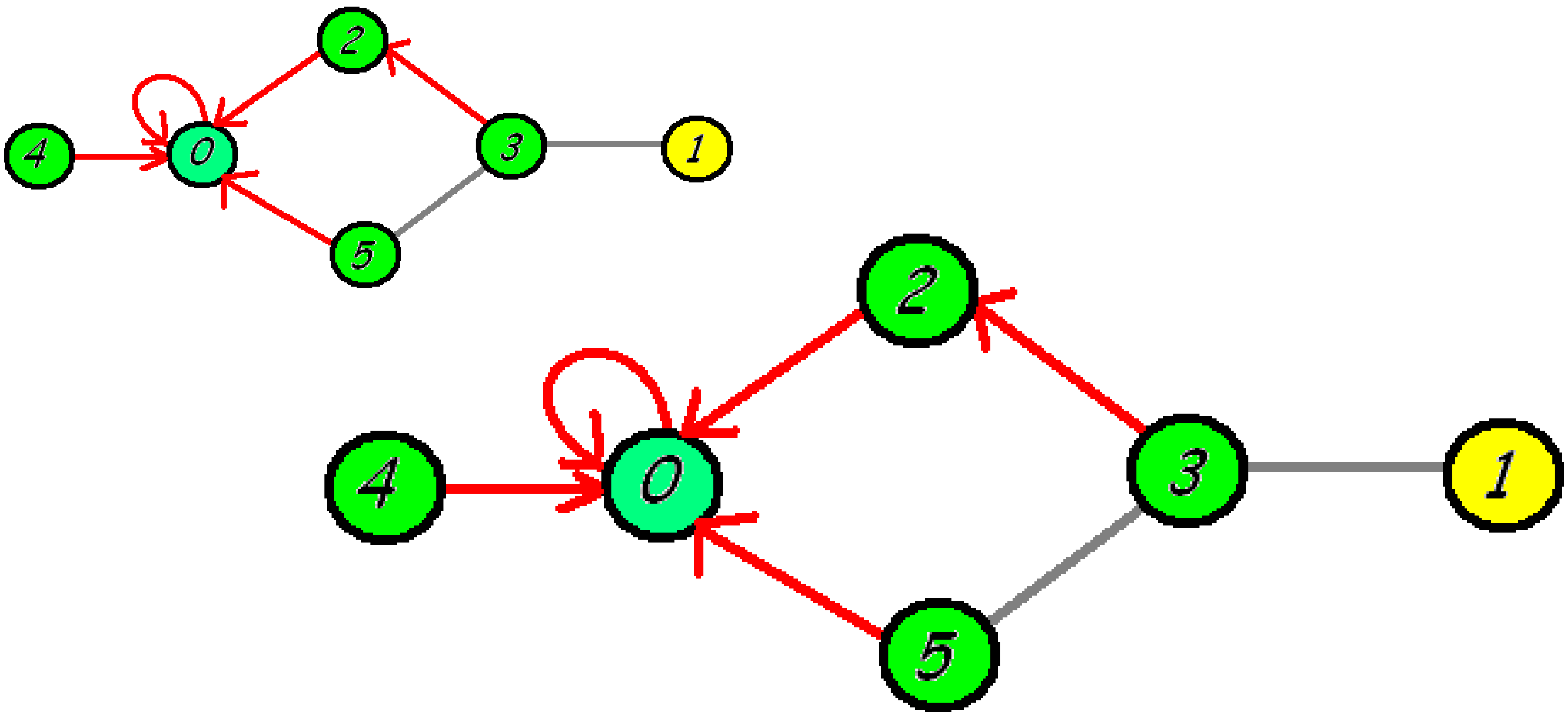
i	0	1	2	3	4	5
queue[i]	0	2	4	5		

BFS: Visit neighbours of 2: 0, 3



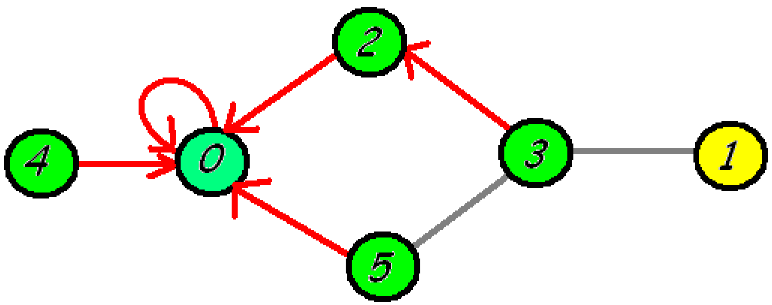
i	0	1	2	3	4	5
queue[i]	0	2	4	5	3	

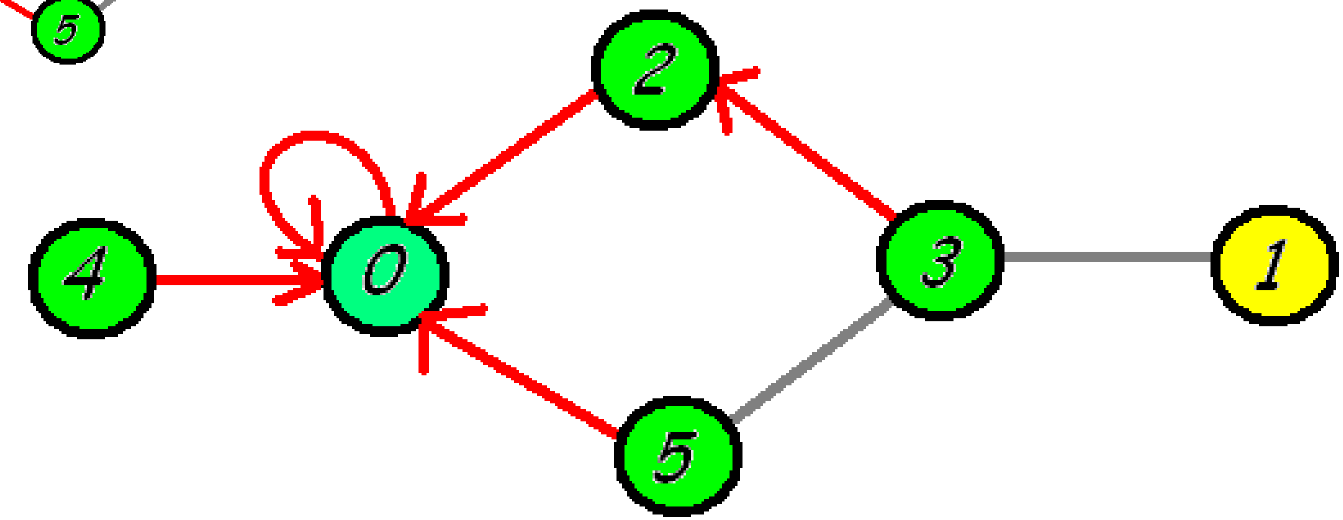
BFS: Visit neighbours of 4: 0



i	0	1	2	3	4	5
queue[i]	0	2	4	5	3	

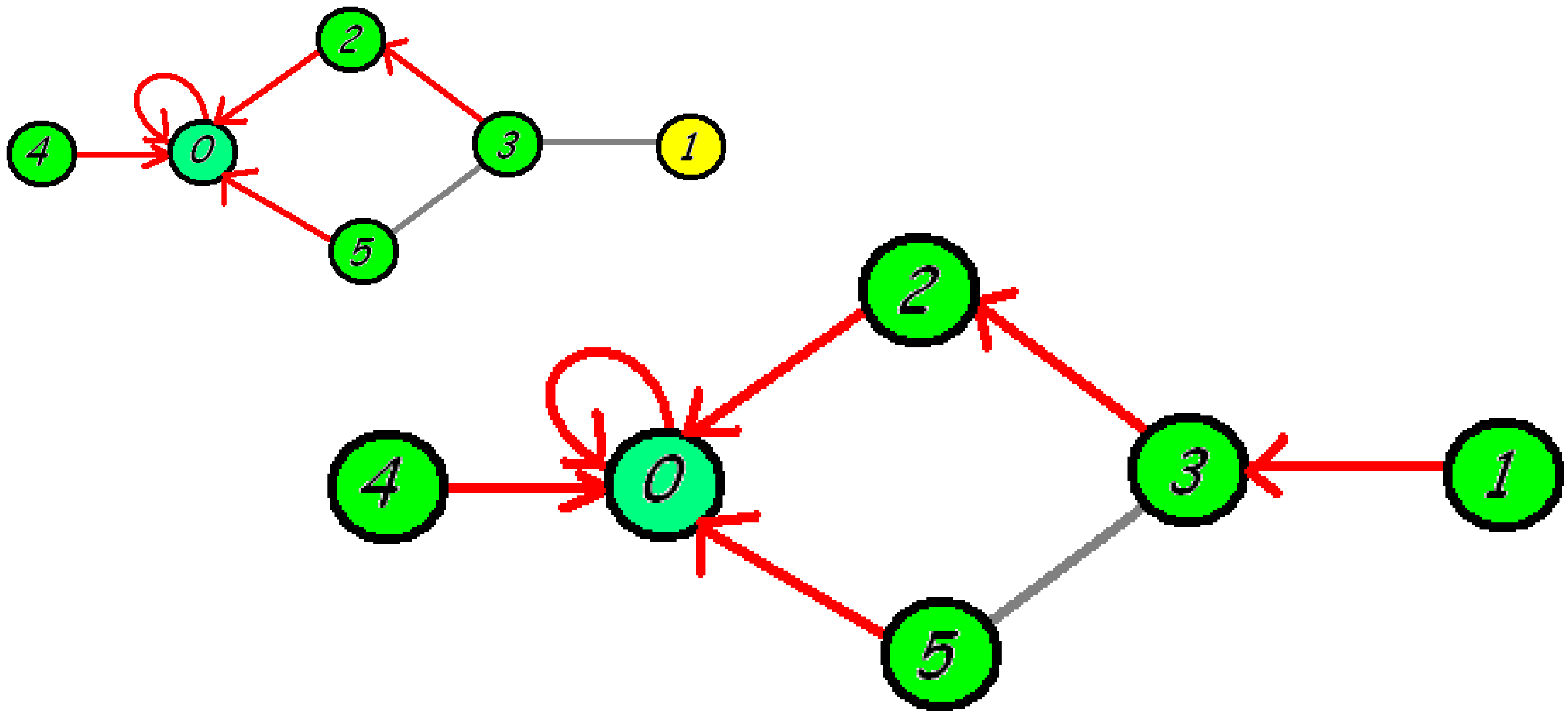
BFS: Visit neighbours of 5: 0, 3





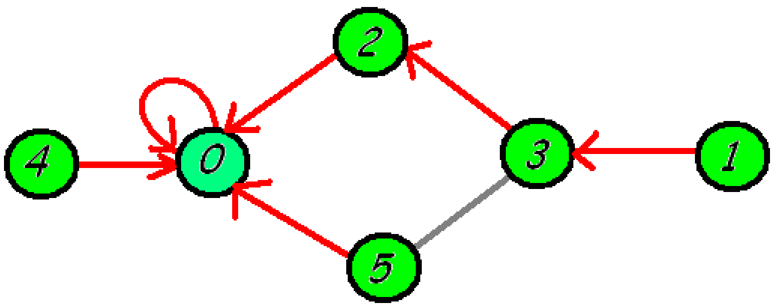
i	0	1	2	3	4	5
queue[i]	0	2	4	5	3	

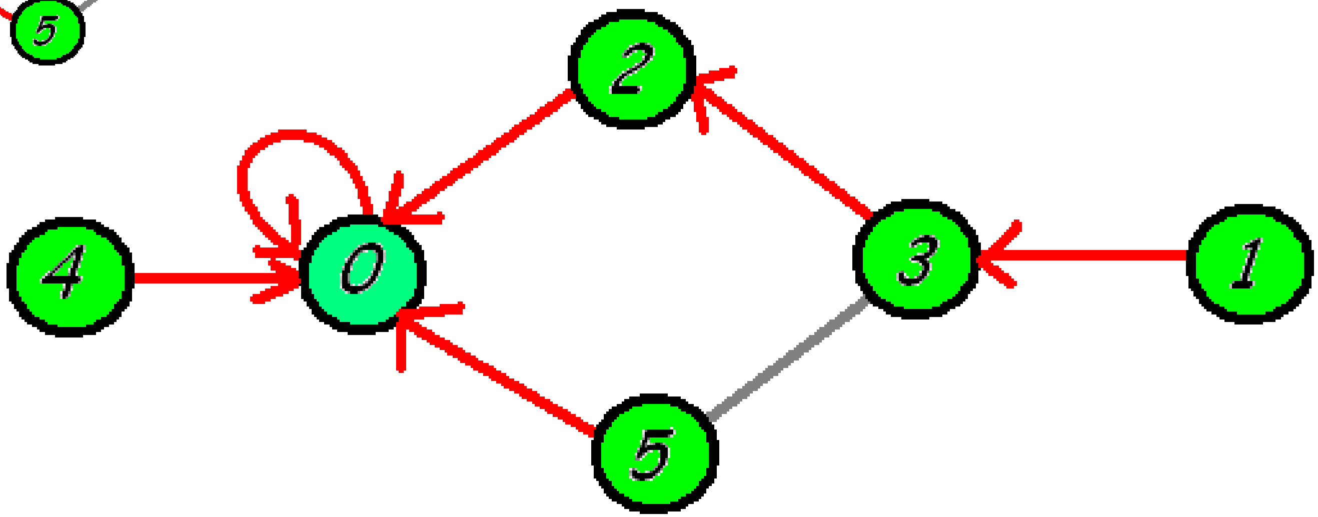
BFS: Visit neighbours of 3: 1, 2, 5



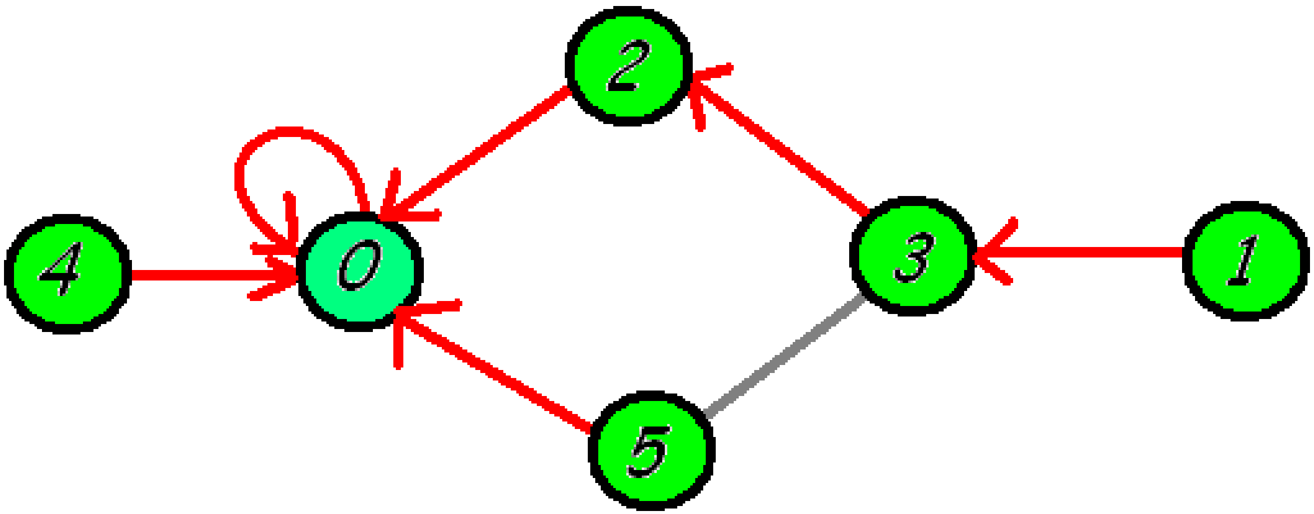
i	0	1	2	3	4	5
queue[i]	0	2	4	5	3	1

BFS: Visit neighbours of 1: 3



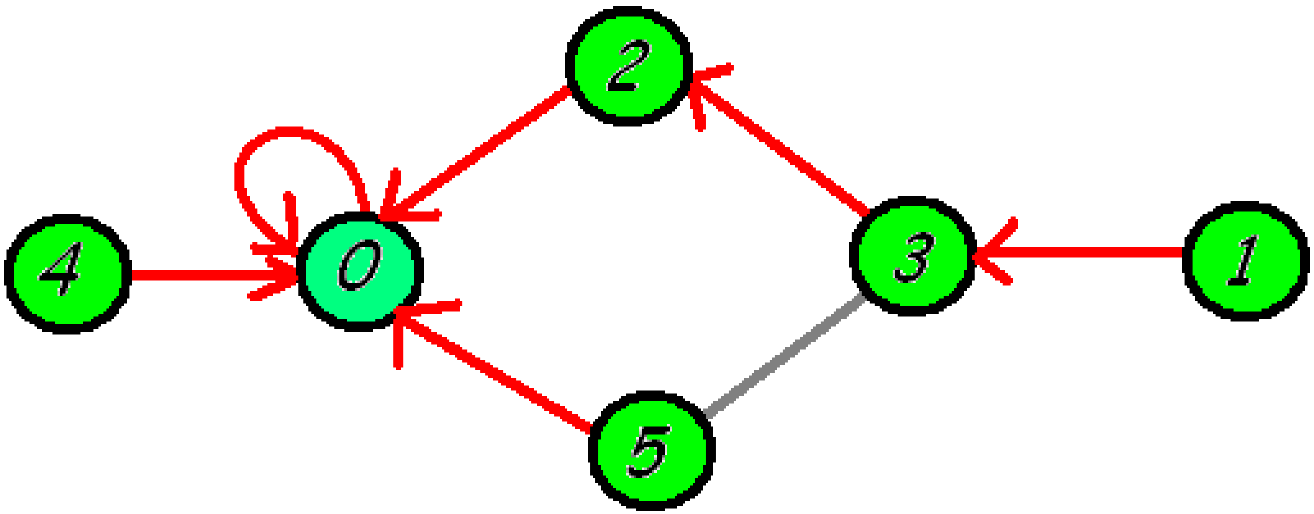


i	0	1	2	3	4	5
queue[i]	0	2	4	5	3	1



i	0	1	2	3	4	5
queue[i]	0	2	4	5	3	1

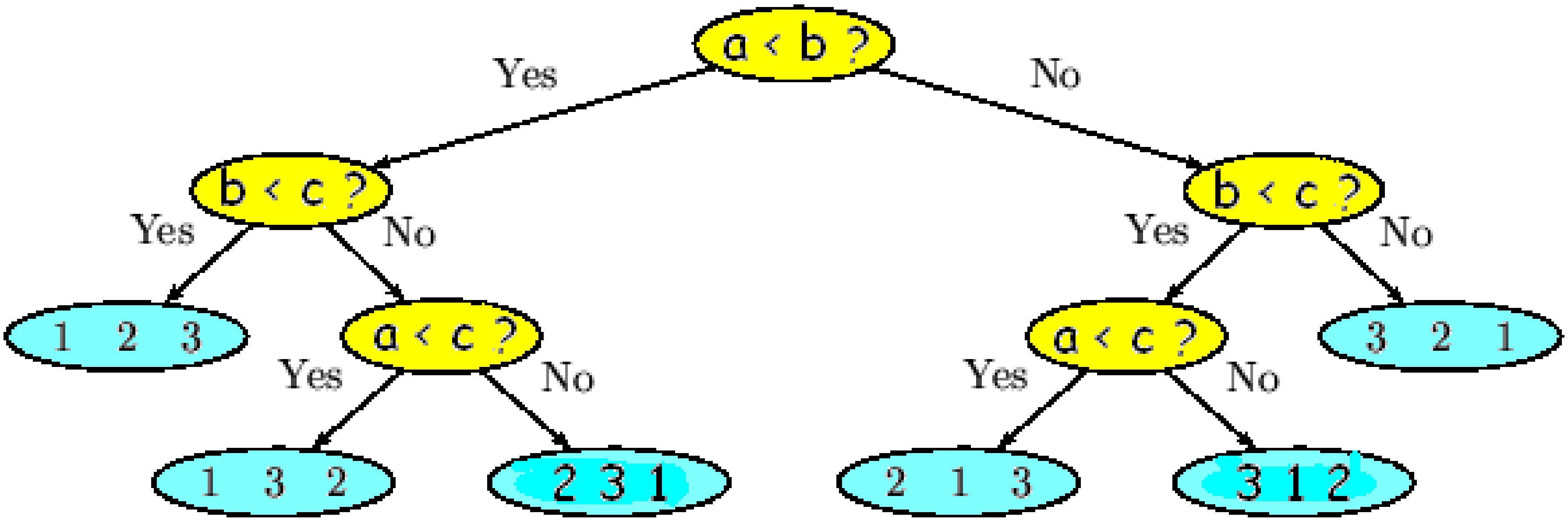
i	0	1	2	3	4	5
BFI[i]	0	5	1	4	2	3



i	0	1	2	3	4	5
parent[i]	0	3	0	2	0	0

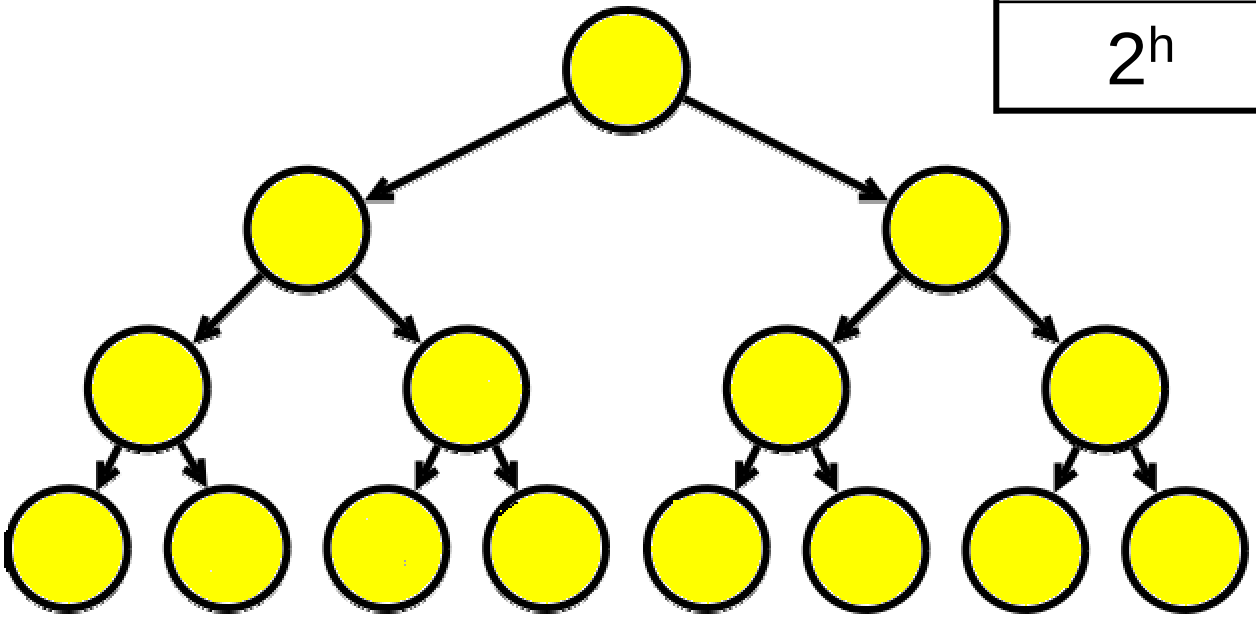
i	0	1	2	3	4	5
level[i]	0	3	1	2	1	1

A Decision Tree: Input is a, b, c



Note that a complete binary tree which has r leaves has height $\theta(\log_2 r)$:

Leaves	Nodes	Height
?	1	0
2	3	1
4	7	2
8	15	3
	…	…
2^h	$2^{h+1} - 1$	h



We can use our tactics for lower and upper bounding to prove that:

$$\log_2(\ n!\ ) \in \theta(n \log_2 n)$$

Which sorting algorithms have optimal time complexities for the comparison model (in a Big Oh sense)?

These $\theta(n \log_2 n)$ in the worst case:

Heapsort, Mergesort, Mediansort

Not optimal since worst case is $\theta(n^2)$:

Quicksort, Maxsort, Binary Tree Sort

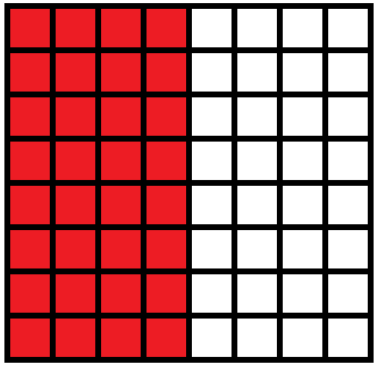
Hashing:

After hashing to choose an initial table location, use a second hash function to choose the jump amount.

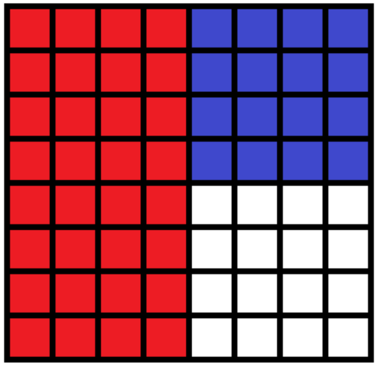
This helps to ensure the probe sequence is likely to hit an empty spot with a probability that corresponds to the percent of open spots in the table.

Worst case examples of O(n) however can still be created (chose keys all having the same hash value and same jump value).

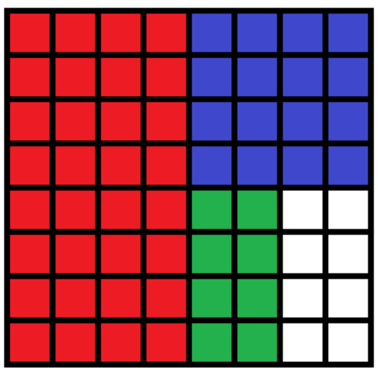
If a hash table was 50% full, and if we assume that each time we probe a cell that the probability it is empty is 50%, what is the expected number of probes needed to insert an element into a hash table?



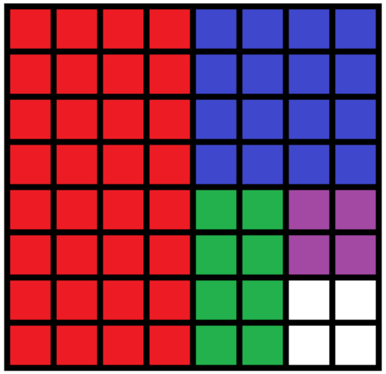
$$\frac{1}{2} = 1 - \frac{1}{2}$$



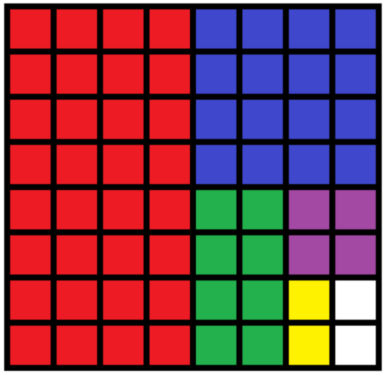
$$\frac{1}{2} + \frac{1}{4} = 1 - \frac{1}{4}$$



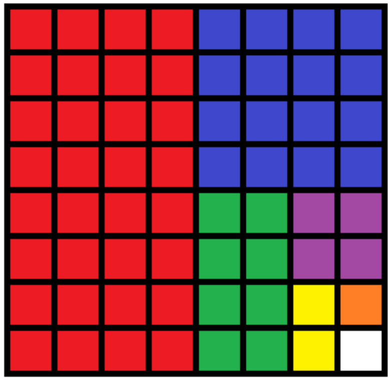
$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} = 1 - \frac{1}{8}$$



$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} = 1 - \frac{1}{16}$$



$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} = 1 - \frac{1}{32}$$



$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{32} + \frac{1}{64} = 1 - \frac{1}{64}$$

H= hash table size, k = H/2.

It takes 1 probe $\frac{1}{2}$ of the time.

It takes 2 probes $\frac{1}{4}$ of the time.

It takes 3 probes $\frac{1}{8}$ of the time.

It takes 4 probes $\frac{1}{16}$ of the time.

…

It takes k probes $1/2^k$ of the time.

The insertion can take at most k+1 probes because the hash table contains k items.

It takes k+1 probes $1/2^k$ of the time.

H= hash table size, k = H/2.

Expected number of probes:

$1 * \frac{1}{2} + 2 * \frac{1}{4} + 3 * \frac{1}{8}$

$+ 4 * \frac{1}{16} + \ldots + k * 1/2^k$

$+ (k+1) * 1/2^k$

What is this sum?

The insertion can take at most k+1 probes because the hash table contains k items.

It takes k+1 probes $1/2^k$ of the time.