Course: Analysis of Algorithms
Code: CS33104
Branch: MCA -3rd Semester

Lecture – 2 : Brute Force and Exhaustive Search

Faculty & Coordinator : Dr. J Sathish Kumar (JSK)

Department of Computer Science and Engineering

Motilal Nehru National Institute of Technology Allahabad,
Prayagraj-211004

Introduction

- ***Brute force*** is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.
- "force" implied "Just do it!"
- Brute-force strategy is indeed the one that is easiest to apply.
- Example:
  - Brute-force approach to the problem of sorting: given a list of *n* orderable items (e.g., numbers, characters from some alphabet, character strings), rearrange them in nondecreasing order.
  - **Bubble Sort, Selection Sort, Insertion Sort**

Sorting Algorithms

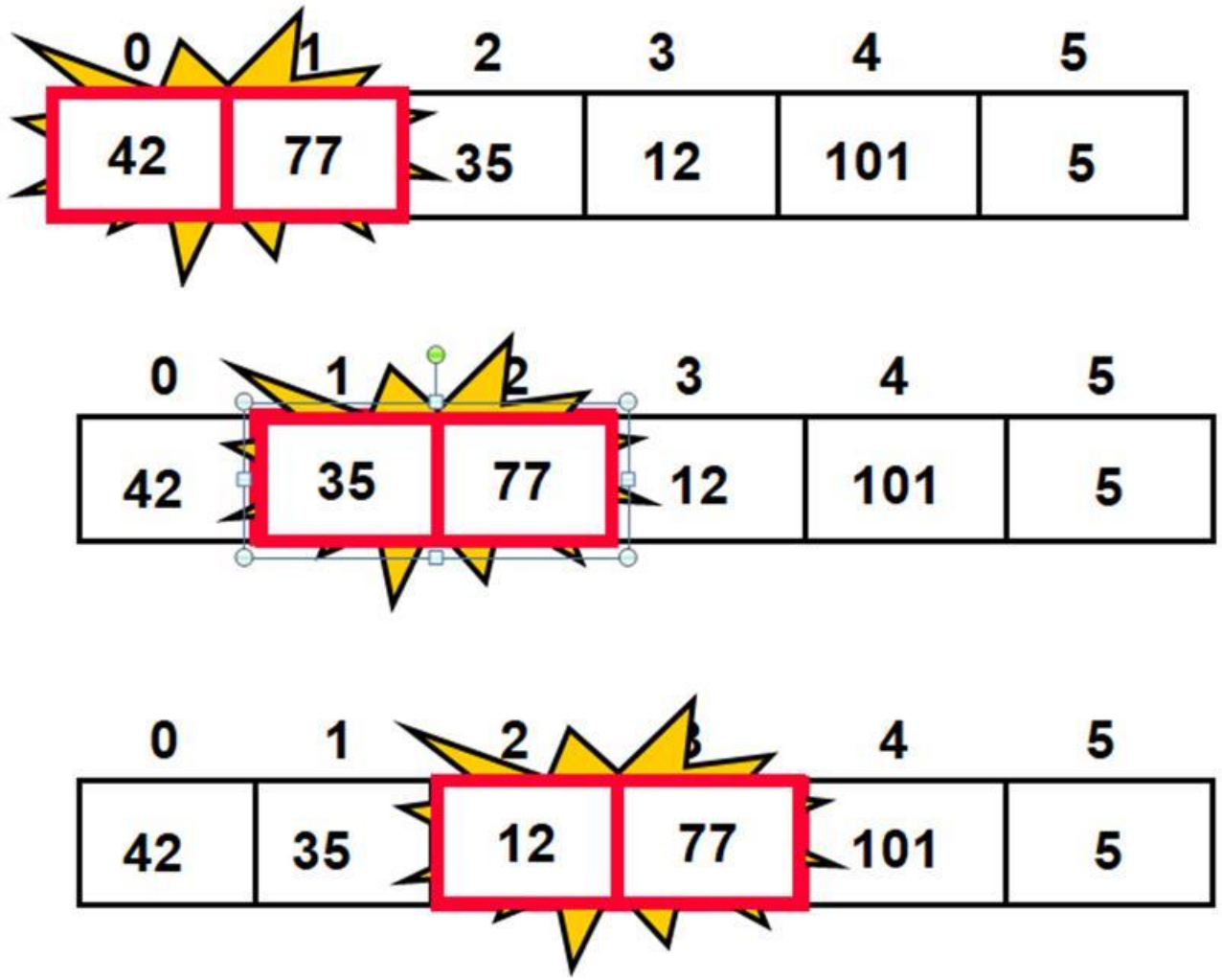
- **Bubble Sort**
- **Selection Sort**
- **Insertion Sort**

Bubble sort

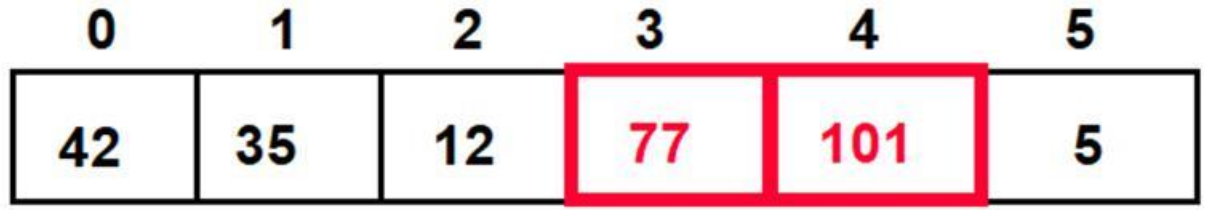
- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary

- Specifically:
  - scan the list, exchanging adjacent elements if they are not in relative order; this bubbles the highest value to the top
  - scan the list again, bubbling up the second highest value
  - repeat until all elements have been placed in their proper order

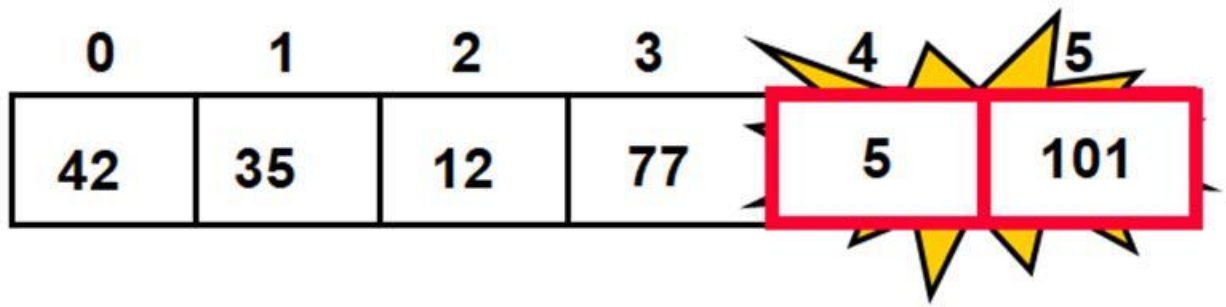
Example

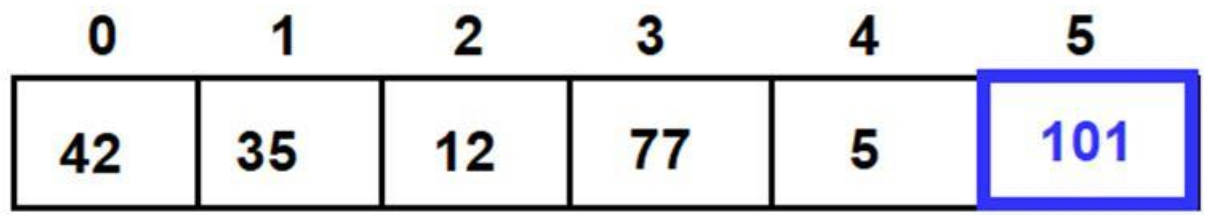

0
1
2
3
4
5
42
77
35
12
101
5
0
1
2
3
4
5
42
35
77
12
101
5
0
1
2
3
4
5
42
35
12
77
101
5

Example



No need to swap





Largest value correctly placed

Class Exercise #1

14, 2, 10, 5, 1, 3, 17, 7

Class Exercise #1

Pass1

- 14,2,10,5,1,3,17,7
- 2, 14, 10,5,1,3,17,7
- 2,10,14,5,1,3,17,7
- 2,10,5,14,1,3,17,7
- 2,10,5,1,14,3,17,7
- 2,10,5,1,3,14,17,7
- 2,10,5,1,3,14,7,**17**

Pass2

- 2,10,5,1,3,14,7,17
- 2,5,10,1,3,14,7,17
- 2,5,1,10,3,14,7,17
- 2,5,1,3,10,14,7,17
- 2,5,1,3,10,7,**14,17**

Class Exercise #1

Pass3

- 2,5,1,3,10,7,14,17
- 2,1,5,3,10,7,14,17
- 2,1,3,5,10,7,14,17
- 2,1,3,5,7,**10,14,17**

Pass4

- 2,1,3,5,7,10,14,17
- 1,2,3,5,7,10,14,17

Pseudo Code

ALGORITHM *BubbleSort*$(A[0..n-1])$

//Sorts a given array by bubble sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow 0$ **to** $n-2-i$ **do**
        if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

Running Time

Running time (# comparisons) for input size n:

$$\sum_{i=0}^{n-1}\sum_{j=1}^{n-1-i} 1 = \sum_{i=0}^{n-1}(n-1-i)$$

$$= n\sum_{i=0}^{n-1} 1 - \sum_{i=0}^{n-1} 1 - \sum_{i=0}^{n-1} i$$

$$= n^2 - n - \frac{(n-1)n}{2}$$

$$= \Theta(n^2)$$

Selection Sort

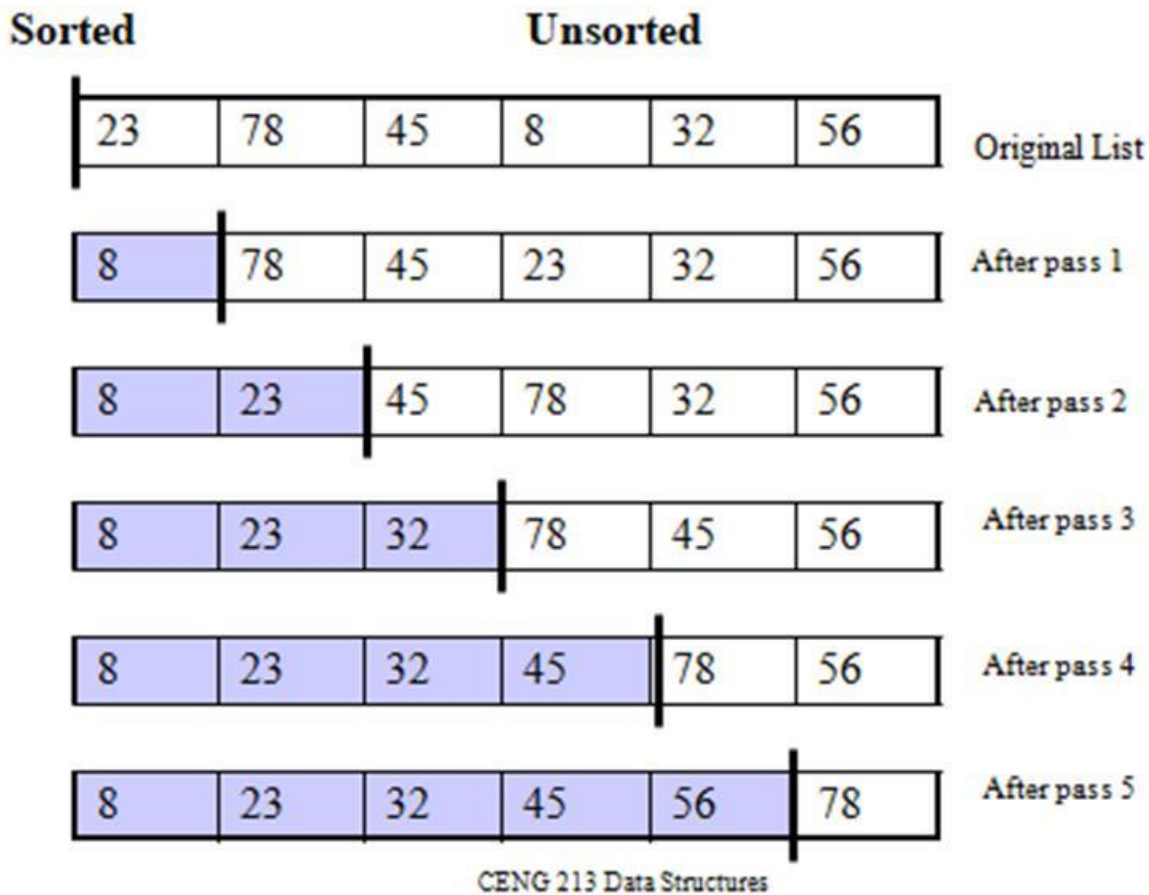
- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position

- Specifically:
  - find the smallest value in the list
  - switch it with the value in the first position
  - find the next smallest value in the list
  - switch it with the value in the second position
  - repeat until all values are in their proper places

Class Exercise:

23,78, 45, 8, 32, 56

Class Excercise

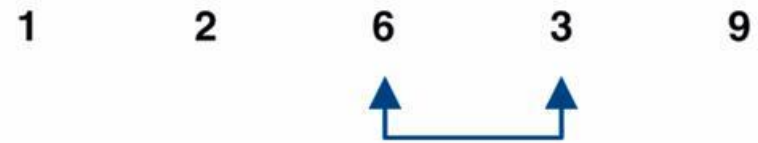


Example

Scan right starting with 3.
1 is the smallest. Exchange 1 and 3.

Scan right starting with 9.
2 is the smallest. Exchange 9 and 2.

Scan right starting with 6.
3 is the smallest. Exchange 6 and 3.



Scan right starting with 6.
6 is the smallest. Exchange 6 and 6.

Pseudo Code

ALGORITHM *SelectionSort*$(A[0..n-1])$

```
//Sorts a given array by selection sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if A[j] < A[min]  min ← j
    swap A[i] and A[min]
```

Running Time

Running time (# comparisons) for input size n:

$$\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-1}(n-1-(i+1)+1)$$

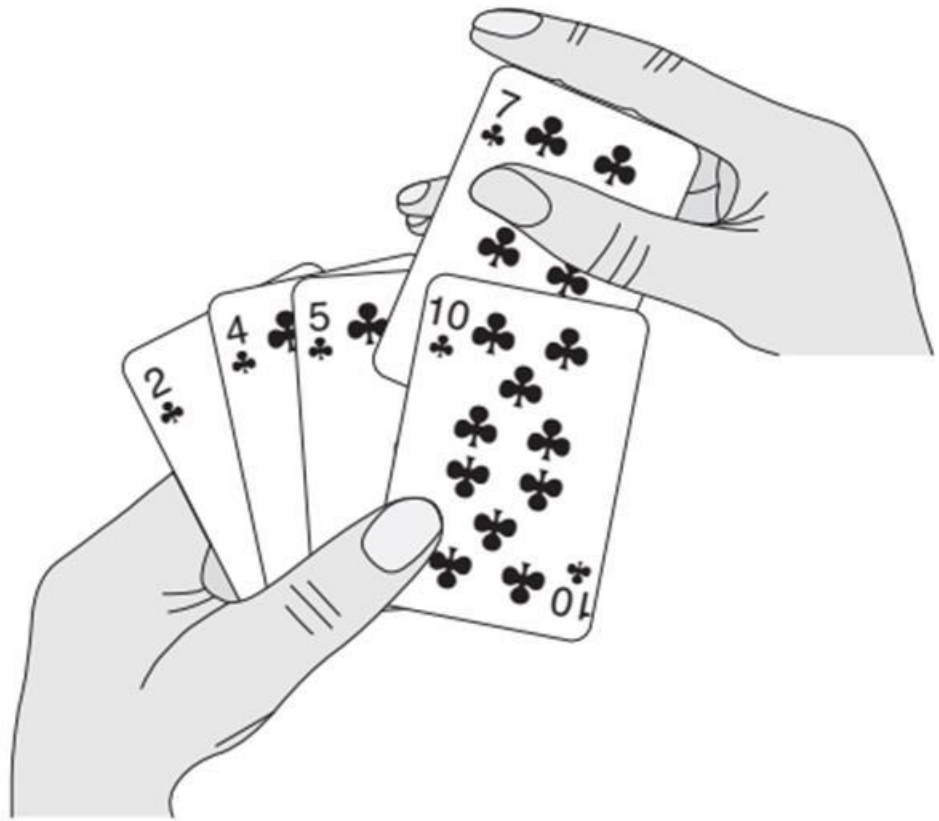
$$= \sum_{i=0}^{n-1}(n-i-1)$$

$$= n\sum_{i=0}^{n-1} 1 - \sum_{i=0}^{n-1} i - \sum_{i=0}^{n-1} 1$$

$$= n^2 - \frac{(n-1)n}{2} - n$$

$$= \Theta(n^2)$$

Insertion Sort



Insertion Sort

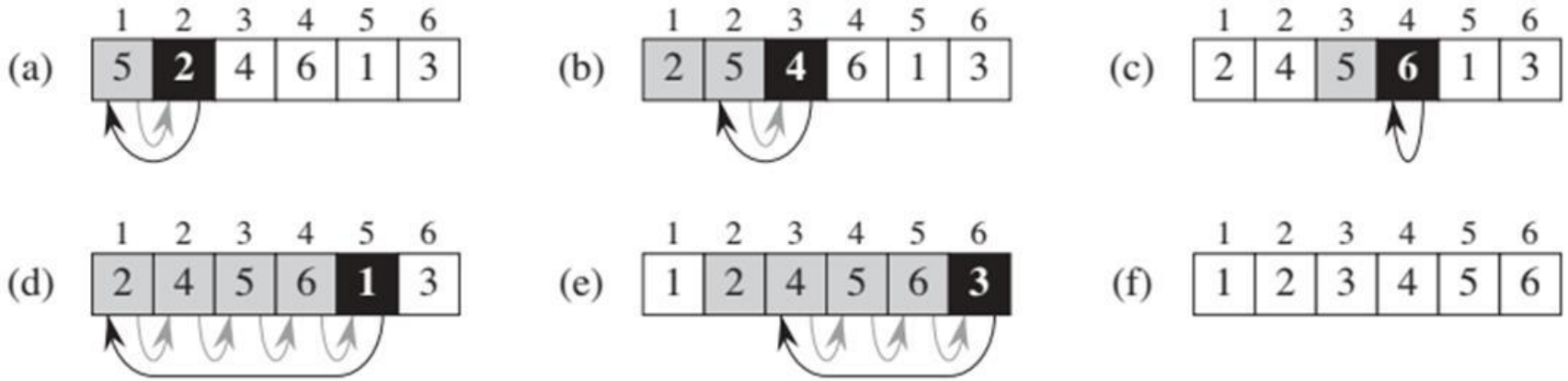


Figure The operation of INSERTION-SORT on the array $A = \langle 5, 2, 4, 6, 1, 3 \rangle$. Array indices appear above the rectangles, and values stored in the array positions appear within the rectangles. **(a)–(e)** The iterations of the **for** loop of lines 1–8. In each iteration, the black rectangle holds the key taken from $A[j]$, which is compared with the values in shaded rectangles to its left in the test of line 5. Shaded arrows show array values moved one position to the right in line 6, and black arrows indicate where the key moves to in line 8. **(f)** The final sorted array.

Insertion Sort

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

Class Exercise #2

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

$$\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$$

and

$$\sum_{j=2}^{n}(j-1) = \frac{n(n-1)}{2}$$

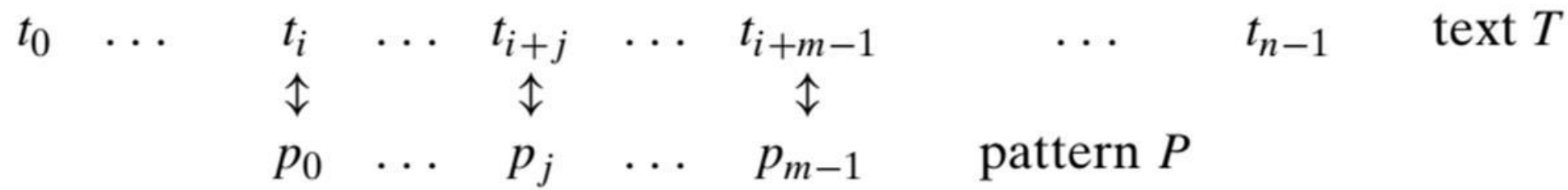
$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1)$$

$O(n^2)$

String searching and Pattern matching

- Given a string of *n* characters called the ***text*** and a string of *m* characters *(m ≤ n)* called the ***pattern***, find a substring of the text that matches the pattern.
- To put it more precisely, we want to find *i*—the index of the leftmost character of the first matching substring in the text—such that

$$t_i = p_0, \ldots, t_{i+j} = p_j, \ldots, t_{i+m-1} = p_{m-1}$$



Applications

- Web Searches
- Database Queries
- Detecting Plagiarism

String searching and Pattern matching

- If matches other than the first one need to be found, a string-matching algorithm can simply continue working until the entire text is exhausted.
- A brute-force algorithm for the string-matching problem is quite obvious:
  - Align the pattern against the first *m* characters of the text.
  - Start matching the corresponding pairs of characters from left to right until either all the *m* pairs of the characters match (then the algorithm can stop) or a mismatching pair is encountered.
  - In the latter case, shift the pattern one position to the right and resume the character comparisons, starting again with the first character of the pattern and its counterpart in the text.

String searching and Pattern matching

- Note that the last position in the text that can still be a beginning of a matching substring is *n* - *m* (provided the text positions are indexed from 0 to *n* - 1).
- Beyond that position, there are not enough characters to match the entire pattern; hence, the algorithm need not make any comparisons there.

```
N  O  B  O  D  Y  _  N  O  T  I  C  E  D  _  H  I  M
N  O  T
   N  O  T
      N  O  T
         N  O  T
            N  O  T
               N  O  T
                  N  O  T
                     N  O  T
                        N  O  T
```

Example of brute-force string matching. The pattern's characters that are compared with their text counterparts are in bold type.

String searching and Pattern matching

ALGORITHM *BruteForceStringMatch*$(T[0..n-1],\ P[0..m-1])$

```
//Implements brute-force string matching
//Input: An array T[0..n − 1] of n characters representing a text and
//       an array P[0..m − 1] of m characters representing a pattern
//Output: The index of the first character in the text that starts a
//        matching substring or −1 if the search is unsuccessful
for i ← 0 to n − m do
    j ← 0
    while j < m and P[j] = T[i + j] do
        j ← j + 1
    if j = m return i
return −1
```

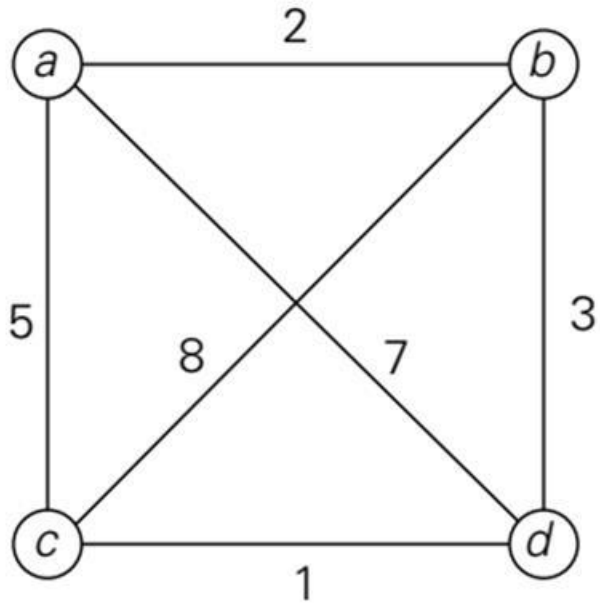
String searching and Pattern matching

- The algorithm shifts the pattern almost always after a single character comparison.
- The worst case is much worse: the algorithm may have to make all *m* comparisons before shifting the pattern, and this can happen for each of the *n* - *m* + 1 tries.
- Thus, in the worst case, the algorithm makes *m(n* - *m* + 1*)* character comparisons, which puts it in the *O(nm)* class.
- For a typical word search in a natural language text, however, we should expect that most shifts would happen after very few comparisons.
- Therefore, the average-case efficiency should be considerably better than the worst-case efficiency. Indeed it is: for searching in random texts, it has been shown to be linear, i.e.,O*(n+m)*.

Exhaustive Search : *Traveling salesman problem (TSP)*

- The ***traveling salesman problem (TSP)*** has been intriguing researchers for the last 150 years by its seemingly simple formulation, important applications, and interesting connections to other combinatorial problems.
- The problem asks to find the shortest tour through a given set of *n* cities that visits each city exactly once before returning to the city where it started.
- The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances. Then the problem can be stated as the problem of finding the shortest ***Hamiltonian circuit*** of the graph.

Introduction



Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

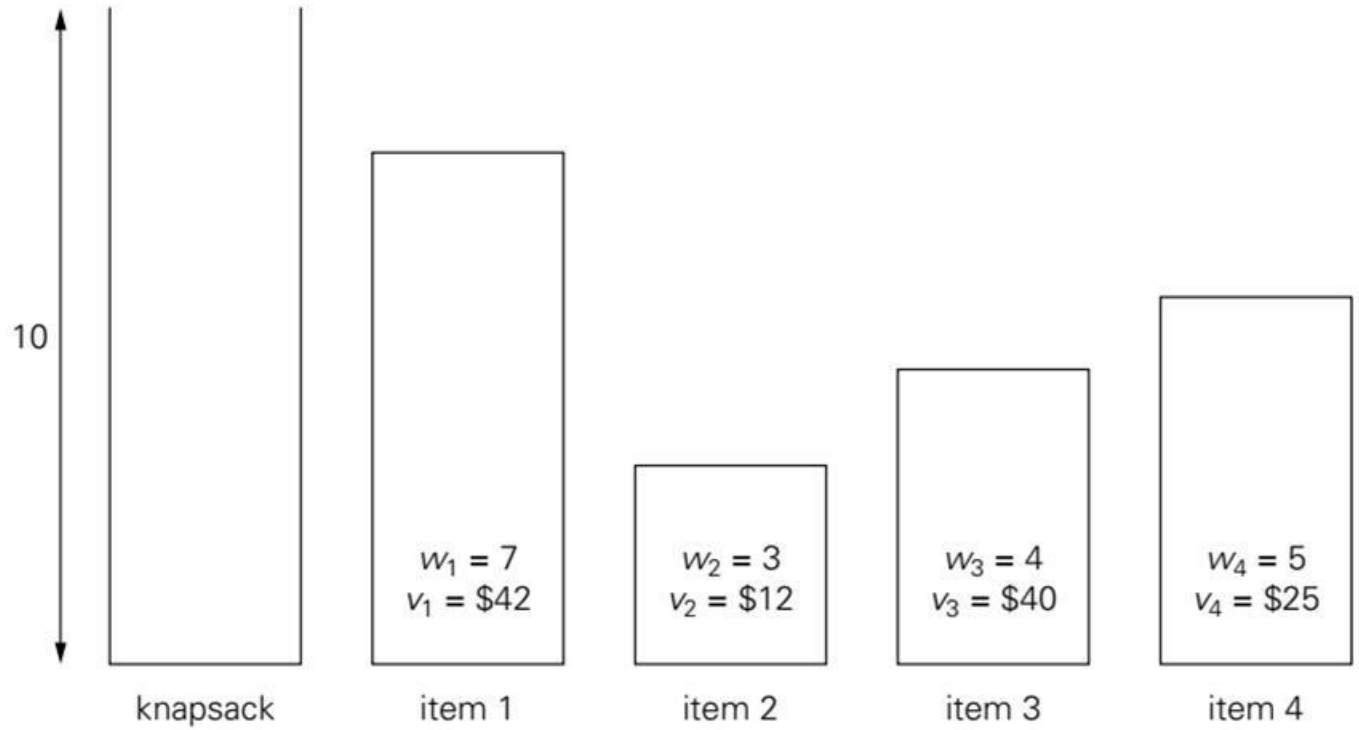
Brute Force and Exhaustive Search

- Thus, we can get all the tours by generating all the permutations of n - 1 intermediate cities, compute the tour lengths, and find the shortest among them i.e. $(n - 1)!$ which makes the exhaustive-search approach impractical for all but very small values of n.
- In fact, these problem is the best-known examples of so called ***NP-hard problems***.
- No polynomial-time algorithm is known for any *NP*-hard problem. Hence, approximation algorithms

Knapsack problem

A thief breaks into a museum. Fabulous paintings, sculptures, and jewels are everywhere. The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector's market. But, the thief has only brought a single knapsack to the scene of the robbery, and can take away only what he can carry. What items should the thief take to maximize the haul?

Knapsack problem



(a)

Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

(b)

Knapsack problem

- The exhaustive-search approach to this problem leads to generating all the
  subsets of the set of *n* items given, computing the total weight of each subset in order to identify feasible subsets (i.e., the ones with the total weight not exceeding the knapsack capacity), and finding a subset of the largest value among them.
- As an example, the solution to the instance is given in Figure.
- Since the number of subsets of an *n*-element set is 2^n, the exhaustive search leads to a $O(2^n)$ algorithm, no matter how efficiently individual subsets are generated.

Assignment Problem

- Assigning *n* people to *n* jobs so that the total cost of the assignment is as small as possible.
- Select one element in each row of the matrix so that no two selected elements are in the same column and their sum is the smallest possible.

$$C = \begin{array}{c} \begin{matrix} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{matrix} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{matrix} \\ \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{matrix}$$

Assignment Problem

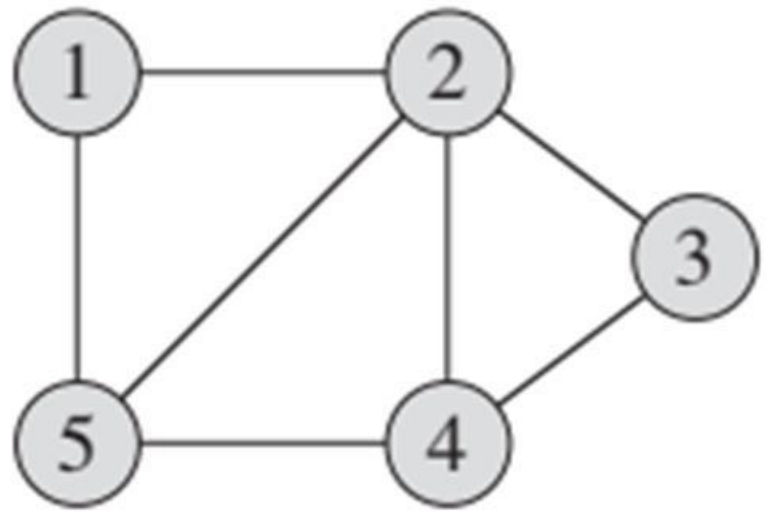
$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

<1, 2, 3, 4> cost = 9 + 4 + 1 + 4 = 18
<1, 2, 4, 3> cost = 9 + 4 + 8 + 9 = 30
<1, 3, 2, 4> cost = 9 + 3 + 8 + 4 = 24
<1, 3, 4, 2> cost = 9 + 3 + 8 + 6 = 26
<1, 4, 2, 3> cost = 9 + 7 + 8 + 9 = 33
<1, 4, 3, 2> cost = 9 + 7 + 1 + 6 = 23

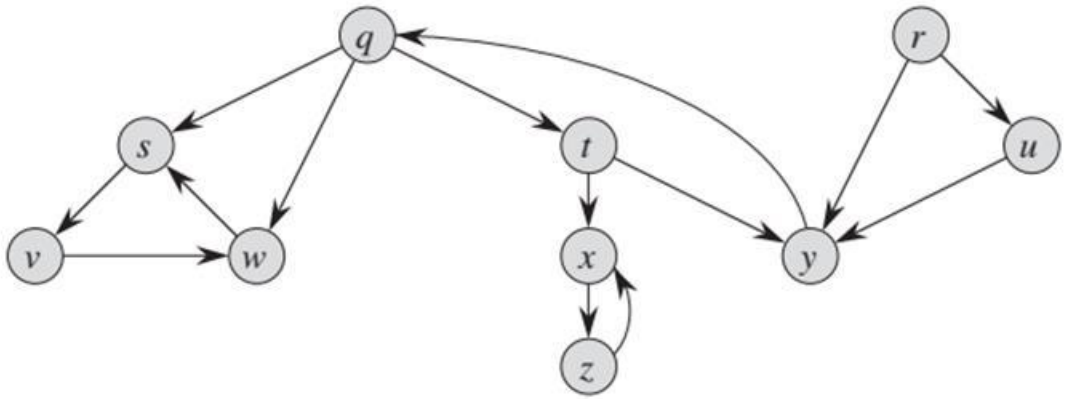
etc.

Graph

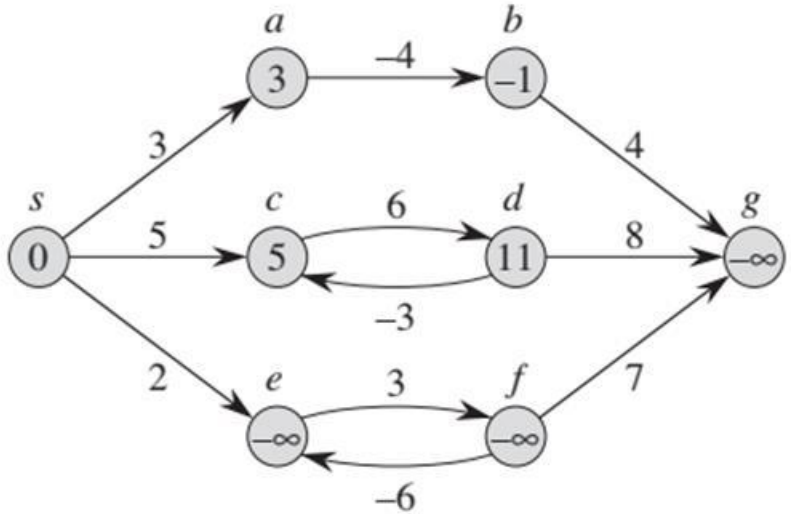
- **Graph :** A graph G consists of two sets:
  - A set V, called set of all vertices (or nodes)
  - A set E, called set of all edges (or arcs).
  - Example
    - V={1,2,3,4,5},
    - E={((1,2), (2,3), (3,4), (4,5), (5,1), (2,5), (2,4)}



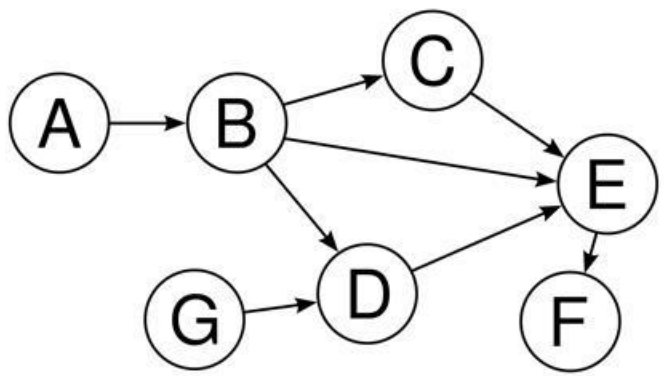
Graph Variants



Directed Graph/Digraph



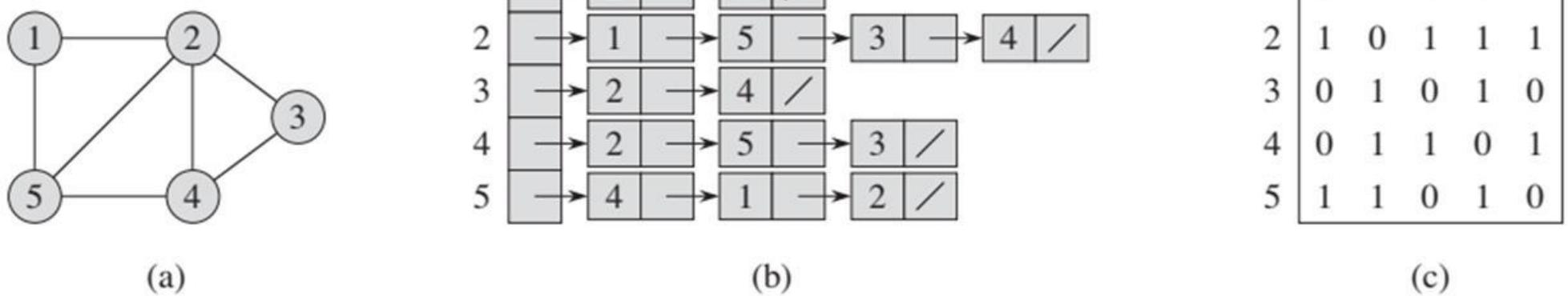
Weighted Digraph



Acyclic Graph

Graph Representation

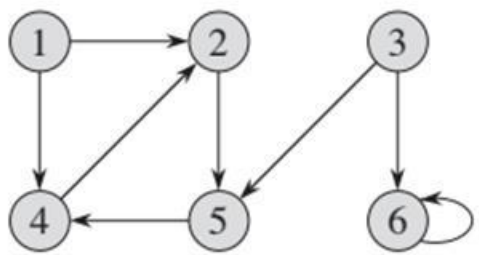
- Two standard ways to represent a graph G =(V, E), as a collection of adjacency lists or as an adjacency matrix.



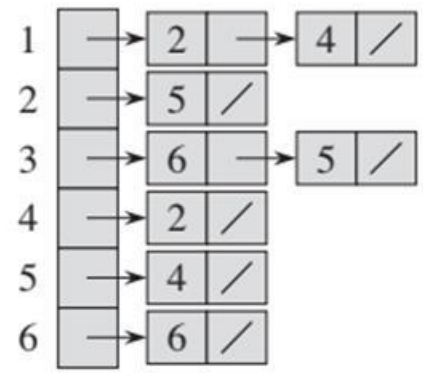
Two representations of an undirected graph. **(a)** An undirected graph G with 5 vertices and 7 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Graph Representation

- Two standard ways to represent a graph G =(V, E), as a collection of adjacency lists or as an adjacency matrix.



(a)



(b)

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

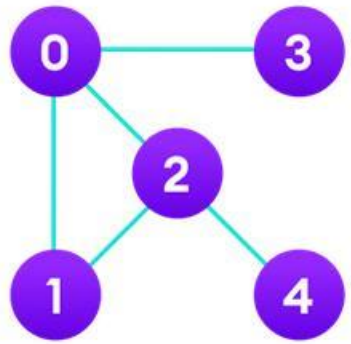
(c)

Two representations of a directed graph. **(a)** A directed graph G with 6 vertices and 8 edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Searching and Traversal of Graphs

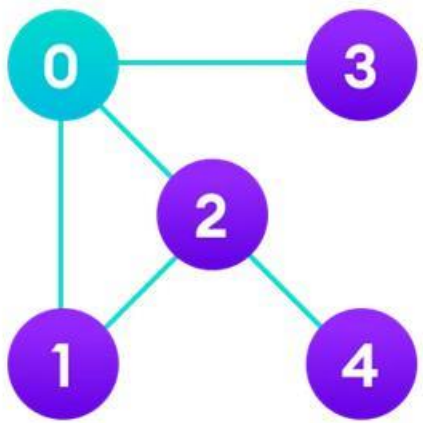
- DFS – Depth First Search Traversal
- BFS – Breadth First Search Traversal

Depth First Search [DFS]



Visited

Stack



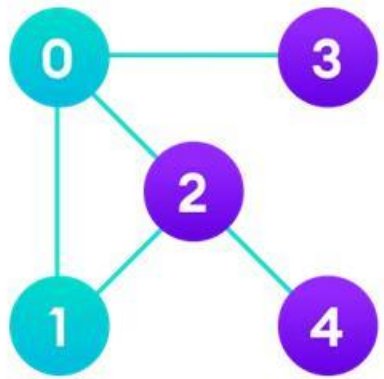
0				

Visited

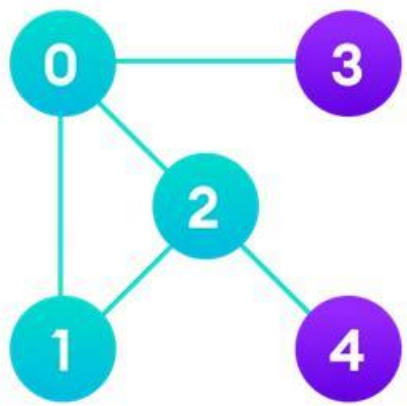
1	2	3		

Stack

Depth First Search [DFS]

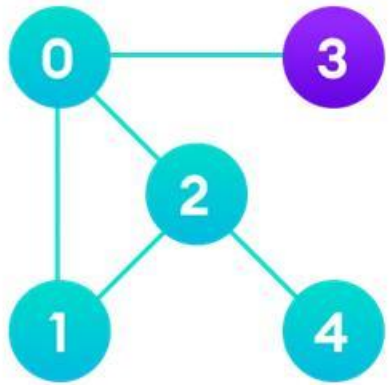


0	1				Visited
2	3				Stack



0	1	2			Visited
4	3				Stack

Depth First Search [DFS]

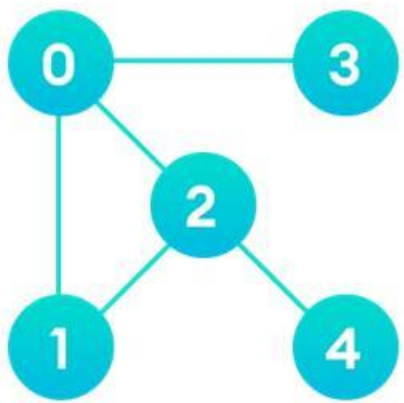


0	1	2	4	

Visited

3				

Stack



0	1	2	4	3

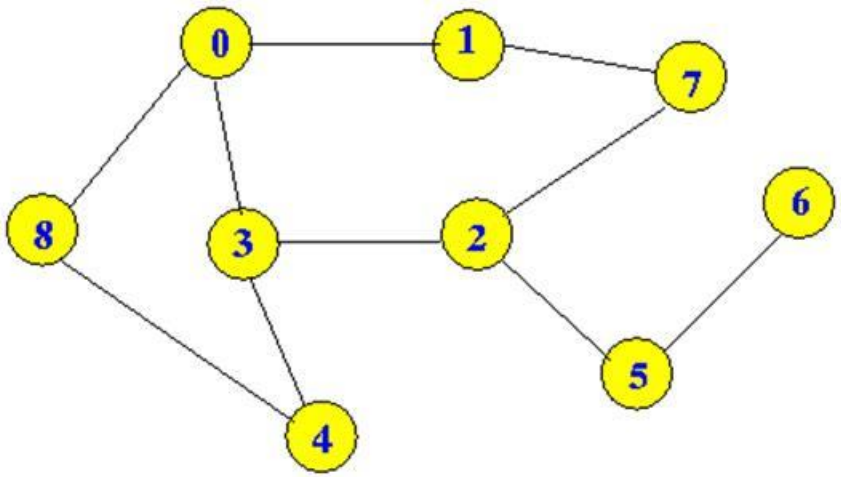
Visited

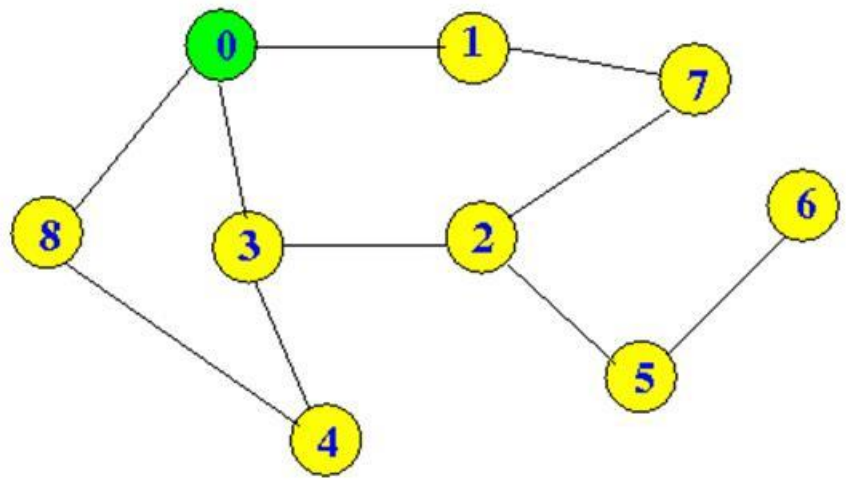
Stack

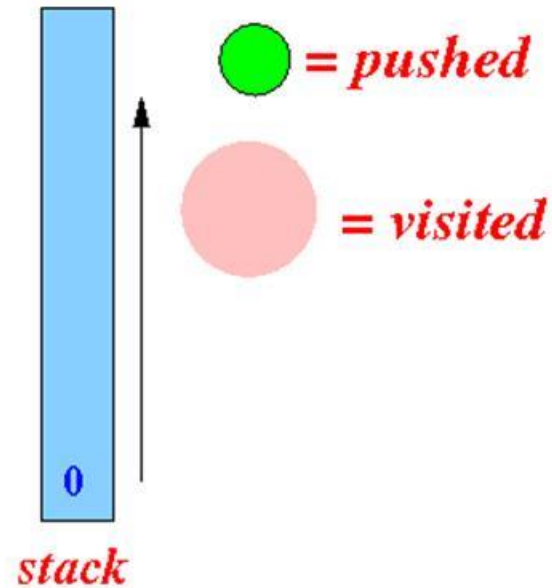
Pseudo code

```
Set all nodes to "not visited";

s = new Stack();      ******* Change to use a stack

s.push(initial node);     ***** Push() stores a value in a stack

while ( s ≠ empty ) do
{
   x = s.pop();          ****** Pop() remove a value from the stack

   if ( x has not been visited )
   {
      visited[x] = true;          // Visit node x !

      for ( every edge (x, y)  /* we are using all edges ! */ )
         if ( y has not been visited )
            s.push(y);        ***** Use push() !
   }
}
```

Depth First Search [DFS]

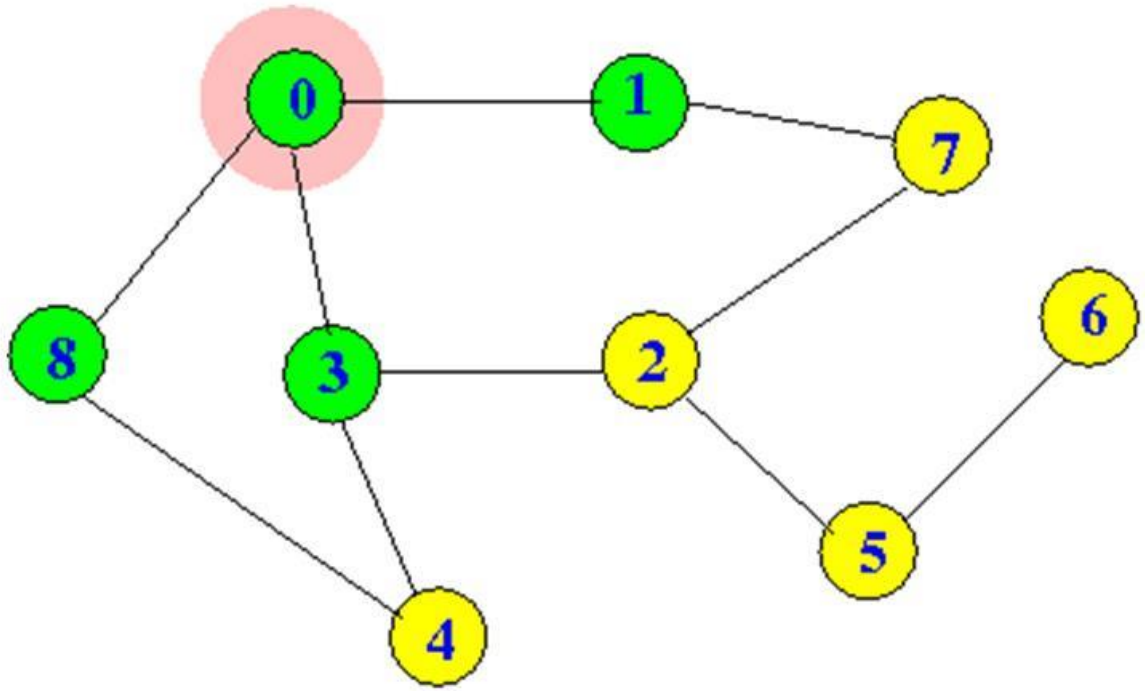


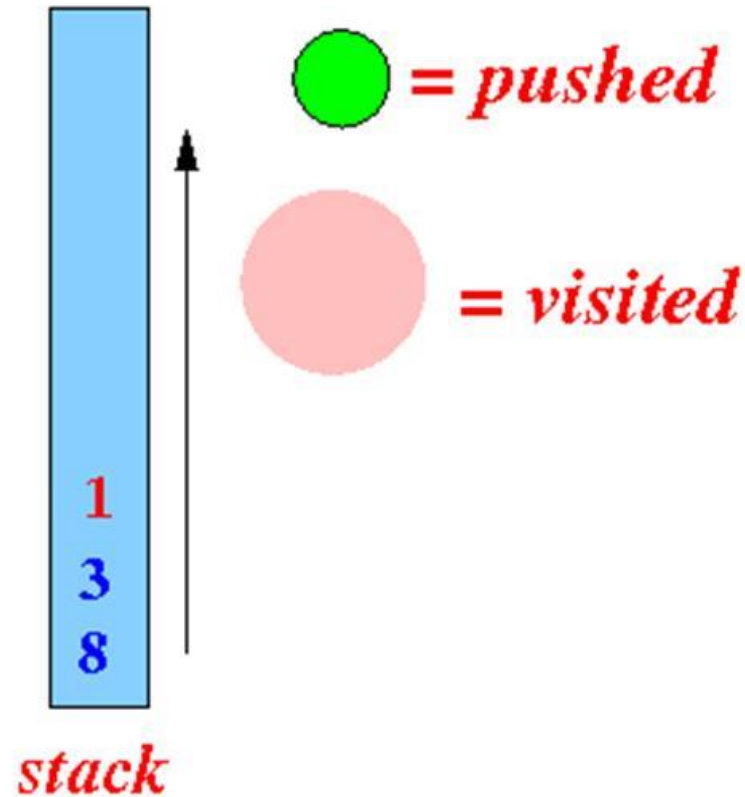




0 ->

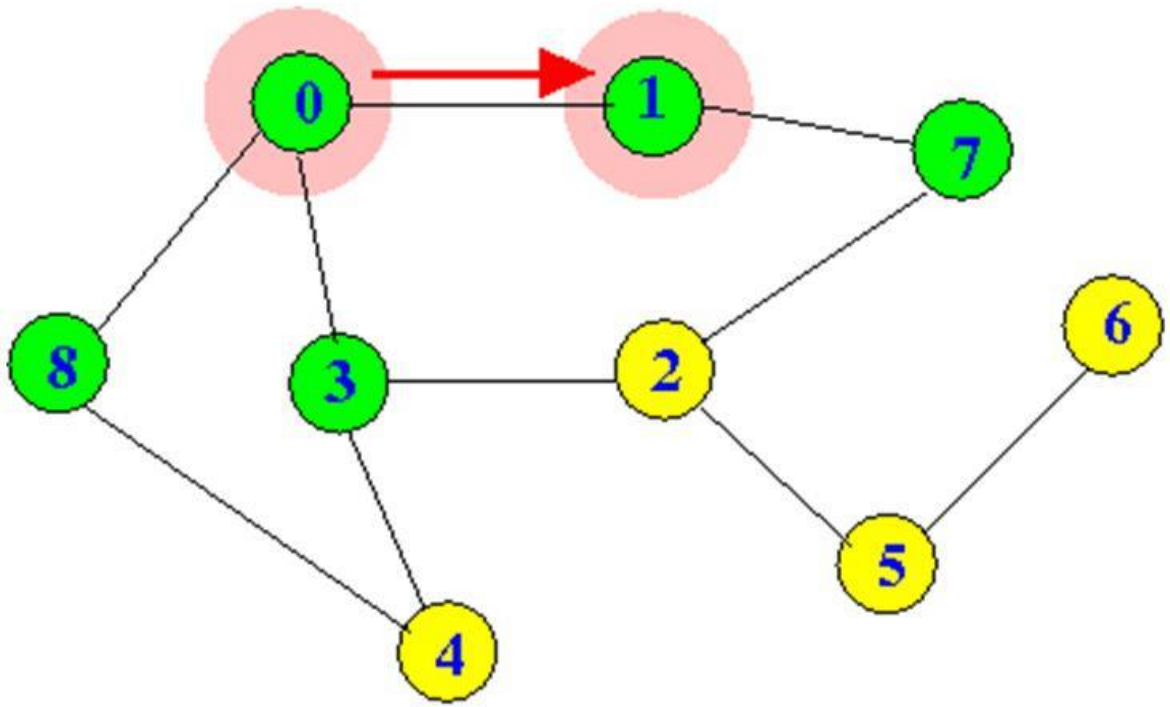
Depth First Search [DFS]

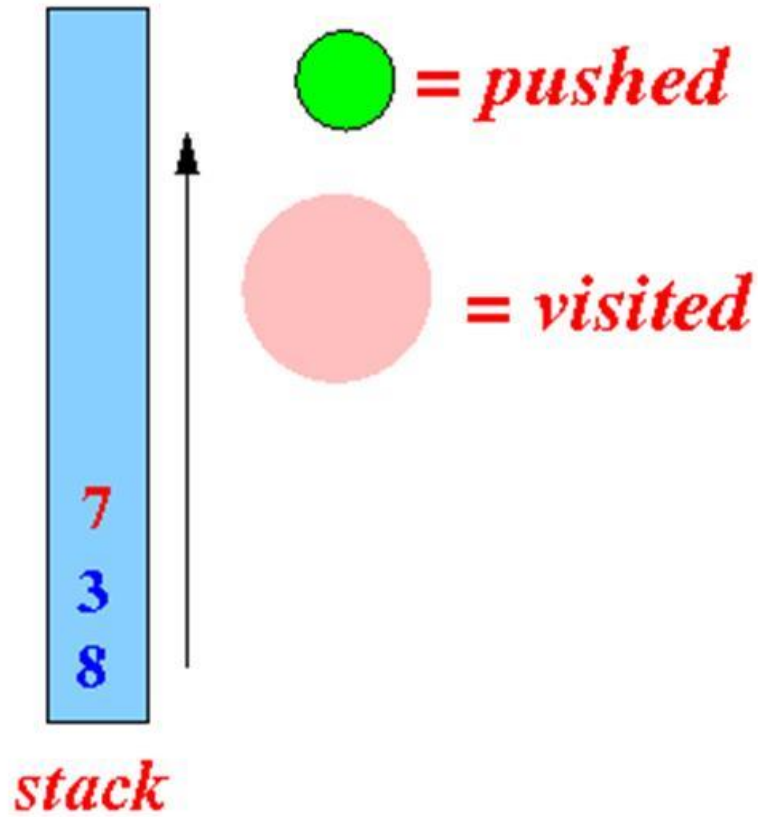




0 -> 1 ->

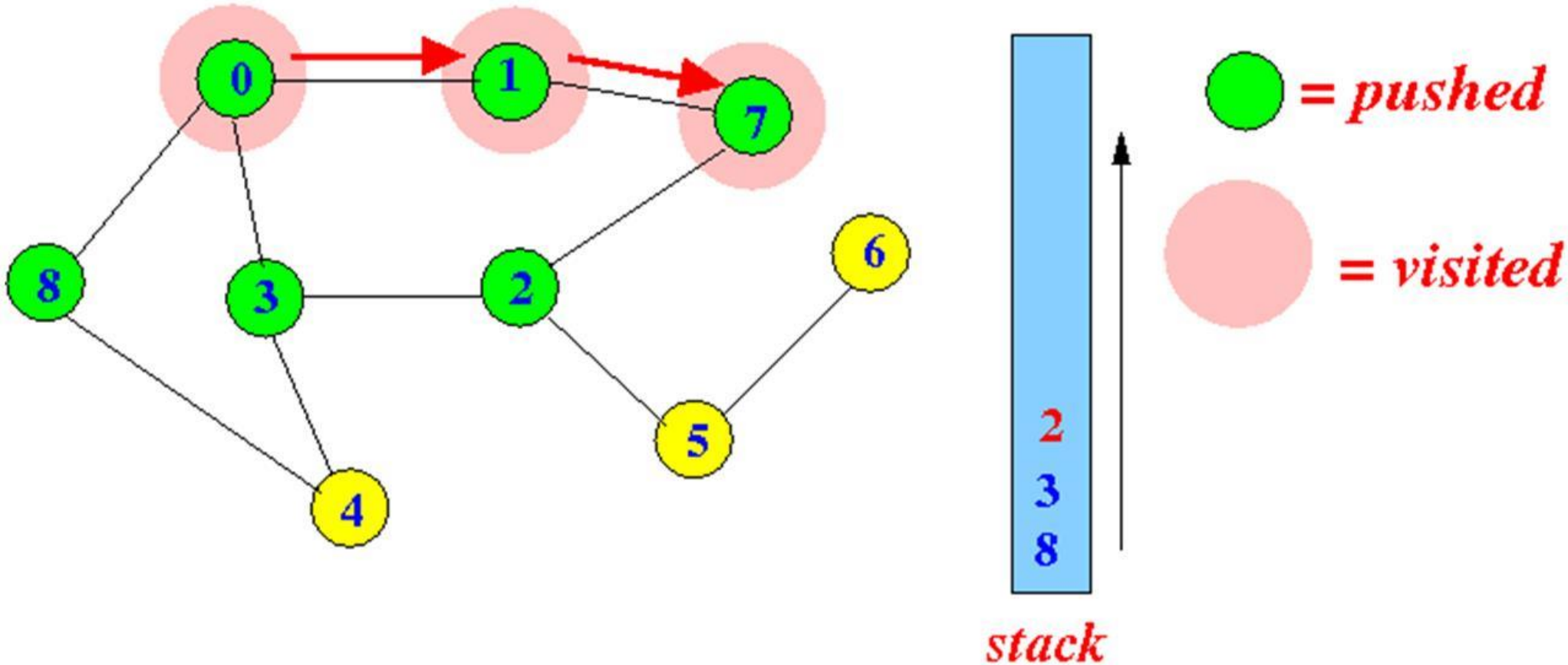
Depth First Search [DFS]





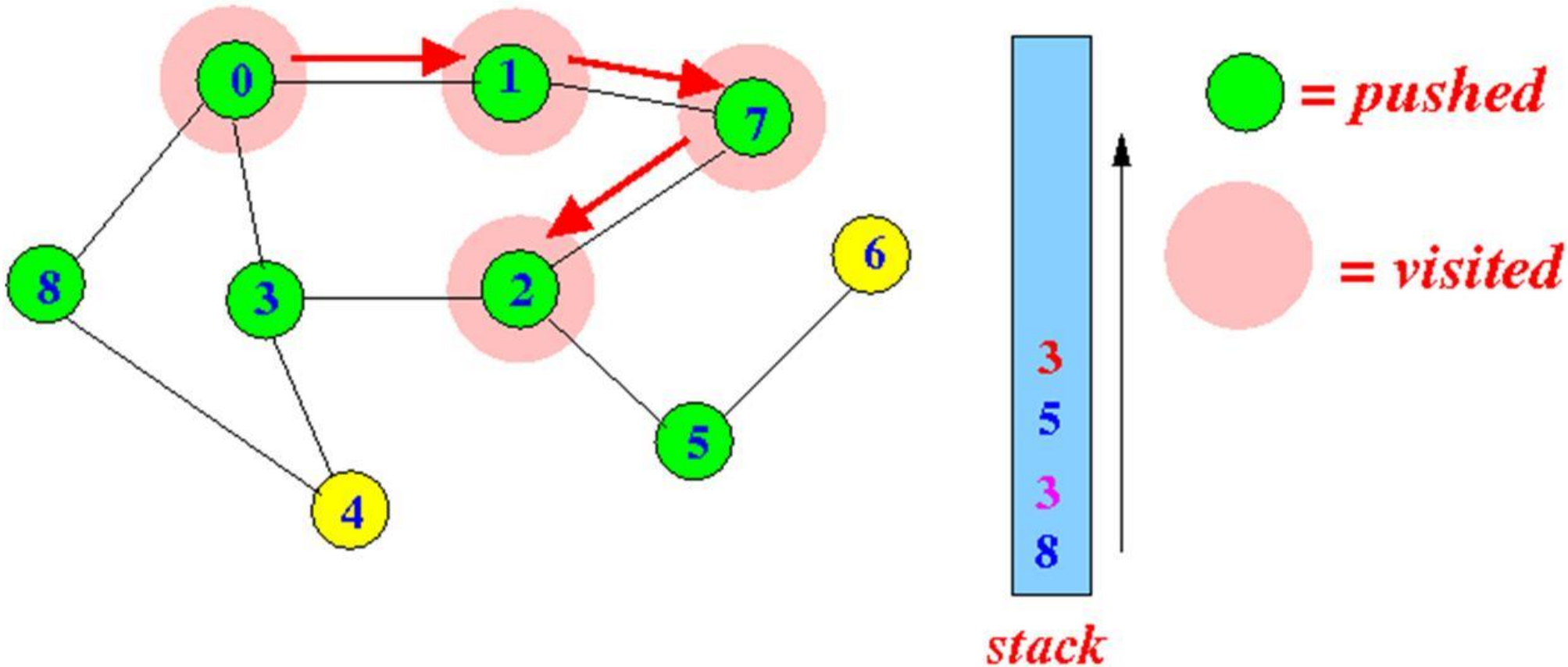
0 -> 1 -> 7 ->

Depth First Search [DFS]



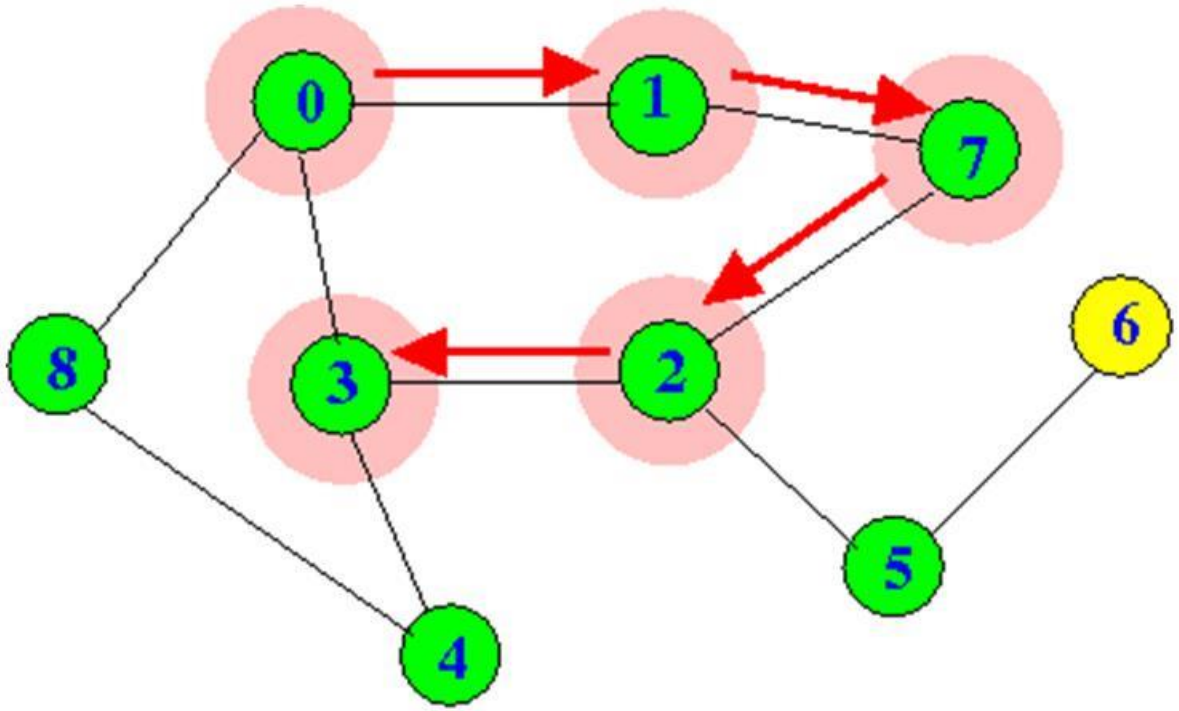
0 -> 1 -> 7 -> 2 ->

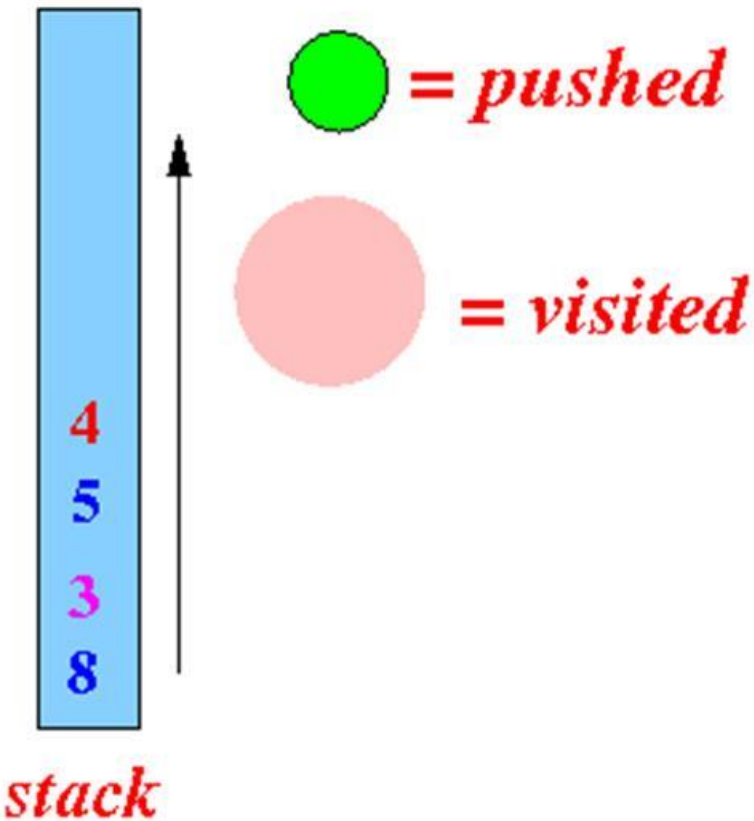
Depth First Search [DFS]



0 -> 1 -> 7 -> 2 -> 3 ->

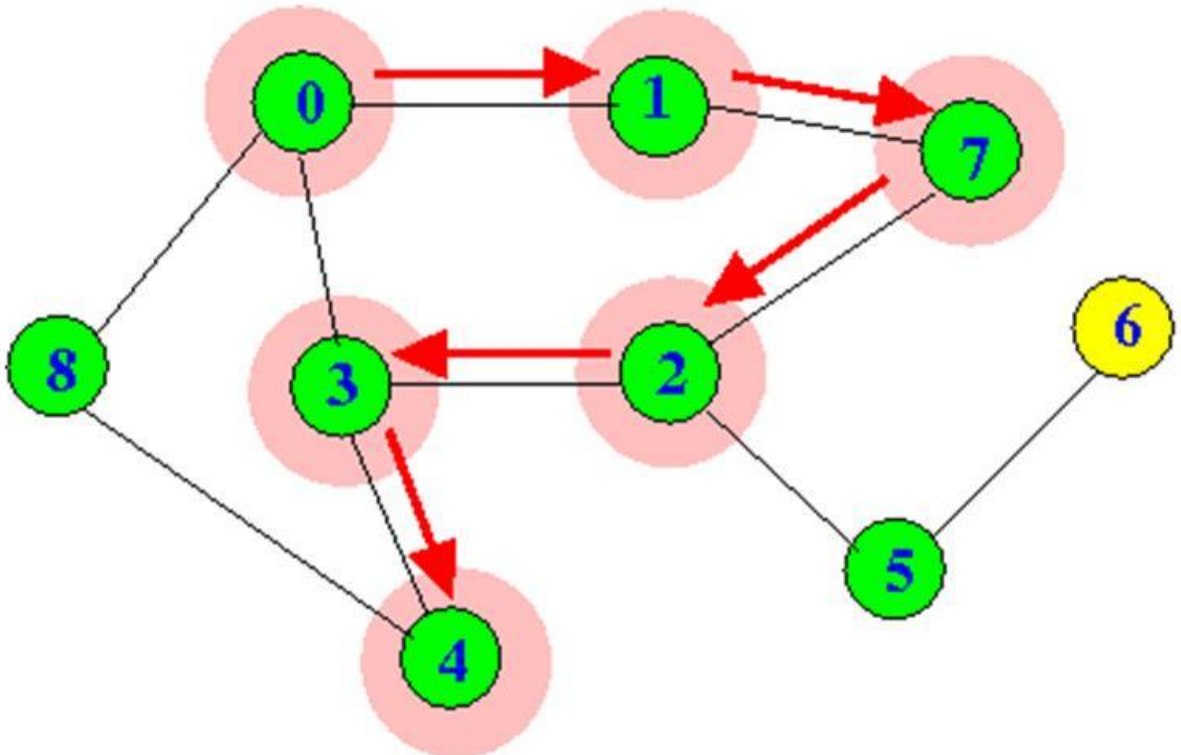
Depth First Search [DFS]

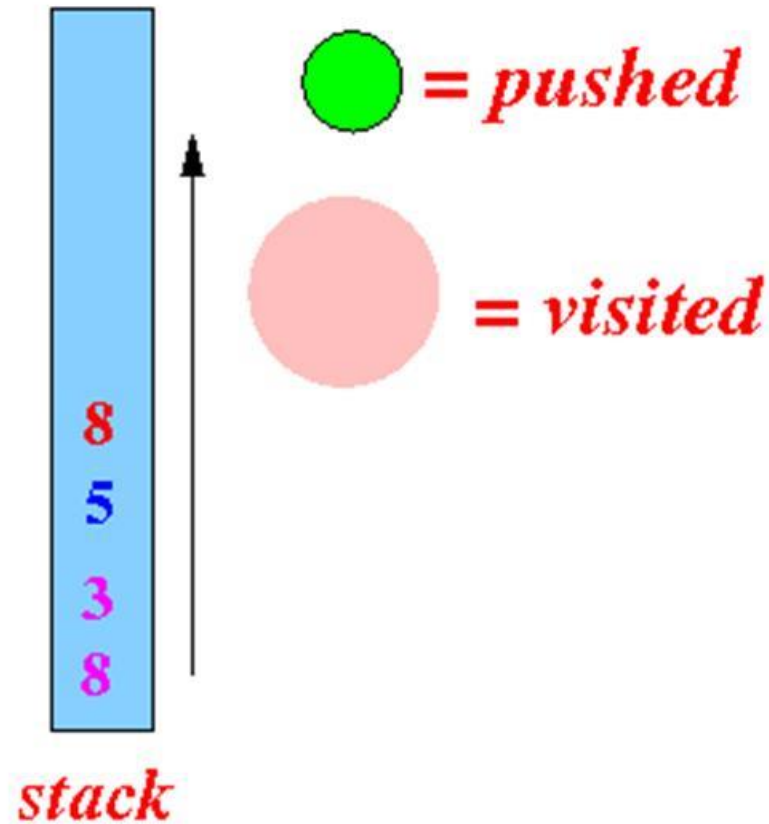




0 -> 1 -> 7 -> 2 -> 3 -> 4 ->

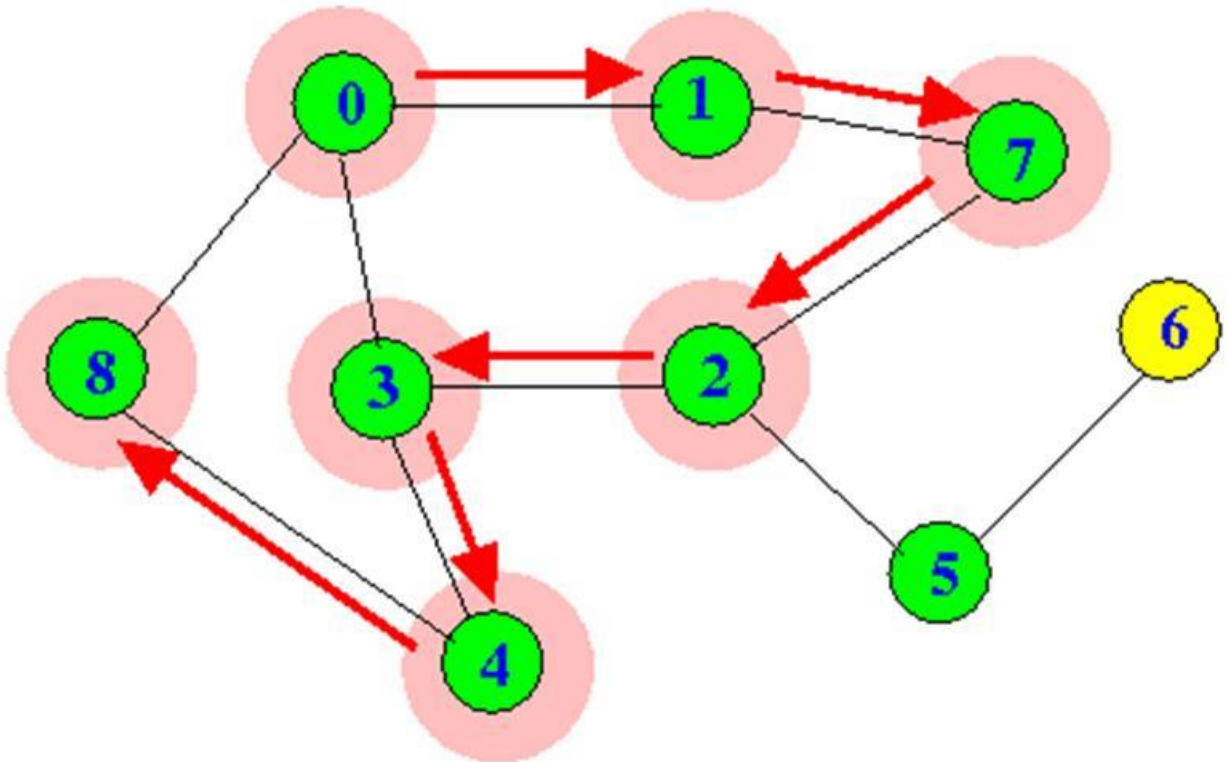
Depth First Search [DFS]

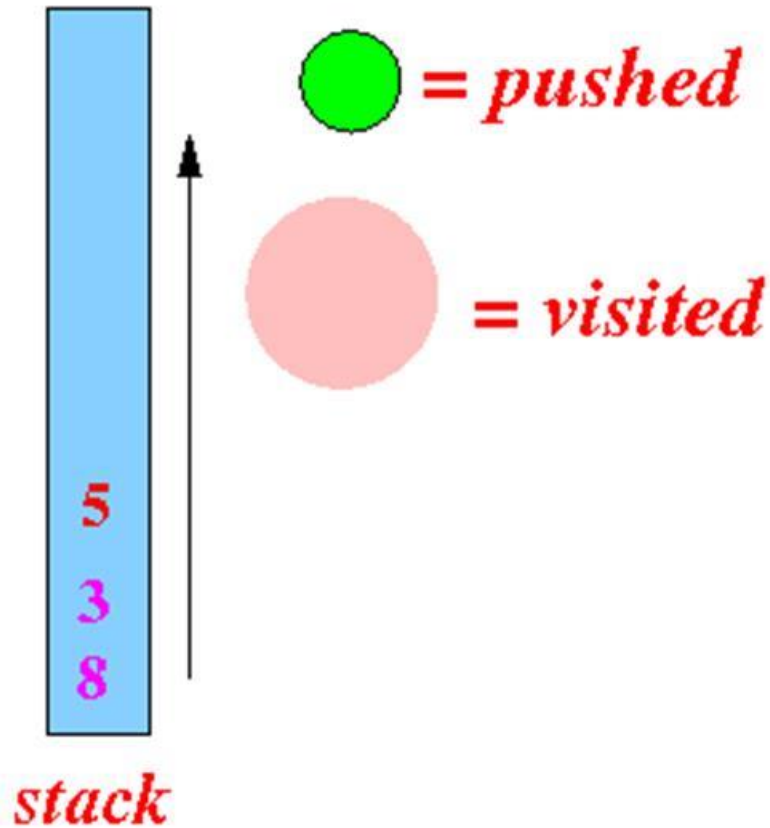




0 -> 1 -> 7 -> 2 -> 3 -> 4 -> 8 ->

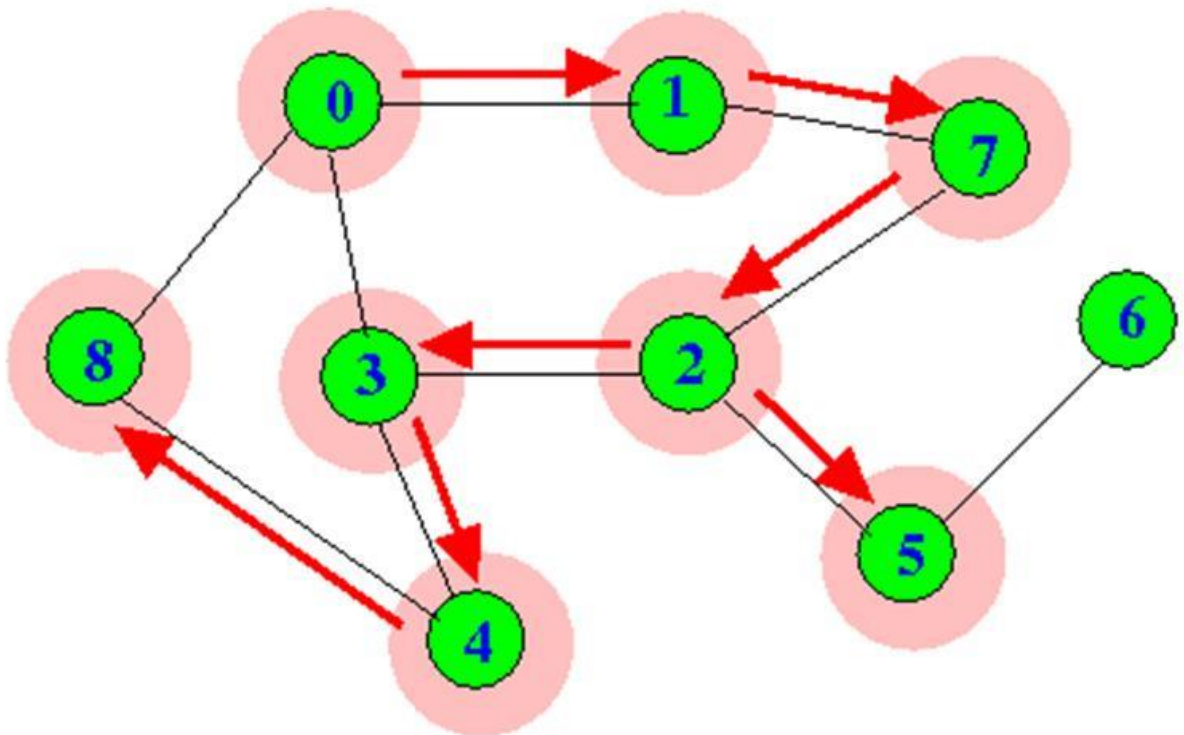
Depth First Search [DFS]

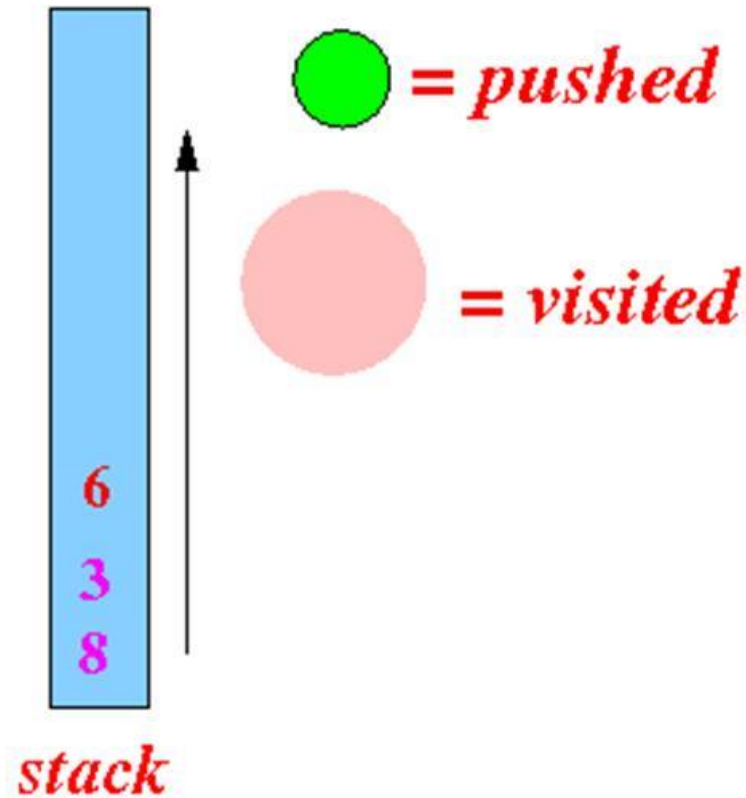




0 -> 1 -> 7 -> 2 -> 3 -> 4 -> 8 -> 5 ->

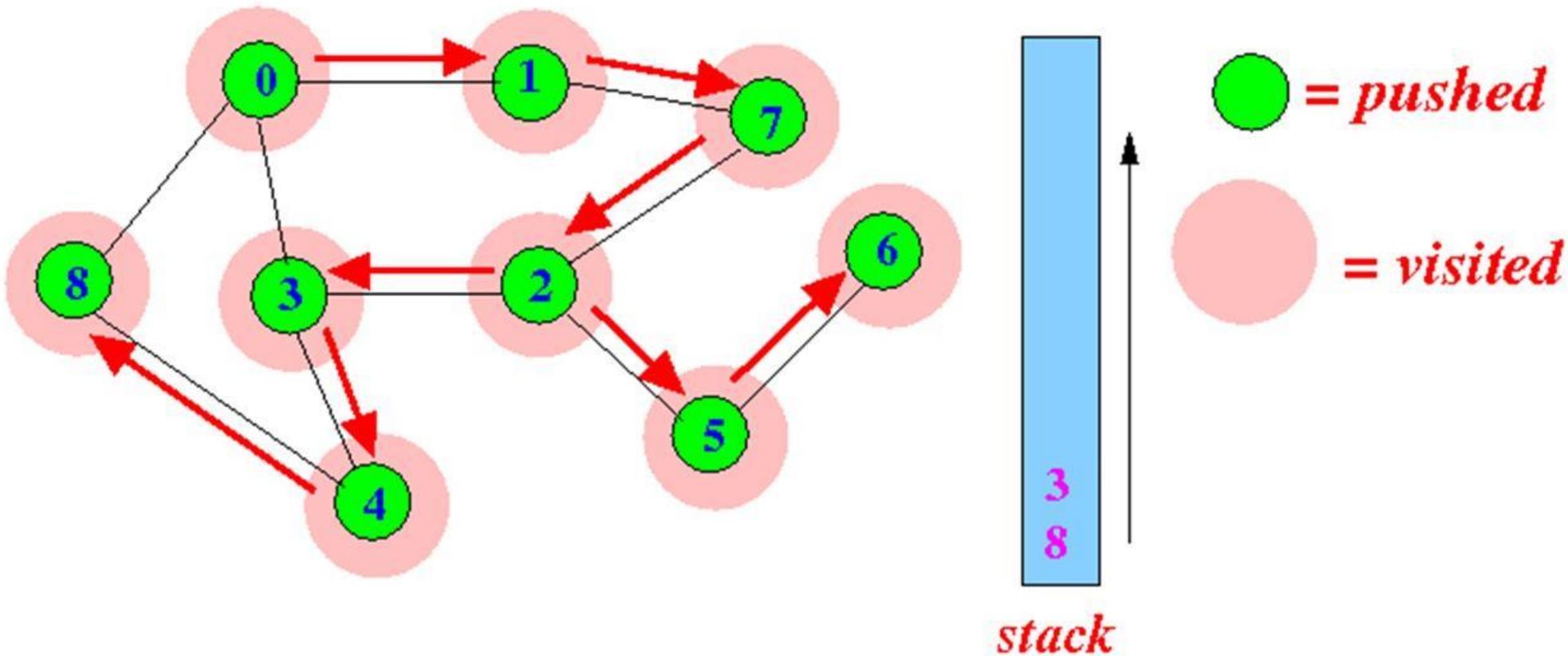
Depth First Search [DFS]





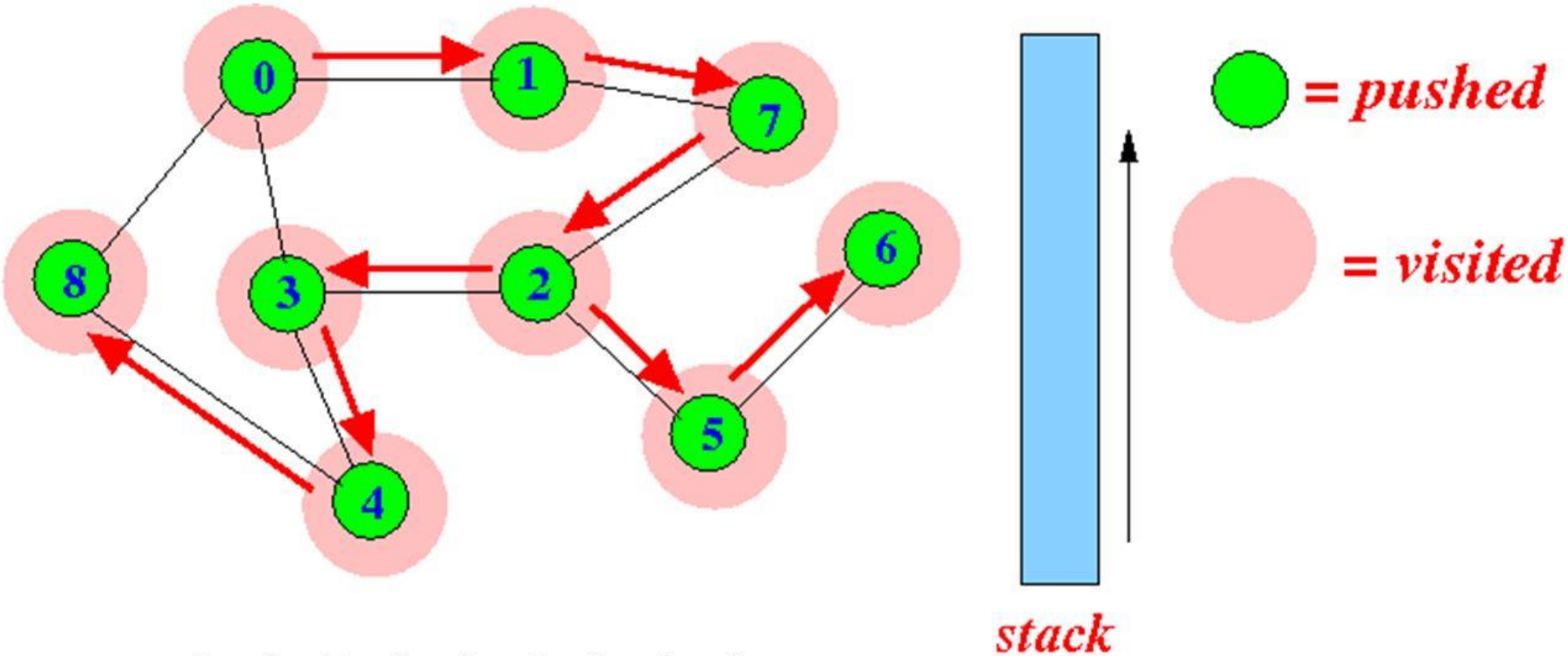
0 -> 1 -> 7 -> 2 -> 3 -> 4 -> 8 -> 5 -> 6

Depth First Search [DFS]



0 -> 1 -> 7 -> 2 -> 3 -> 4 -> 8 -> 5 -> 6

Depth First Search [DFS]

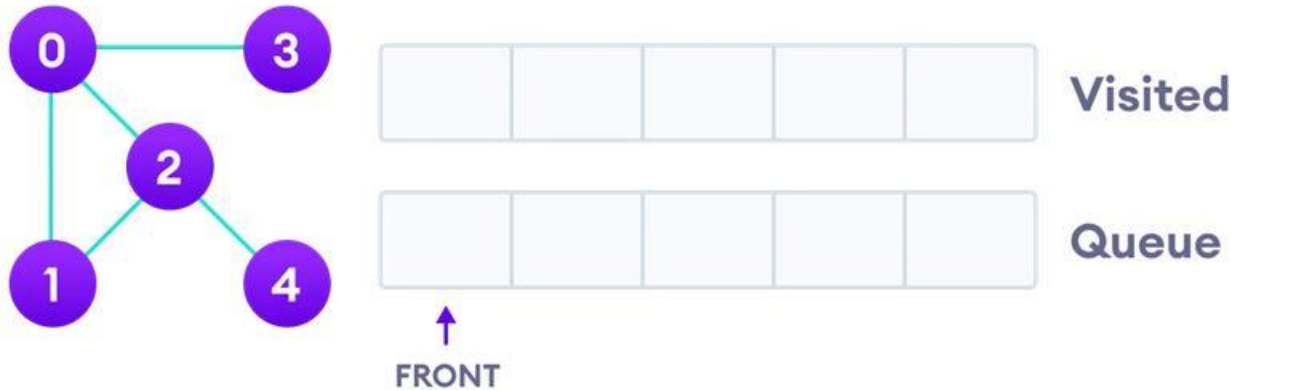


0 -> 1 -> 7 -> 2 -> 3 -> 4 -> 8 -> 5 -> 6

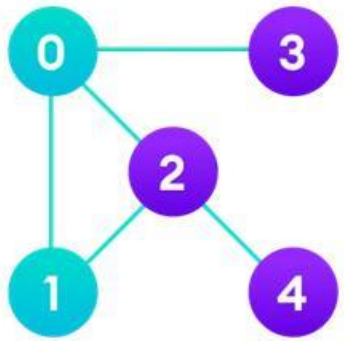
Depth First Search [DFS]

- The time complexity of DFS is O($V + E$), if we use adjacency lists for representing the graphs.
- This is because we are starting at a vertex and processing the adjacent nodes only if they are not visited.
- Similarly, if an adjacency matrix is used for a graph representation, then all edges adjacent to a vertex can't be found efficiently, and this gives O(V^2) complexity.
- **Applications of DFS**
  - Finding connected components
  - Solving puzzles such as mazes
  - Topological sorting

Breadth First Search [BFS]

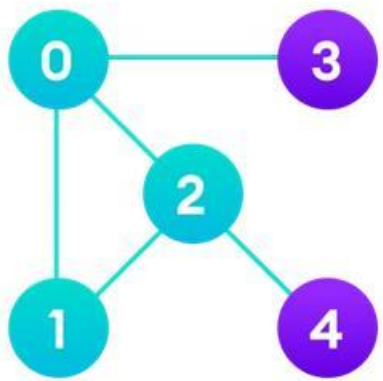


Breadth First Search [BFS]



0	1				Visited
2	3				Queue

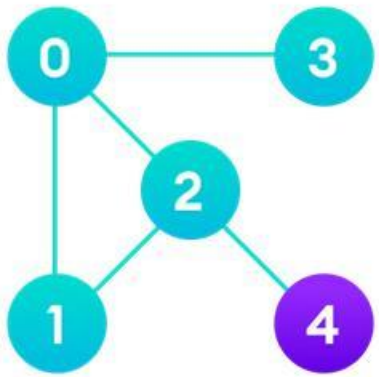
FRONT



0	1	2			Visited
3	4				Queue

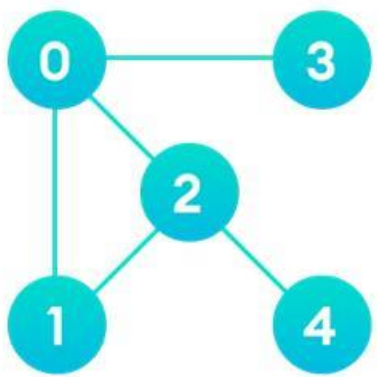
FRONT

Breadth First Search [BFS]



0	1	2	3		Visited
4					Queue

FRONT



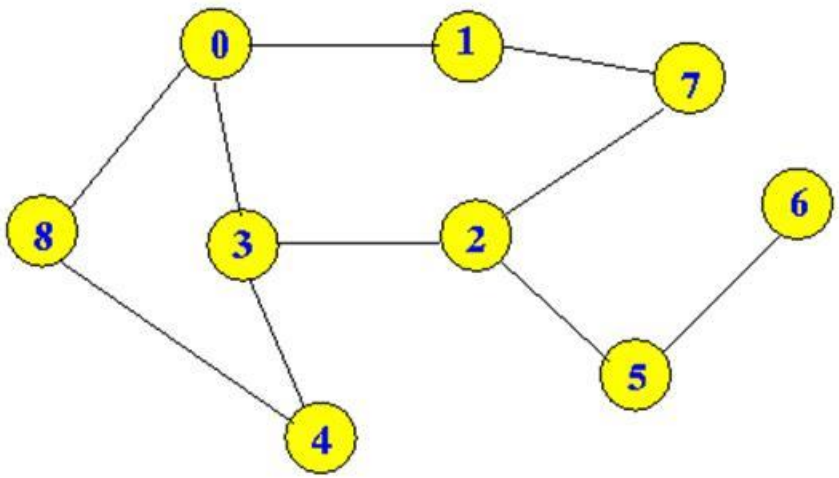
0	1	2	3	4	Visited
					Queue

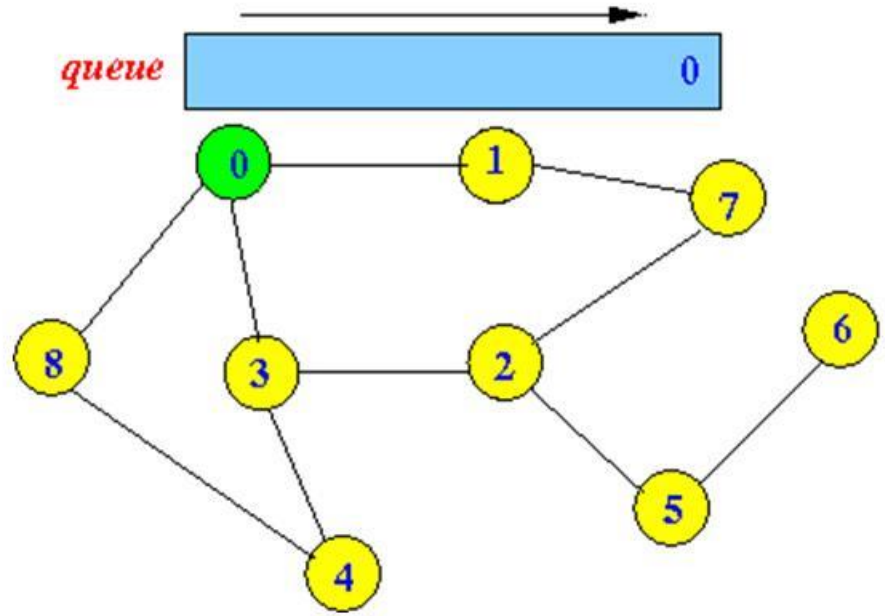
FRONT

Breadth First Search [BFS]

```
Set all nodes to "not visited";

q = new Queue();

q.enqueue(initial node);

while ( q ≠ empty ) do
{
   x = q.dequeue();

   if ( x has not been visited )
   {
      visited[x] = true;          // Visit node x !

      for ( every edge (x, y)  /* we are using all edges ! */ )
         if ( y has not been visited )
            q.enqueue(y);        // Use the edge (x,y) !!!
   }
}
```

Breadth First Search [BFS]



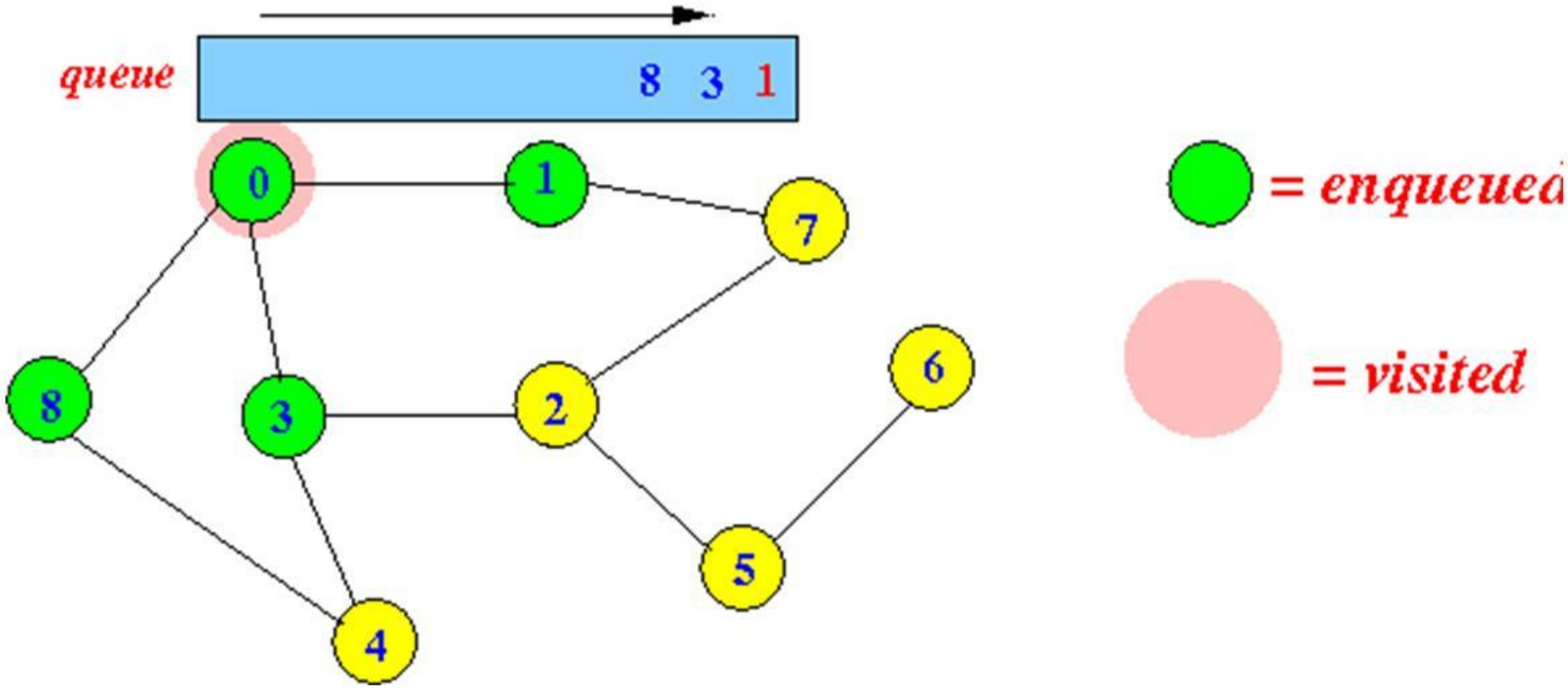


= enqueued

= visited

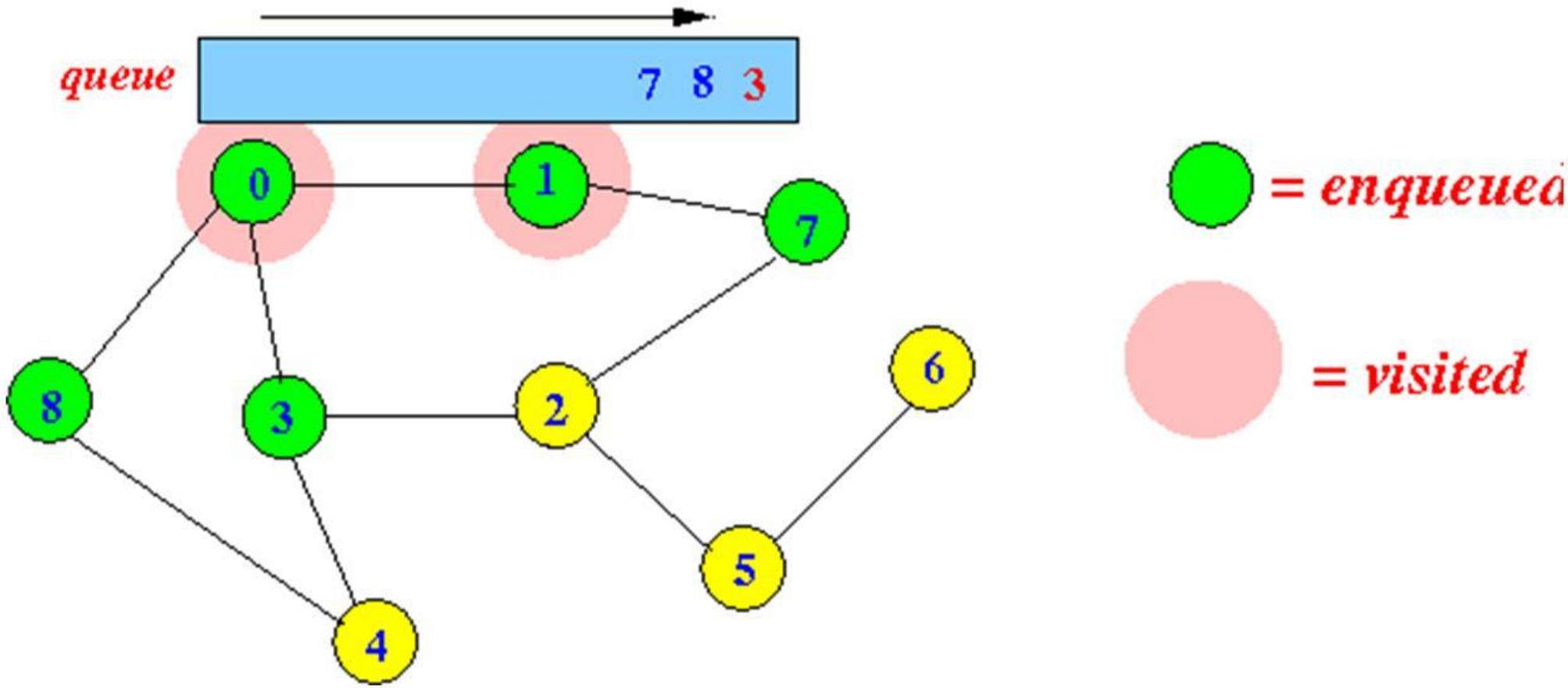
0 ->

Breadth First Search [BFS]



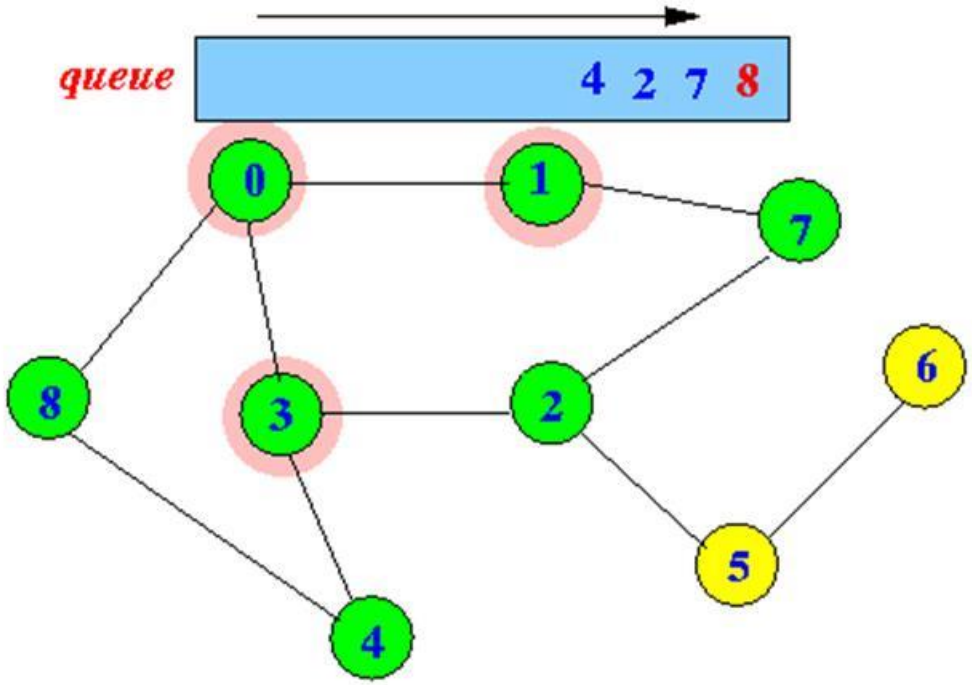
0 -> 1

Breadth First Search [BFS]

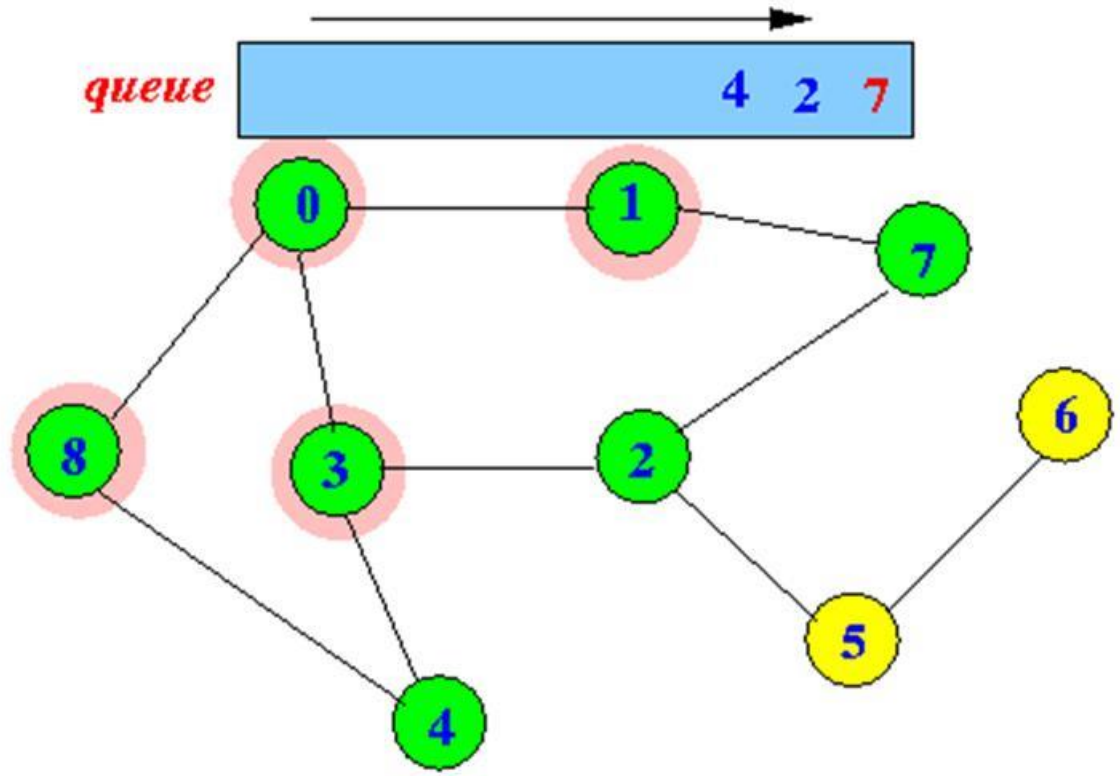


0 -> 1 -> 3 ->

Breadth First Search [BFS]

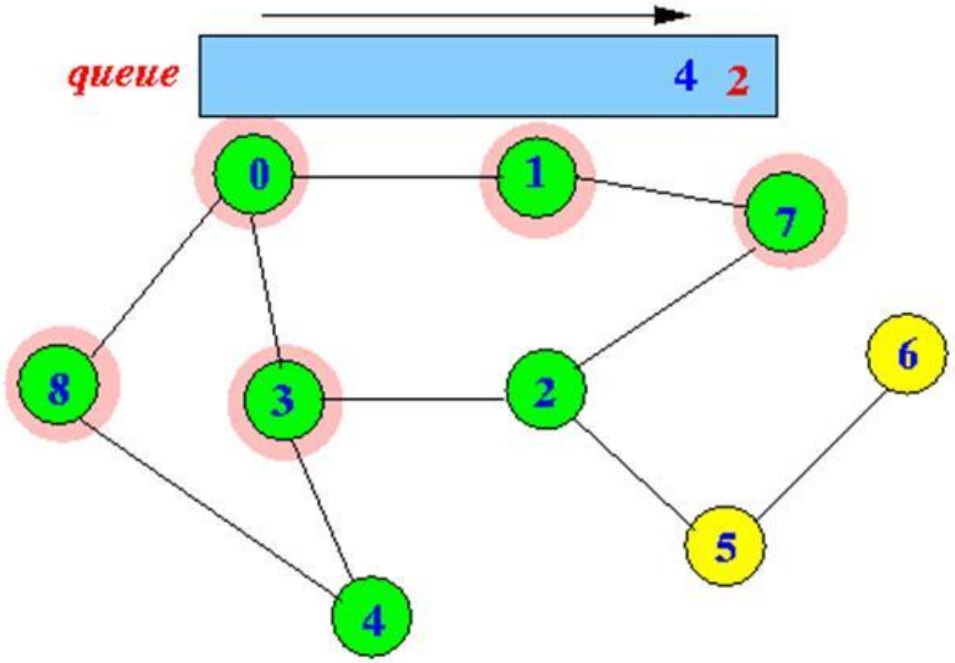


0 -> 1 -> 3 -> 8 ->

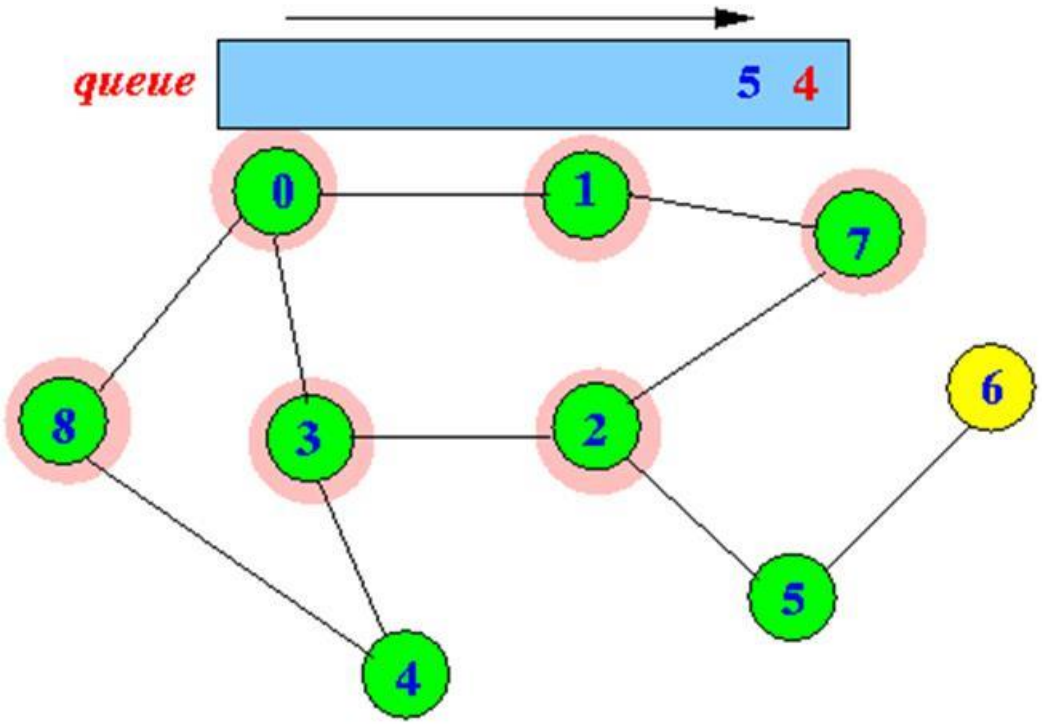


0 -> 1 -> 3 -> 8 -> 7 ->

Breadth First Search [BFS]

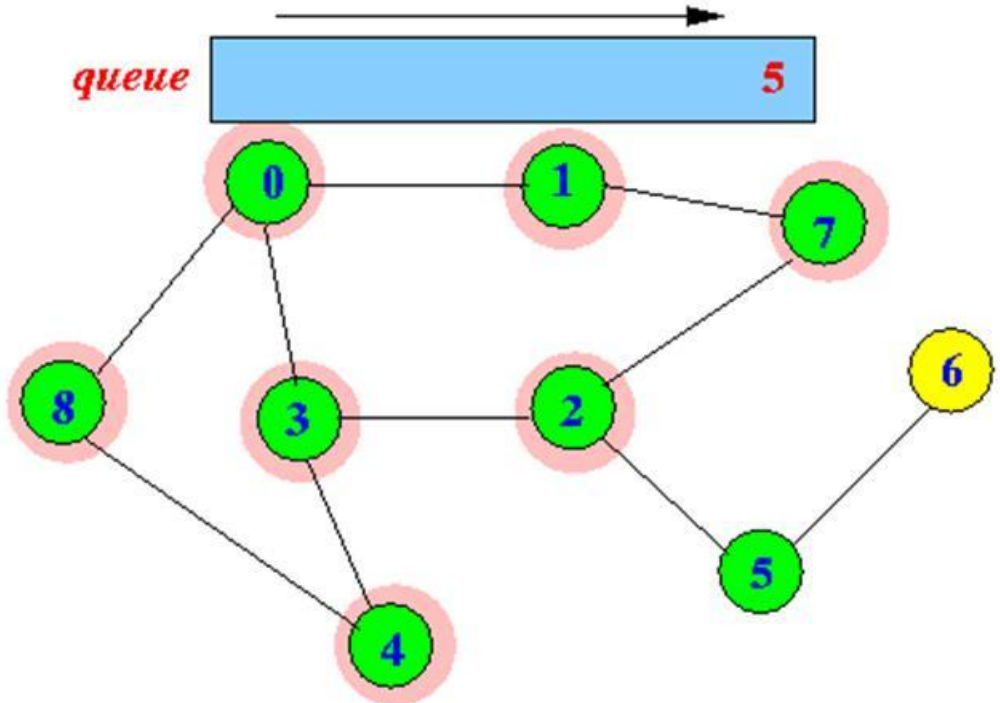


0 -> 1 -> 3 -> 8 -> 7 -> 2

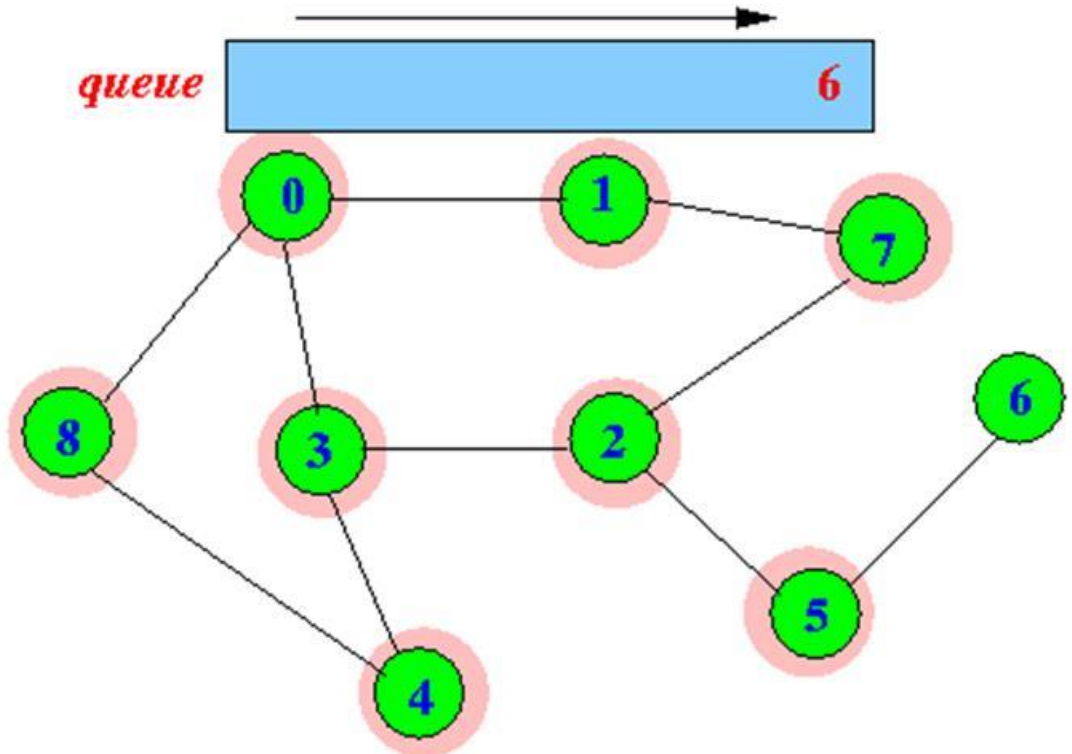


0 -> 1 -> 3 -> 8 -> 7 -> 2 -> 4 ->

Breadth First Search [BFS]



0 -> 1 -> 3 -> 8 -> 7 -> 2 -> 4 -> 5 ->



0 -> 1 -> 3 -> 8 -> 7 -> 2 -> 4 -> 5 -> 6

Breadth First Search [BFS]

- Time complexity of BFS is O($V + E$), if we use adjacency lists for representing the graphs, and O(V^2) for adjacency matrix representation.
- **Applications of BFS**
  - Finding all connected components in a graph
  - Finding all nodes within one connected component
  - Finding the shortest path between two nodes
  - Testing a graph for bipartiteness