1. Write a note on Asymptotic notations.

Asymptotic Notations

Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.

For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

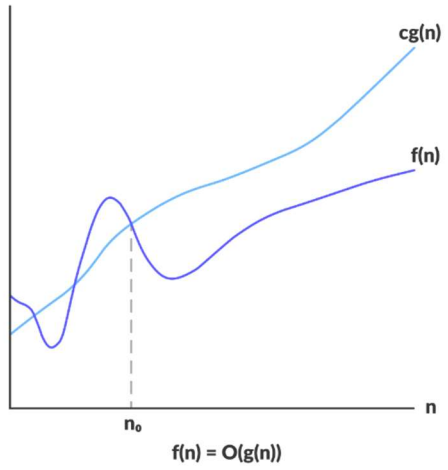
When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations:

- Big-O notation
- Omega notation
- Theta notation

Big-O Notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.



Big-O gives the upper bound of a function

O(g(n)) = { f(n): there exist positive constants c and n_0

such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$ }

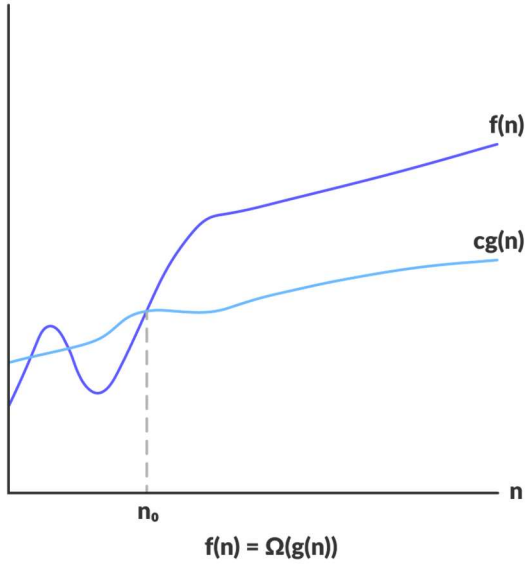
The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and cg(n), for sufficiently large n.

For any value of n, the running time of an algorithm does not cross the time provided by O(g(n)).

Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.

Omega Notation (Ω-notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.



Omega gives the lower bound of a function

$\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n_0

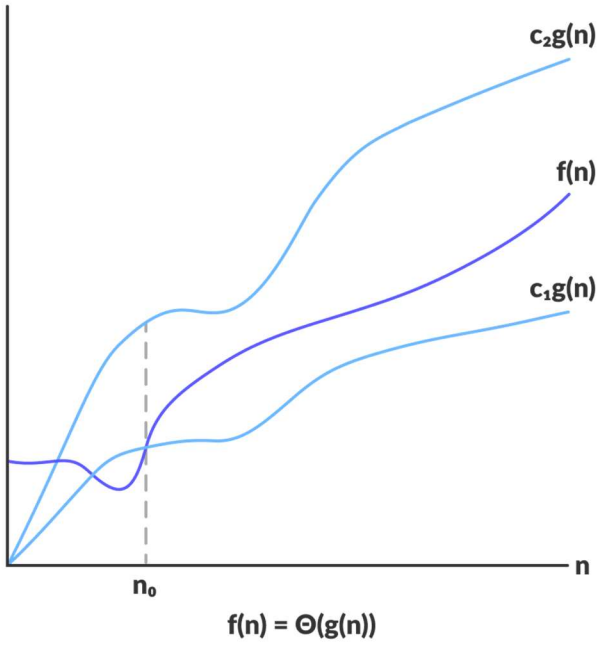
such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0 \}$

The above expression can be described as a function f(n) belongs to the set Ω(g(n)) if there exists a positive constant c such that it lies above cg(n), for sufficiently large n.

For any value of n, the minimum time required by the algorithm is given by Omega Ω(g(n)).

Theta Notation (Θ-notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.



Theta bounds the function within constants factors

For a function g(n), Θ(g(n)) is given by the relation:

Θ(g(n)) = { f(n): there exist positive constants c_1, c_2 and n_0

such that $0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n \geq n_0$ }

The above expression can be described as a function f(n) belongs to the set Θ(g(n)) if there exist positive constants c_1 and c_2 such that it can be sandwiched between $c_1g(n)$ and $c_2g(n)$, for sufficiently large n.

If a function f(n) lies anywhere in between $c_1g(n)$ and $c_2g(n)$ for all $n \geq n0$, then f(n) is said to be asymptotically tight bound.

2. Write limitations of any 2 data structures.

Stack

- Stack memory is of limited size.
- The total of size of the stack must be defined before.
- If too many objects are created then it can lead to stack overflow.
- Random accessing is not possible in stack.
- If the stack falls outside the memory it can lead to abnormal termination.

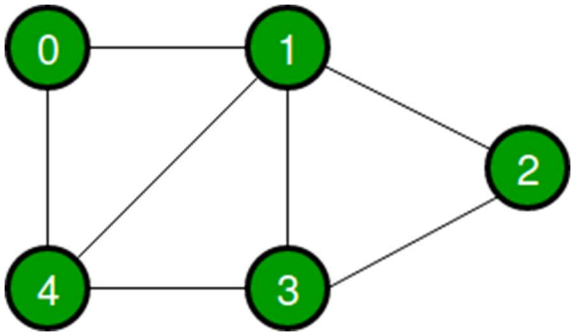
Queue

- The operations such as insertion and deletion of elements from the middle are time consuming.
- Limited Space.
- In a classical queue, a new element can only be inserted when the existing elements are deleted from the queue.
- Searching an element takes O(N) time.
- Maximum size of a queue must be defined prior.

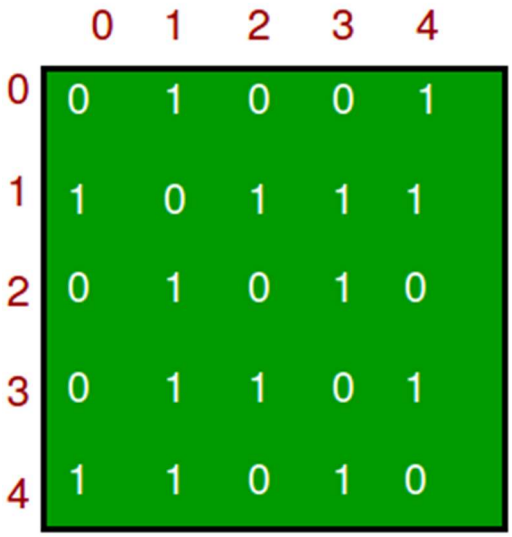
3. How to represent graphs using matrix.

An adjacency matrix is a way of representing a graph as a matrix of booleans (0's and 1's). A finite graph can be represented in the form of a square matrix on a computer, where the boolean value of the matrix indicates if there is a direct path between two vertices.

Adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.



The following two are the most commonly used representations of a graph.
1. Adjacency Matrix
2. Adjacency List
There are other representations also like, Incidence Matrix and Incidence List. The choice of graph representation is situation-specific. It totally depends on the type of operations to be performed and ease of use.



4. How Dijkstra algorithm is different from Bellman ford.

Dijkstra's Algorithm

- Single source shortest path algorithm.
- Doesn't work when there are negative edges.
- Result contains the vertices containing whole information about the network.
- Cannot be implemented easily in a distributed way.
- Time complexity is O(ElogV).
- Greedy approach is taken to implement the algorithm.

Bellman-Ford Algorithm

- Single source shortest path algorithm.
- Works when there is a negative edge. Can also detect negative weight cycle.
- Result contains vertices which contains the information about the other vertices they are connected to.
- Can be easily implemented in a distributed way.
- Time complexity is O(VE).
- Dynamic programming approach is taken to implement the algorithm.

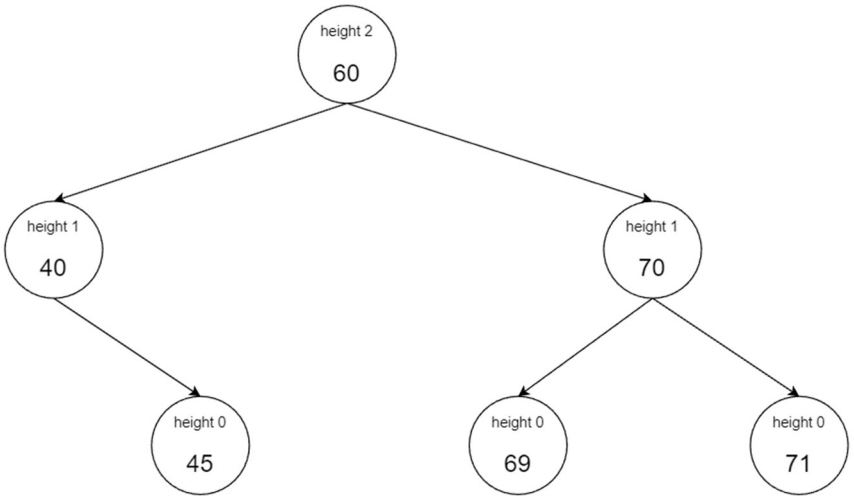
5. Illustrate applications of graphs.

- In Computer science graphs are used to represent the flow of computation.
- Google maps uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.
- In World Wide Web, web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example of Directed graph.
- In Operating System, we come across the Resource Allocation Graph where each process and resources are considered to be vertices.
- Microsoft Excel uses DAG means Directed Acyclic Graphs.
- In the Dijkstra algorithm, we use a graph. we find the smallest path between two or many nodes.
- On social media sites, we use graphs to track the data of the users. liked showing preferred post suggestions, recommendations, etc.

6. How binary search trees are different from balanced binary search trees? Why balanced BSTs are better than BSTs?

A binary search tree (BST) is a tree in which every node has exactly two children (left and right), every node's left child has a smaller value than the node, and every node's right child has a larger value than the node.

Let's look at the following BST.



You can see that at each level of the tree, a node either has no child, one child, or two children. When the height of the children nodes differs by more than 1, the BST is considered unbalanced.

Where there is a null child (i.e., no child is present), the height is taken to be -1.

A balanced BST, then, is a tree where the heights of the children nodes differ by at max 1, as shown in the example below.

Self-balancing binary search tree

A self-balancing binary search tree is a type of BST which tries to balance itself, in case of arbitrary insertions and deletions, by using rotations.

For example, consider the following BST.

Now, the last value entered in the tree is 80. If this were a self-balancing BST, it would do some rotation to make the tree balanced. That rotation is shown below.

We can see that the self-balancing BST balances itself by making the parent node a grandparent and moving the grandparent as a left child. Now, the tree is balanced.

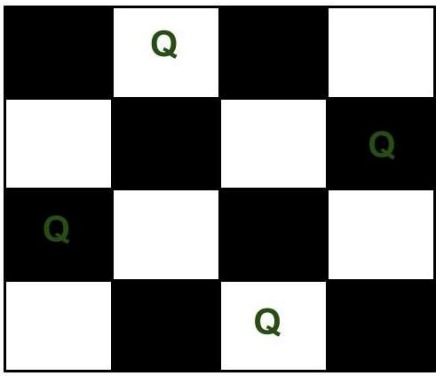
7. Discuss about n-queen problem.

This problem is to find an arrangement of N queens on a chess board, such that no queen can attack any other queens on the board.

The chess queens can attack in any direction as horizontal, vertical, horizontal and diagonal way.

A binary matrix is used to display the positions of N Queens, where no queens can attack other queens.

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, the following is a solution for the 4 Queen problem.



The expected output is a binary matrix that has 1s for the blocks where queens are placed. For example, the following is the output matrix for the above 4 queen solution.

{ 0, 1, 0, 0}
{ 0, 0, 0, 1}
{ 1, 0, 0, 0}
{ 0, 0, 1, 0}

N Queen problem is the classical Example of backtracking. N-Queen problem is defined as, "given N x N chess board, arrange N queens in such a way that no two queens attack each other by being in same row, column or diagonal".

- For N = 1, this is trivial case. For N = 2 and N = 3, solution is not possible. So we start with N = 4 and we will generalize it for N queens.

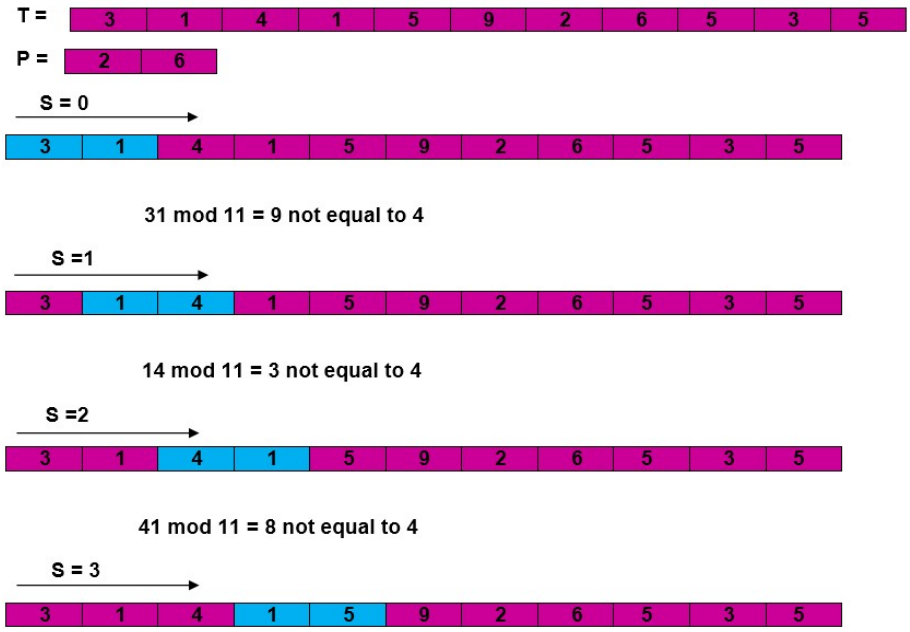
8. Explain working of Rabin Karp algorithm with example of suitable example.

The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for next M-character sequence. If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match.

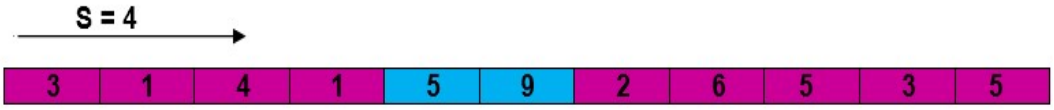
Example: For string matching, working module q = 11, how many spurious hits does the Rabin-Karp matcher encounters in Text T = 31415926535.....

1. T = 31415926535.....
2. P = 26
3. Here T.Length =11 so Q = 11
4. And P mod Q = 26 mod 11 = 4
5. Now find the exact match of P mod Q...

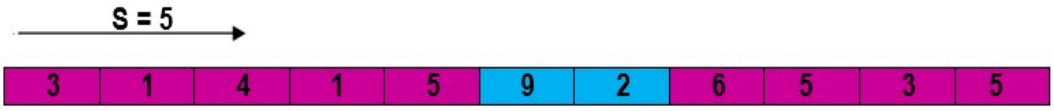
Solution:



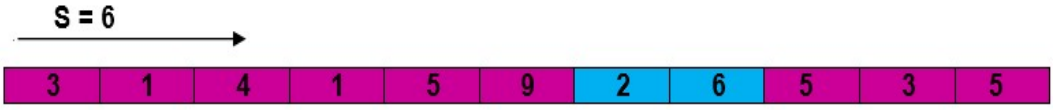
15 mod 11 = 4 equal to 4 **SPURIOUS HIT**



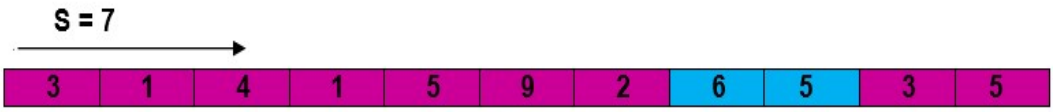
59 mod 11 = 4 equal to 4 **SPURIOUS HIT**

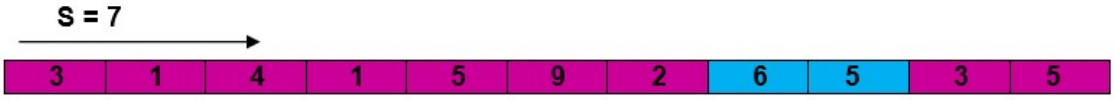


92 mod 11 = 4 equal to 4 **SPURIOUS HIT**



26 mod 11 = 4 **EXACT MATCH**





65 mod 11 = 10 not equal to 4

53 mod 11 = 9 not equal to 4

35 mod 11 = 2 not equal to 4

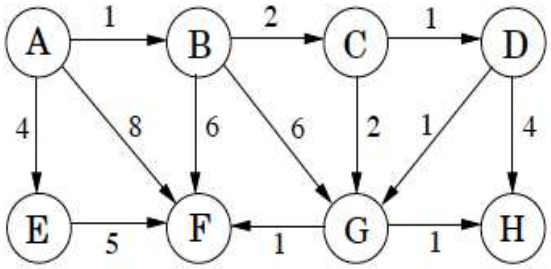
The Pattern occurs with shift 6.

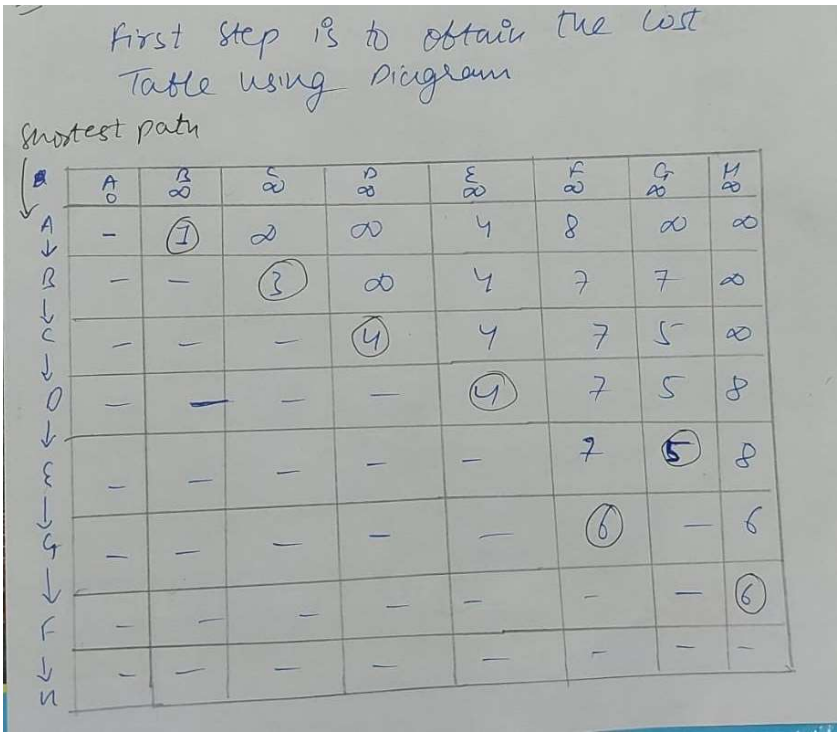
Complexity:

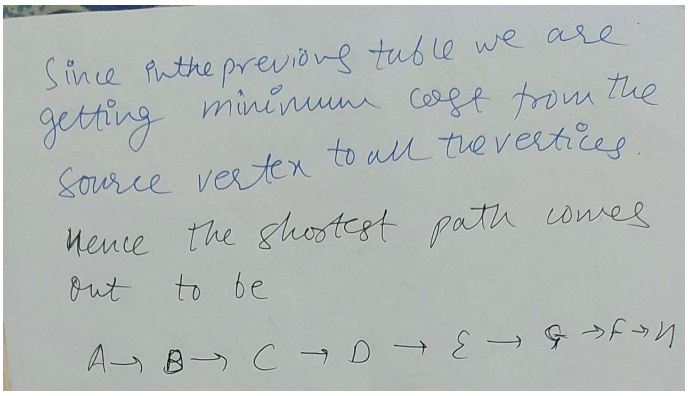
The running time of **RABIN-KARP-MATCHER** in the worst case scenario **O ((n-m+1) m** but it has a good average case running time. If the expected number of strong shifts is small **O (1)** and prime q is chosen to be quite large, then the Rabin-Karp algorithm can be expected to run in time **O (n+m)** plus the time to require to process spurious hits.

9. Find the shortest path from the given source S to all other vertices in the given graph using Dijkstra algorithm.

Start from source S=A.

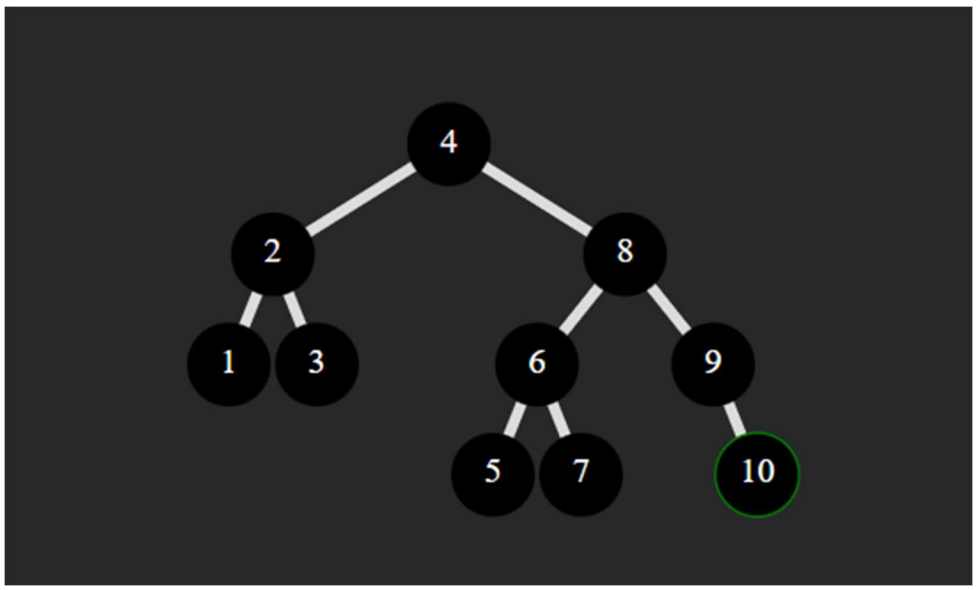






10.

Construct BST.
1 2 3 4 5 6 7 8 9 10



Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

- It is called a binary tree because each tree node has a maximum of two children.
- It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

- The left subtree contains only the keys which are lesser than the key of the node.
- The right subtree contains only the keys which are greater than the key of the node.

- The left and right subtree both should be binary search tree.

Real-time Application of Binary Search tree:

- BSTs are used for indexing in databases.
- It is used to implement searching algorithms.
- BSTs are used to implement Huffman coding algorithm.
- It is also used to implement dictionaries.

Advantages of Binary Search Tree:

- BST is fast in insertion and deletion when balanced.
- BST is efficient.
- We can also do range queries – find keys between N and M (N <= M).
- BST code is simple as compared to other data structures.

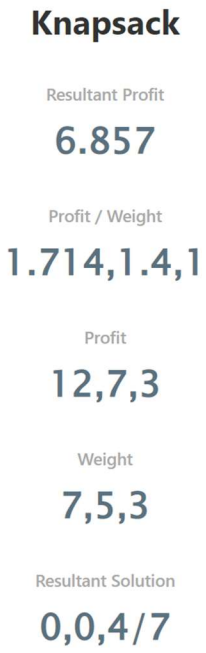
Disadvantages of Binary Search Tree:

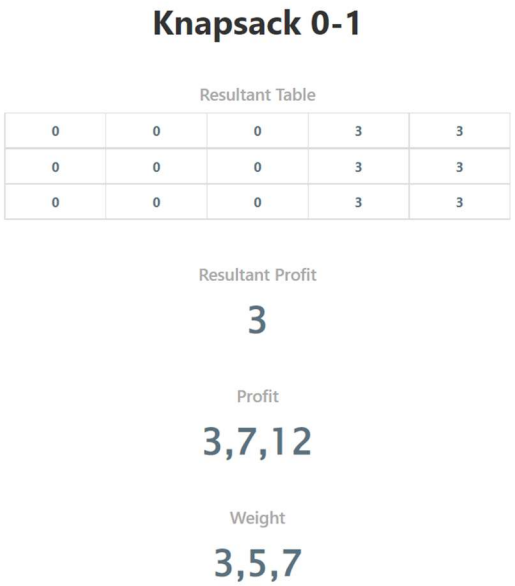
- The main disadvantage is that we should always implement a balanced binary search tree. Otherwise the cost of operations may not be logarithmic and degenerate into a linear search on an array.
- Accessing the element in BST is slightly slower than array.
- A BST can be imbalanced or degenerated which can increase the complexity.

11.

Solve the following instance of 0/1 Knapsack problem using Dynamic programming n = 3; (W1, W2, W3) = (3, 5, 7);

(P1, P2, P3) = (3, 7, 12); M = 4.





(Similar Example with Steps)

https://www.ques10.com/p/66448/define-knapsack-problem-solve-the-following-01-k-1/

https://www.gatevidyalay.com/0-1-knapsack-problem-using-dynamic-programming-approach/

Find the optimal solution for the 0/1 knapsack problem making use of dynamic programming approach. Consider-

$$n = 4$$

$$w = 5 \text{ kg}$$

$$(w1, w2, w3, w4) = (2, 3, 4, 5)$$

$$(b1, b2, b3, b4) = (3, 4, 5, 6)$$

Item	Weight (kg)	Value ($)
Mirror	2	3
Silver nugget	3	4
Painting	4	5
Vase	5	6

Solution-

Given-

- Knapsack capacity (w) = 5 kg
- Number of items (n) = 4

Step-01:

- Draw a table say 'T' with (n+1) = 4 + 1 = 5 number of rows and (w+1) = 5 + 1 = 6 number of columns.
- Fill all the boxes of 0^{th} row and 0^{th} column with 0.

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

T-Table

Step-02:

Start filling the table row wise top to bottom from left to right using the formula-

$$\mathbf{T(i, j) = max \{ T(i-1, j), value_i + T(i-1, j - weight_i) \}}$$

Finding T(1,1)-

We have,

- i = 1
- j = 1
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

T(1,1) = max { T(1-1 , 1) , 3 + T(1-1 , 1-2) }

$T(1,1) = \max \{ T(0,1) , 3 + T(0,-1) \}$

$T(1,1) = T(0,1)$ { Ignore $T(0,-1)$ }

$T(1,1) = 0$

Finding T(1,2)-

We have,

- $i = 1$
- $j = 2$
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

$T(1,2) = \max \{ T(1-1 , 2) , 3 + T(1-1 , 2-2) \}$

$T(1,2) = \max \{ T(0,2) , 3 + T(0,0) \}$

$T(1,2) = \max \{0 , 3+0\}$

$T(1,2) = 3$

Finding T(1,3)-

We have,

- $i = 1$
- $j = 3$
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

$T(1,3) = \max \{ T(1-1 , 3) , 3 + T(1-1 , 3-2) \}$

$T(1,3) = \max \{ T(0,3) , 3 + T(0,1) \}$

$T(1,3) = \max \{0 , 3+0\}$

T(1,3) = 3

Finding T(1,4)-

We have,

- i = 1
- j = 4
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

T(1,4) = max { T(1-1 , 4) , 3 + T(1-1 , 4-2) }

T(1,4) = max { T(0,4) , 3 + T(0,2) }

T(1,4) = max {0 , 3+0}

T(1,4) = 3

Finding T(1,5)-

We have,

- i = 1
- j = 5
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

T(1,5) = max { T(1-1 , 5) , 3 + T(1-1 , 5-2) }

T(1,5) = max { T(0,5) , 3 + T(0,3) }

T(1,5) = max {0 , 3+0}

T(1,5) = 3

Finding T(2,1)-

We have,

- i = 2
- j = 1
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,1) = max { T(2-1 , 1) , 4 + T(2-1 , 1-3) }

T(2,1) = max { T(1,1) , 4 + T(1,-2) }

T(2,1) = T(1,1) { Ignore T(1,-2) }

T(2,1) = 0

Finding T(2,2)-

We have,

- i = 2
- j = 2
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,2) = max { T(2-1 , 2) , 4 + T(2-1 , 2-3) }

T(2,2) = max { T(1,2) , 4 + T(1,-1) }

T(2,2) = T(1,2) { Ignore T(1,-1) }

T(2,2) = 3

Finding T(2,3)-

We have,

- i = 2
- j = 3
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

$T(2,3) = \max \{ T(2-1 , 3) , 4 + T(2-1 , 3-3) \}$

$T(2,3) = \max \{ T(1,3) , 4 + T(1,0) \}$

$T(2,3) = \max \{ 3 , 4+0 \}$

$T(2,3) = 4$

Finding T(2,4)-

We have,

- i = 2
- j = 4
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

$T(2,4) = \max \{ T(2-1 , 4) , 4 + T(2-1 , 4-3) \}$

$T(2,4) = \max \{ T(1,4) , 4 + T(1,1) \}$

$T(2,4) = \max \{ 3 , 4+0 \}$

$T(2,4) = 4$

Finding T(2,5)-

We have,

- i = 2

- j = 5
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,5) = max { T(2-1 , 5) , 4 + T(2-1 , 5-3) }

T(2,5) = max { T(1,5) , 4 + T(1,2) }

T(2,5) = max { 3 , 4+3 }

T(2,5) = 7

Similarly, compute all the entries.

After all the entries are computed and filled in the table, we get the following table-

	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

T-Table

- The last entry represents the maximum possible value that can be put into the knapsack.
- So, maximum possible value that can be put into the knapsack = 7.

Identifying Items To Be Put Into Knapsack-

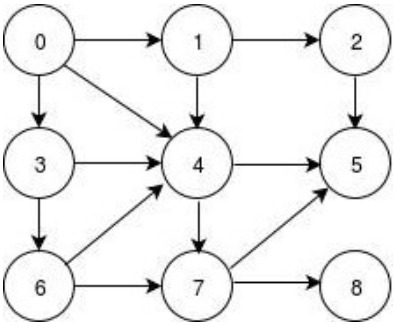
Following Step-04,

- We mark the rows labelled “1” and “2”.
- Thus, items that must be put into the knapsack to obtain the maximum value 7 are-

Item-1 and Item-2

12.

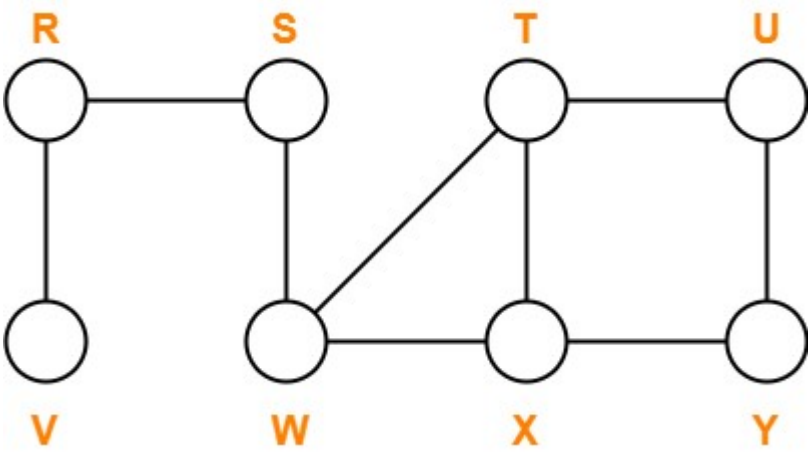
Traverse the following graph using Breadth First Traversal algorithm.



- BFS is useful for analyzing the nodes in a graph and constructing the shortest path of traversing through these.
- BFS can traverse through a graph in the smallest number of iterations.
- The architecture of the BFS algorithm is simple and robust.
- The result of the BFS algorithm holds a high level of accuracy in comparison to other algorithms.
- BFS iterations are seamless, and there is no possibility of this algorithm getting caught up in an infinite loop problem.

https://www.gatevidyalay.com/breadth-first-search-bfs-algorithm/

Similar Problem with Steps



Consider vertex S as the starting vertex.

Solution-

Step-01:

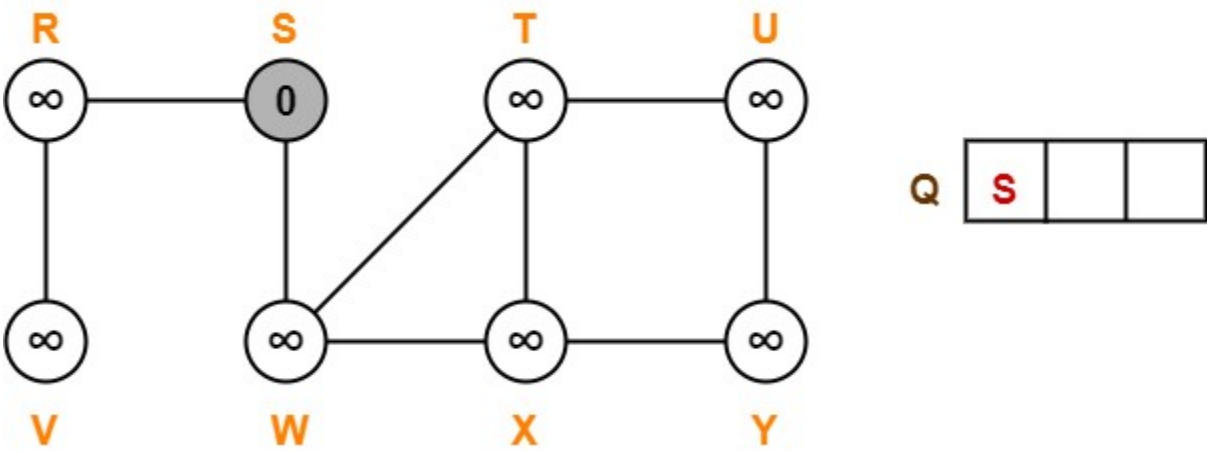
For all the vertices v except source vertex S of the graph, we initialize the variables as-

- color[v] = WHITE
- π[v] = NIL
- d[v] = ∞

For source vertex S, we initialize the variables as-

- color[S] = GREY
- π[S] = NIL
- d[S] = 0

We enqueue the source vertex S in the queue Q.

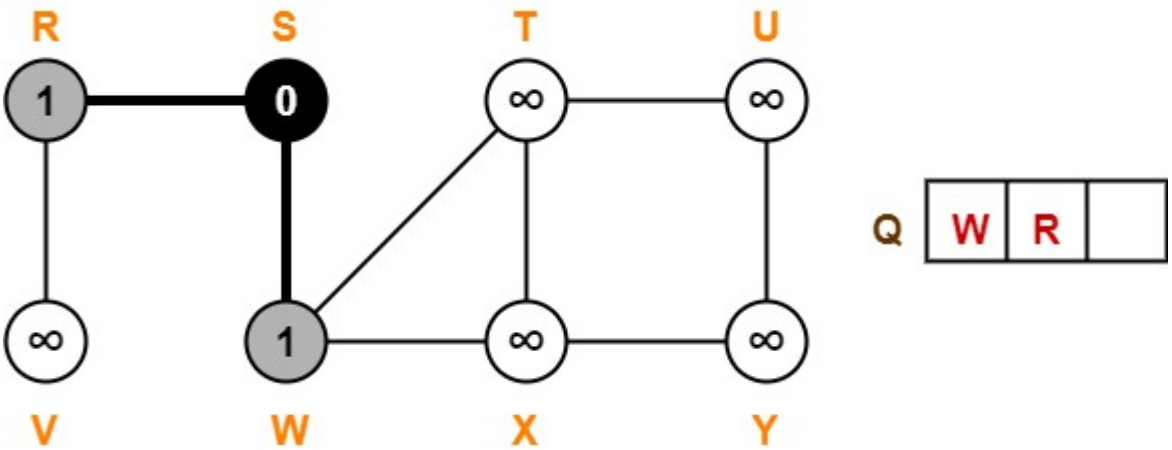


Step-02:

- Dequeue vertex S from the queue Q
- For all adjacent white vertices ‘v’ (vertices R and W) of vertex S, we do-

1. color[v] = GREY
2. d[v] = d[S] + 1 = 0 + 1 = 1
3. π[v] = S
4. Enqueue all adjacent white vertices of S in queue Q

- color[S] = BLACK

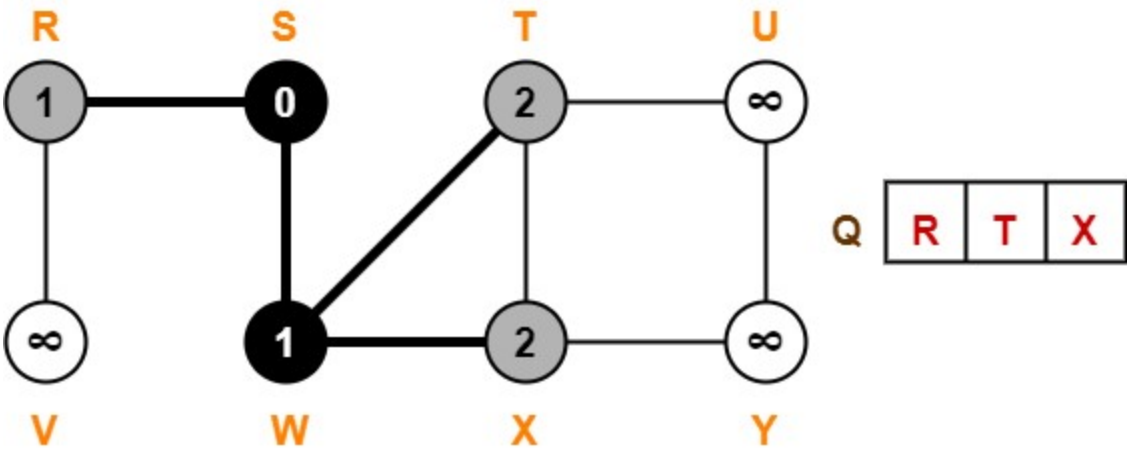


Step-03:

- Dequeue vertex W from the queue Q
- For all adjacent white vertices ‘v’ (vertices T and X) of vertex W, we do-

1. color[v] = GREY
2. d[v] = d[W] + 1 = 1 + 1 = 2
3. π[v] = W
4. Enqueue all adjacent white vertices of W in queue Q

- color[W] = BLACK

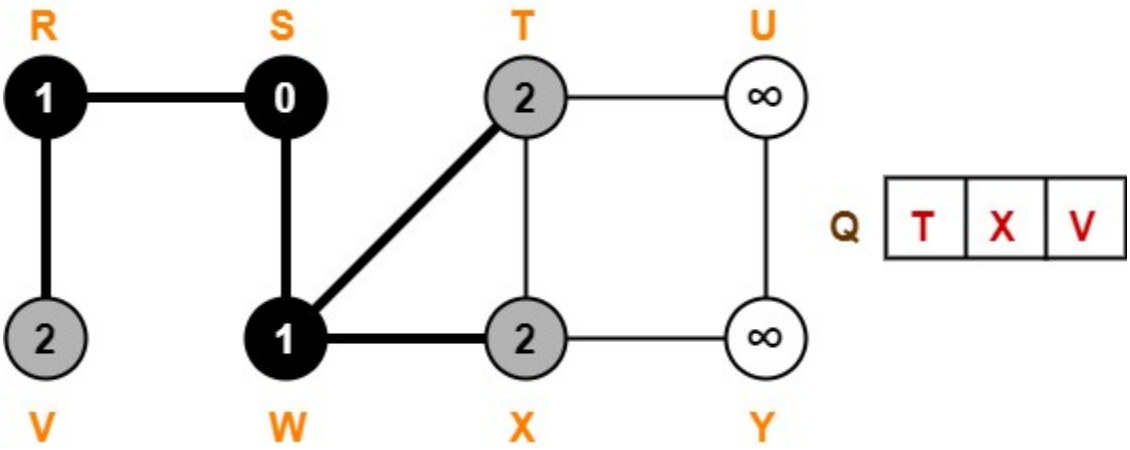


Step-04:

- Dequeue vertex R from the queue Q
- For all adjacent white vertices ‘v’ (vertex V) of vertex R, we do-

1. color[v] = GREY
2. d[v] = d[R] + 1 = 1 + 1 = 2
3. π[v] = R
4. Enqueue all adjacent white vertices of R in queue Q

- color[R] = BLACK

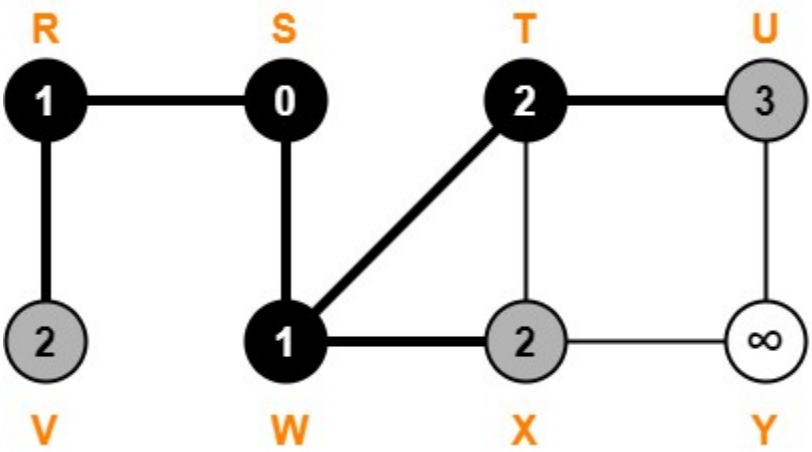


Step-05:

- Dequeue vertex T from the queue Q
- For all adjacent white vertices ‘v’ (vertex U) of vertex T, we do-

1. color[v] = GREY
2. d[v] = d[T] + 1 = 2 + 1 = 3
3. π[v] = T
4. Enqueue all adjacent white vertices of T in queue Q

- color[T] = BLACK

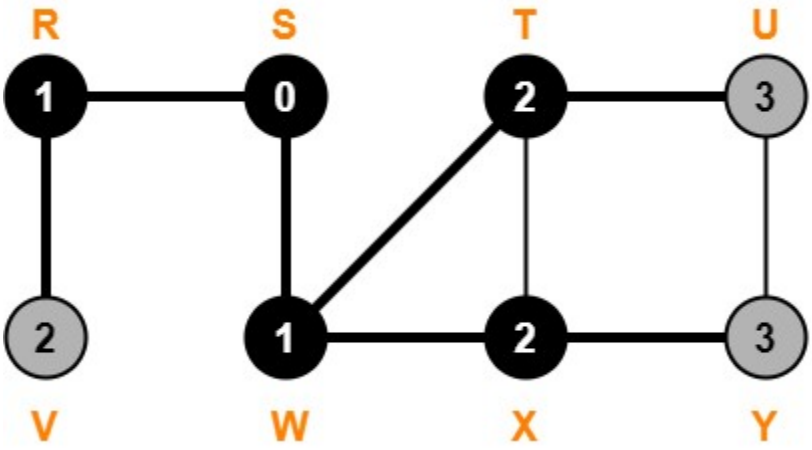


Step-06:

- Dequeue vertex X from the queue Q
- For all adjacent white vertices ‘v’ (vertex Y) of vertex X, we do-

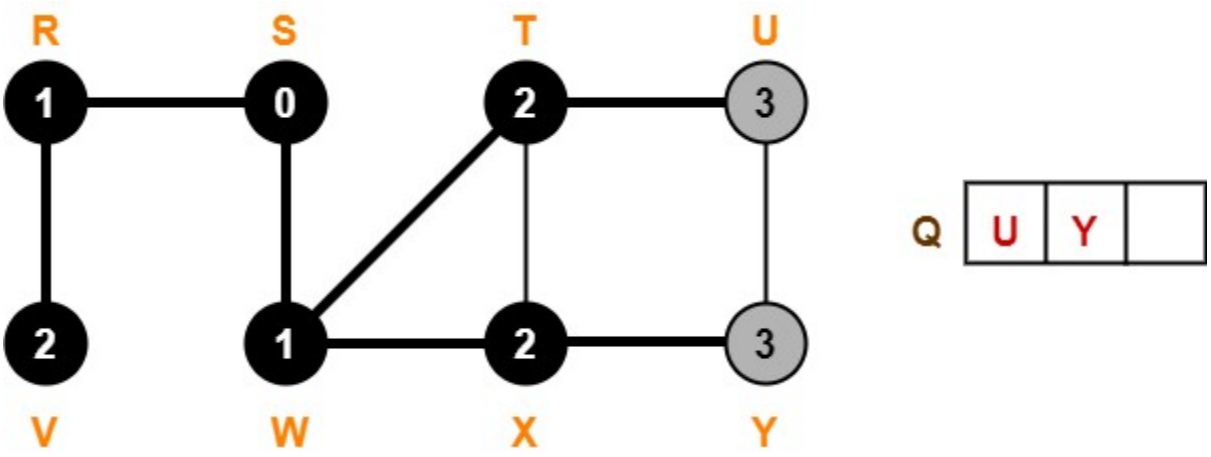
1. color[v] = GREY
2. d[v] = d[X] + 1 = 2 + 1 = 3
3. π[v] = X
4. Enqueue all adjacent white vertices of X in queue Q

- color[X] = BLACK



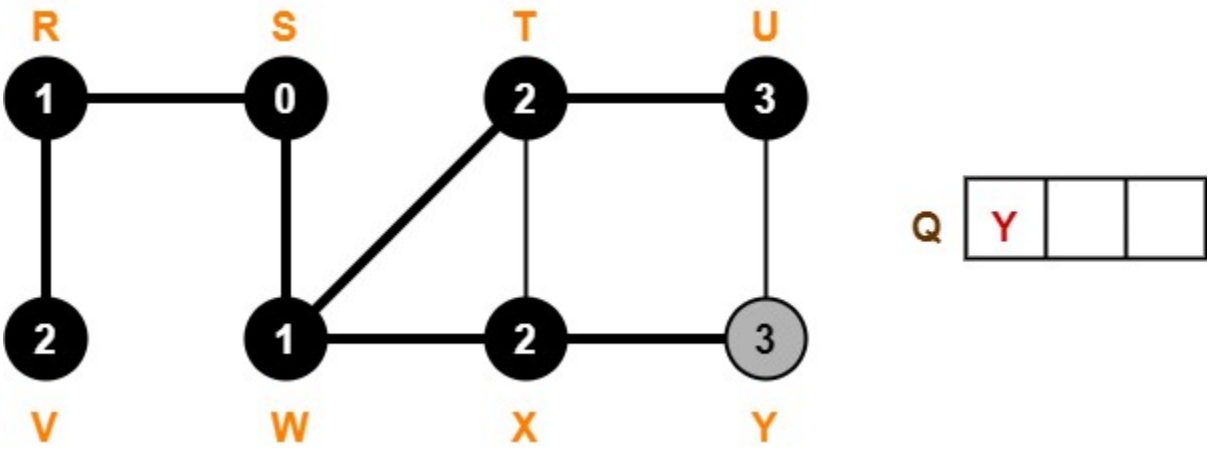
Step-07:

- Dequeue vertex V from the queue Q
- There are no adjacent white vertices to vertex V.
- color[V] = BLACK



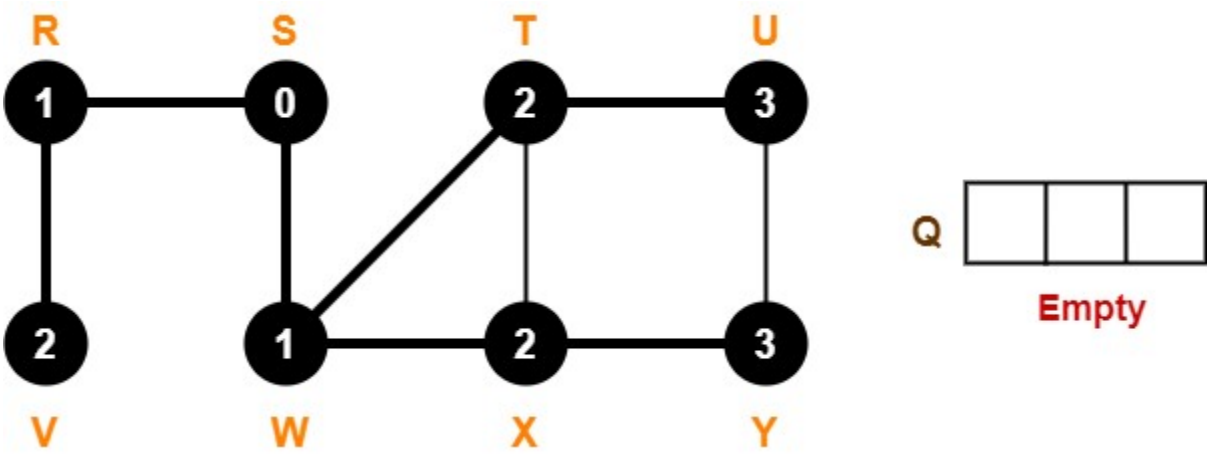
Step-08:

- Dequeue vertex U from the queue Q
- There are no adjacent white vertices to vertex U.
- color[U] = BLACK



Step-09:

- Dequeue vertex Y from the queue Q
- There are no adjacent white vertices to vertex Y.
- color[Y] = BLACK



Since, all the vertices have turned black and the queue has got empty, so we stop