



University Term Test-II

SECOND HALF-2019

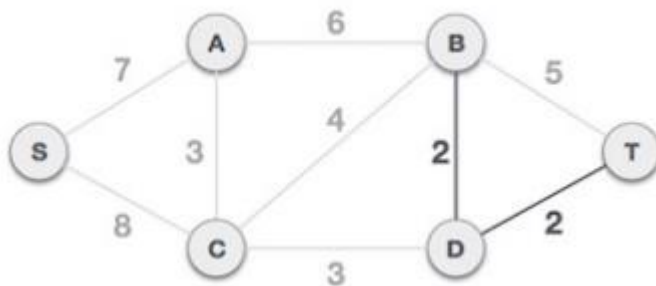
Term Test QP Code:-

Note the following instructions

1. Attempt any two questions.
2. Draw neat diagrams wherever necessary.
3. Write everything in ink (no pencil) only.
4. Assume data, if missing, with justification.

Q.1 a) What is minimum spanning tree? Draw minimum spanning tree using Prime's and Kruskal's algorithm for the following graph.

[10
M]



ANS:

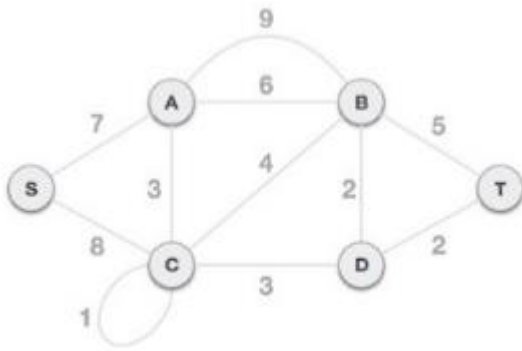
Kruskal's Algorithm

Kruskal's algorithm to find the minimum cost spanning tree uses the greedy approach. This algorithm treats the graph as a forest and every node it has as an individual tree. A tree connects to another only and only if, it has the least cost among all available options and does not violate MST properties.

Algorithm Steps:

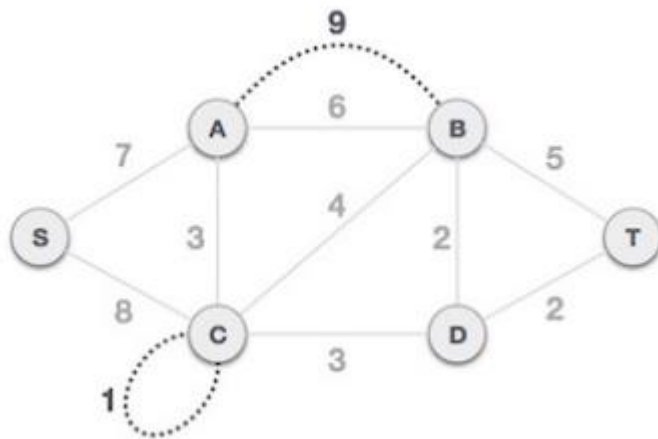
- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle, edges which connect only disconnected components.

- To understand Kruskal's algorithm let us consider the following example

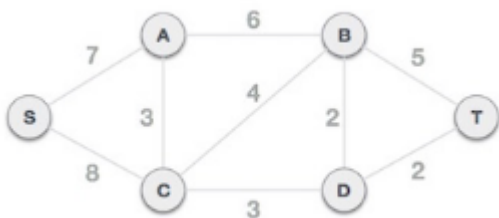


Step 1 - Remove all loops and Parallel Edges

Remove all loops and parallel edges from the given graph.



In case of parallel edges, keep the one which has the least cost associated and remove all others.

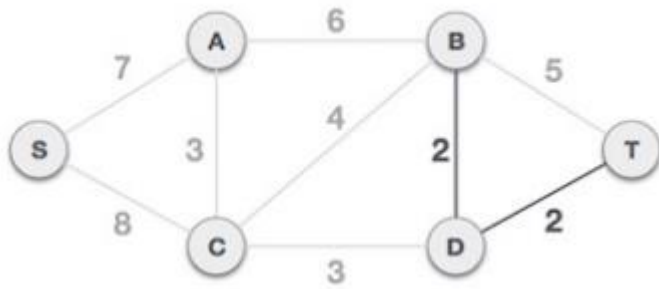


Step 2 - Arrange all edges in their increasing order of weight The next step is to create a set of edges and weight, and arrange them in an ascending order of weightage (cost).

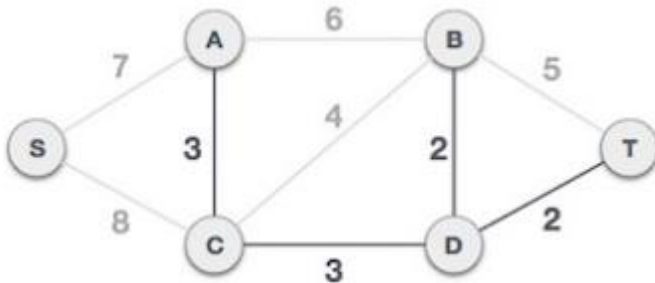
B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

Step 3 - Add the edge which has the least weight age Now we start adding edges to the graph beginning from the one which has the least weight. Throughout, we shall

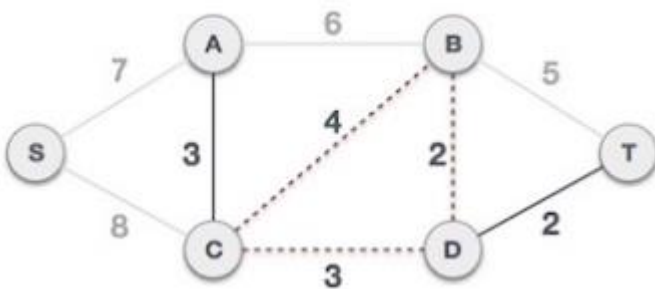
keep checking that the spanning properties remain intact. In case, by adding one edge, the spanning tree property does not hold then we shall consider not including the edge in the graph.



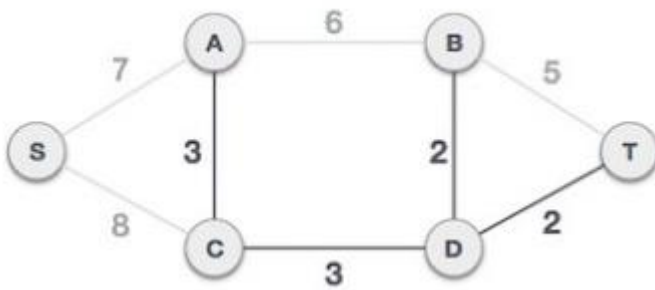
The least cost is 2 and edges involved are B,D and D,T. We add them. Adding them does not violate spanning tree properties, so we continue to our next edge selection. Next cost is 3, and associated edges are A,C and C,D. We add them again –



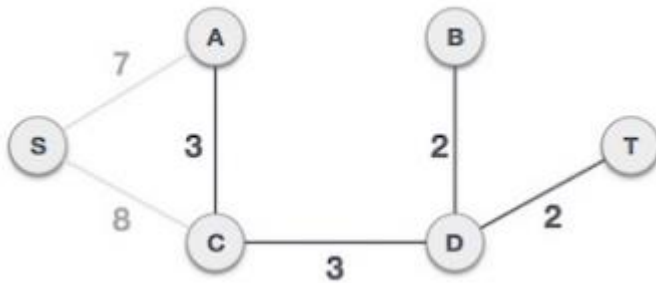
Next cost in the table is 4, and we observe that adding it will create a circuit in the graph.



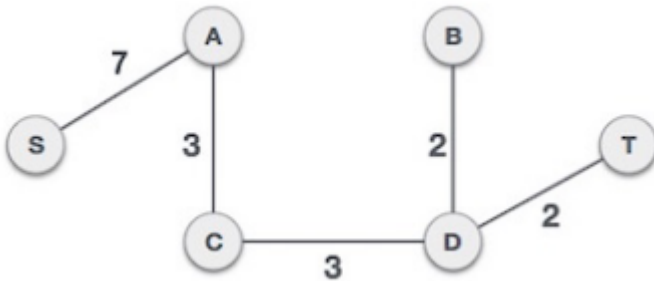
We ignore it. In the process we shall ignore/avoid all edges that create a circuit.



We observe that edges with cost 5 and 6 also create circuits. We ignore them and move on.



Now we are left with only one node to be added. Between the two least cost edges available 7 and 8, we shall add the edge with cost 7.



By adding edge S,A we have included all the nodes of the graph and we now have minimum cost spanning tree.

$$\text{Cost solution} = (S,A) + (C,A) + (D,C) + (D,B) + (D,T) = 7 + 3 + 3 + 2 + 2 = 17$$

Prim's Algorithm

Prim's Algorithm also use Greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an edge in Kruskal's, we add vertex to the growing spanning tree in Prim's.

Algorithm Steps:

- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is

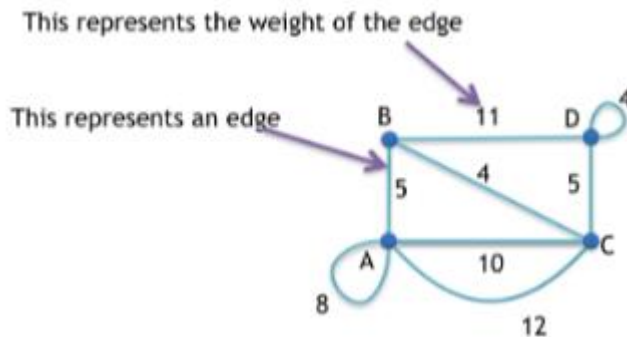
not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.

- Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

Consider the example below:

Graph

Consider the following graph.

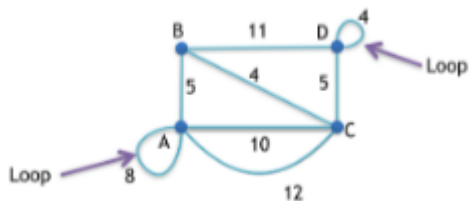


We will find MST for the above graph shown in the image.

Step 1: Remove all loops

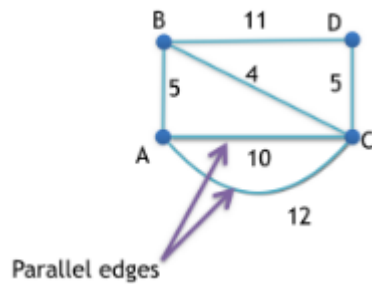
Any edge that starts and ends at the same vertex is a loop.

Loops are marked in the image given below.

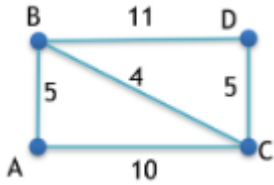


Step 2: Remove all parallel edges between two vertex except the one with least weight

In this graph, vertex A and C are connected by two parallel edges having weight 10 and 12 respectively. So, we will remove 12 and keep 10.

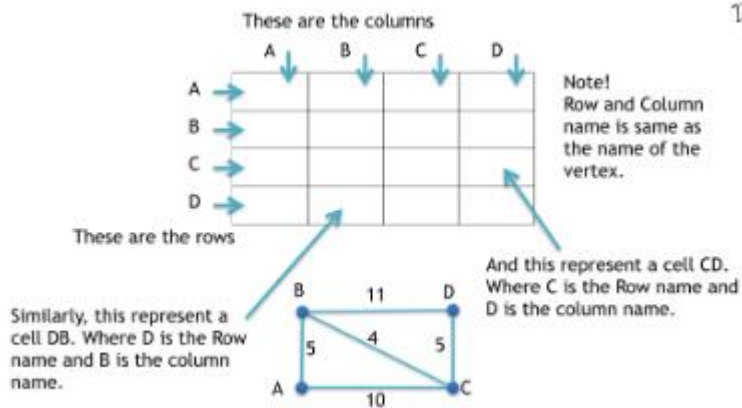


We are now ready to find the minimum spanning tree.



Step 3: Create table

As our graph has 4 vertices, so our table will have 4 rows and 4 columns.

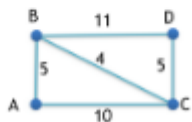


Now, put 0 in cells having same row and column name.

Find the edges that directly connect two vertices and fill the table with the weight of the edge. If no direct edge exists then fill the cell with infinity.

	A	B	C	D
A	0	5	10	-
B	5	0	4	11
C	10	4	0	5
D	-	11	5	0

Our table is completely filled, so our next job is to find the MST



Finding MST

Start from vertex A, find the smallest value in the A-row.

Note! We will not consider 0 as it will correspond to the same vertex.

5 is the smallest unmarked value in the A-row. So, we will mark the edge connecting vertex A and B and tick 5 in AB and BA cell.

	A	B	C	D
A	0	5✓	10	∞
B	5✓	0	4	11
C	10	4	0	5
D	∞	11	5	0

Smallest value in cell AB

As we connected vertex A and B in the previous step, so we will now find the smallest value in the A-row and B-row.

4 is the smallest unmarked value in the A-row and B-row. So, we will mark the edge connecting vertex B and C and tick 4 in BC and CB cell.

	A	B	C	D
A	0	5✓	10	∞
B	5✓	0	4✓	11
C	10	4✓	0	5
D	∞	11	5	0

Smallest value in cell BC

As vertex A-B and B-C were connected in the previous steps, so we will now find the smallest value in A-row, B-row and C-row.

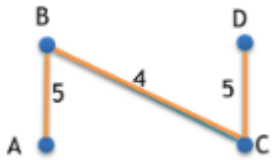
5 is the smallest unmarked value in the A-row, B-row and C-row. So, we will mark the edge connecting vertex C and D and tick 5 in CD and DC cell.

	A	B	C	D
A	0	5✓	10	∞
B	5✓	0	4✓	11
C	10	4✓	0	5✓
D	∞	11	5✓	0

Smallest value in cell CD

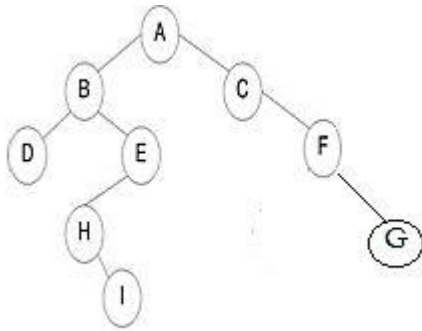
Result

Following is the required Minimum Spanning Tree for the given graph.



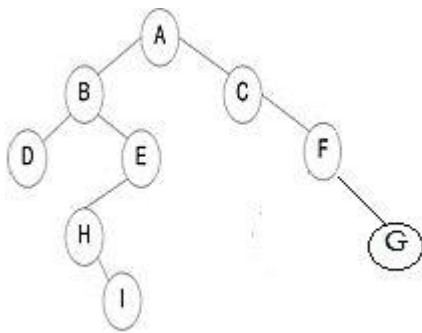
b) Define binary search tree. Find out inorder, preorder and postorder of the tree given below

[10 M]



ANS:

Traverse the binary tree into preorder, inorder and postorder by giving its algorithm.



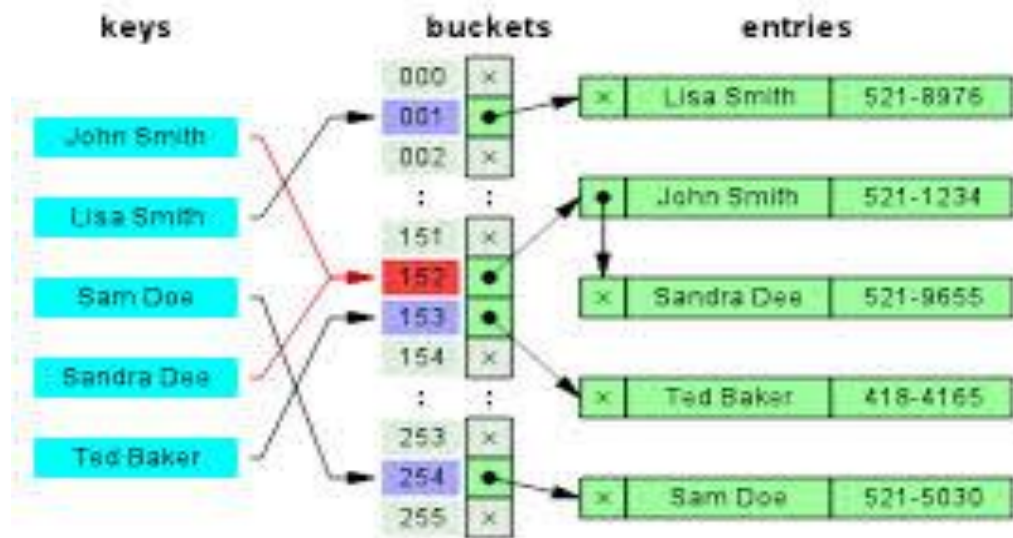
The preorder tree traversal's algorithm is:-

Visit the node.

Traverse the left subtree.

Traverse the right subtree.

	<p>The inorder tree traversal's algorithm is:-</p> <p>Traverse the left subtree</p> <p>Visit the node.</p> <p>Traverse the right subtree.</p> <p>The postorder tree traversal's algorithm is:-</p> <p>Traverse the left subtree.</p> <p>Traverse the right subtree.</p> <p>Visit the node.</p> <p>The preorder traversal is: A-B-D-E-H-I-C-F-G</p> <p>The inorder traversal is: D-B-H-I-E-A-C-G-F</p> <p>The postorder traversal is: D-I-H-E-B-G-F-C</p>	
Q.2	<p>a) What is collision? What are the methods to resolve the collision? Explain linear probing with an example.</p> <p>ANS:</p> <p>As a hash function gets us a small number for a key which is a big integer or string, there is a possibility that two keys result in the same value. The situation where a newly inserted key maps to an already occupied slot in the hash table is called collision and must be handled using some collision handling technique. Collisions are very likely even if we have big table to store keys. An important observation is Birthday Paradox. With only 23 persons, the probability that two people have the same birthday is 50%.</p> <p>Hash collision resolution techniques:</p> <p>Open Hashing (Separate chaining)</p> <p>Open Hashing, is a technique in which the data is not directly stored at the hash key index (k) of the Hash table. Rather the data at the key index (k) in the hash table is a pointer to the head of the data structure where the data is actually stored. In the most simple and common implementations the data structure adopted for storing the element is a linked-list.</p>	[10 M]



In this technique when a data needs to be searched, it might become necessary (worst case) to traverse all the nodes in the linked list to retrieve the data.

Note that the order in which the data is stored in each of these linked lists (or other data structures) is completely based on implementation requirements. Some of the popular criteria are insertion order, frequency of access etc.

closed hashing (open Addressing)

In this technique a hash table with pre-identified size is considered. All items are stored in the hash table itself. In addition to the data, each hash bucket also maintains the three states: EMPTY, OCCUPIED, DELETED. While inserting, if a collision occurs, alternative cells are tried until an empty bucket is found. For which one of the following technique is adopted.

Linear Probing (this is prone to clustering of data + Some other constrains.

Quadratic probing

Double hashing (in short in case of collision another hashing function is used with the key value as an input to identify where in the open addressing scheme the data should actually be stored.)

Linear Probing: In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also. let **hash(x)** be the slot index computed using hash function and **S** be the table size

If slot $\text{hash}(x) \% S$ is full, then we try $(\text{hash}(x) + 1) \% S$

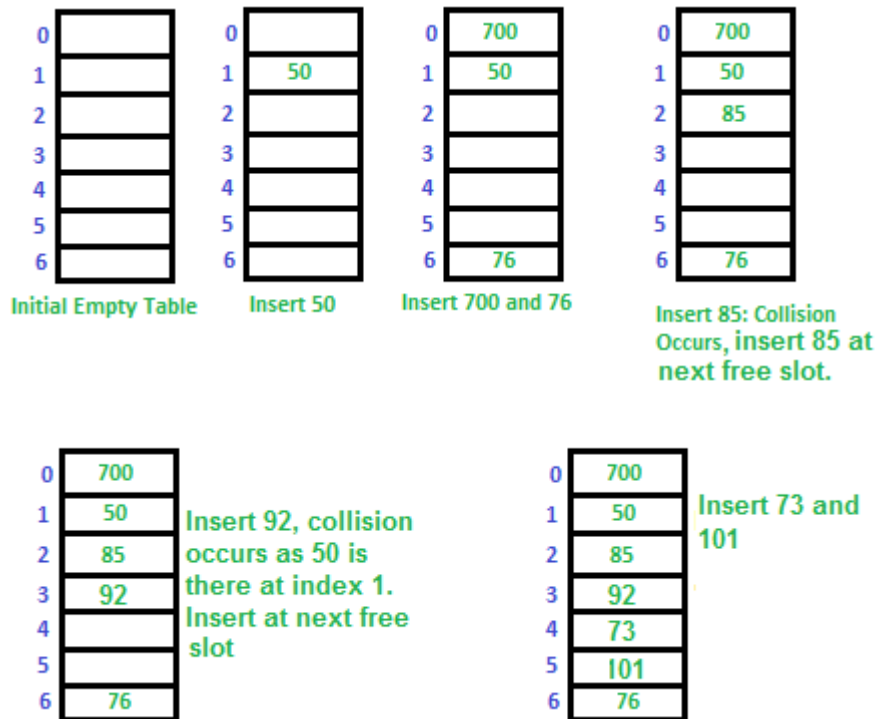
If $(\text{hash}(x) + 1) \% S$ is also full, then we try $(\text{hash}(x) + 2) \% S$

If $(\text{hash}(x) + 2) \% S$ is also full, then we try $(\text{hash}(x) + 3) \% S$

.....

.....

Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

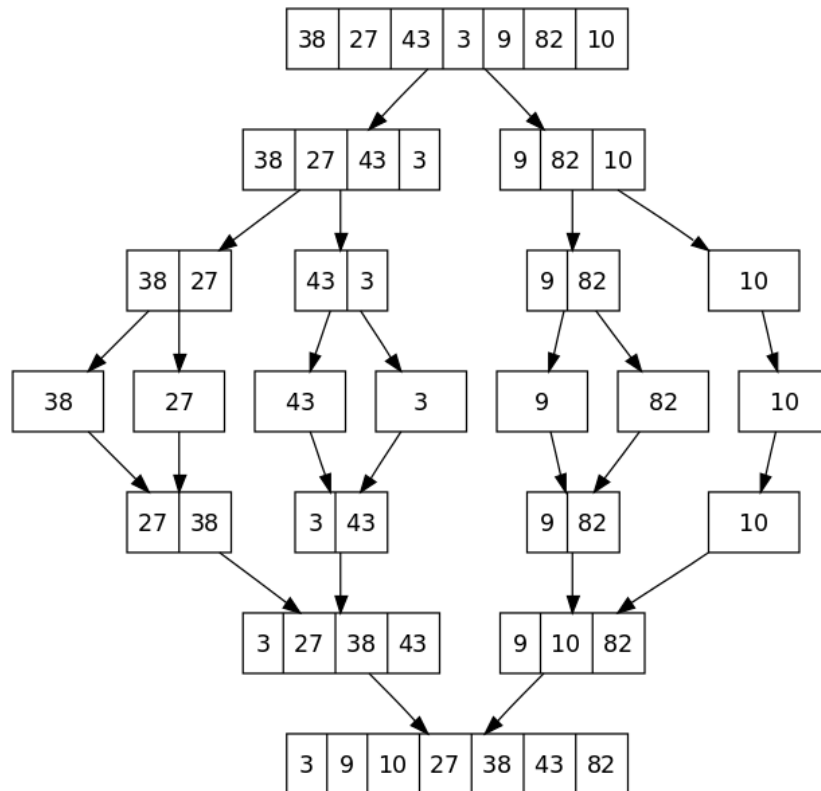


Clustering: The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

b) Write an algorithm for Merge sort and comment on its time complexity.
ANS:

[10
M]

- Merge Sort is a type of **recursive algorithm**.
- Merge Sort is a **stable sort**, which means the “equal” elements appear in the same order in the sorted array as they were in the unsorted array.
- Merge Sort is a best-sorting technique for sorting **Linked Lists**. Because in linked list it can be implemented without extra space as elements can be inserted in the middle in $O(1)$ extra space and $O(1)$ time.



In the above picture we see that the size of the array is ($n=7$). As we know $\lceil 7/2 \rceil = 3$, so while the first division of array, the size of one part will be 3 and another part will be ($7-3=4$).

Similarly, the array will recursively divide into two halves till we get all sub-array containing of exactly 1 element. Then each element is compared with the adjacent array to sort and merge the two adjacent arrays till the complete array is merged. Finally, all the elements are sorted and merged.

// algorithm for merge sort

MERGE (ARR, BEG, MID, END)

Step 1: [INITIALIZE] SET $I = \text{BEG}$, $J = \text{MID} + 1$, INDEX =

Step 2: Repeat while ($I \leq \text{MID}$) AND ($J \leq \text{END}$)

IF $\text{ARR}[I] < \text{ARR}[J]$

SET $\text{TEMP}[\text{INDEX}] = \text{ARR}[I]$

SET $I = I + 1$

ELSE

SET TEMP[INDEX] = ARR[J]

SET J = J + 1

[END OF IF]

SET INDEX = INDEX + 1

[END OF LOOP]

Step 3: [Copy the remaining elements of right sub-array, if any]

IF I > MID

Repeat while J <= END

SET TEMP[INDEX] = ARR[J]

SET INDEX = INDEX + 1, SET J = J + 1

[END OF LOOP]

[Copy the remaining elements of left sub-array, if any]

ELSE

Repeat while I <= MID

SET TEMP[INDEX] = ARR[I]

SET INDEX = INDEX + 1, SET I = I + 1

[END OF LOOP]

[END OF IF]

Step 4: [Copy the contents of TEMP back to ARR] SET K=

Step 5: Repeat while K < INDEX

SET ARR[K] = TEMP[K]

SET K = K + 1

[END OF LOOP]

Step 6: END

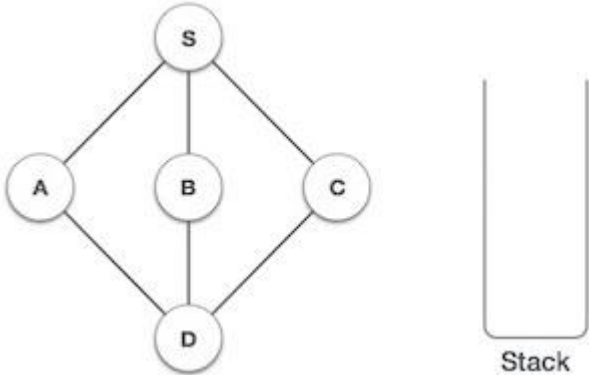
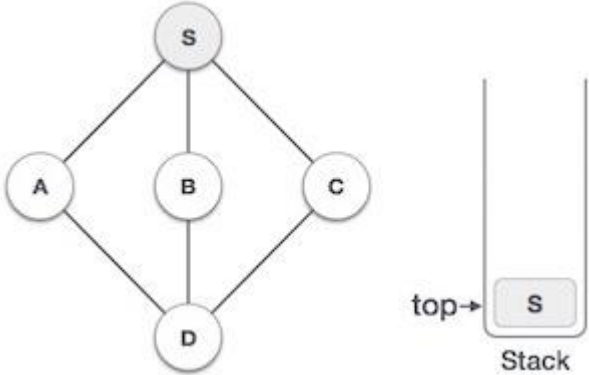
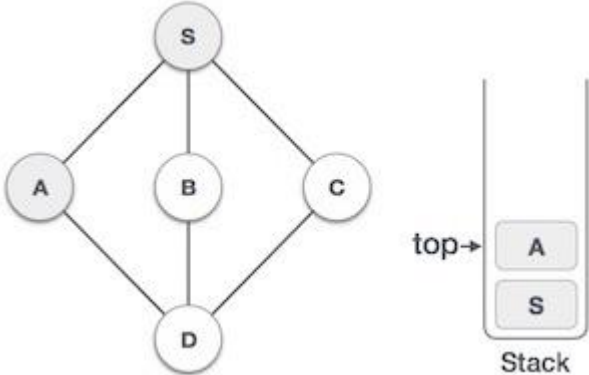
MERGE_SORT(ARR, BEG, END)

Step 1: IF BEG < END

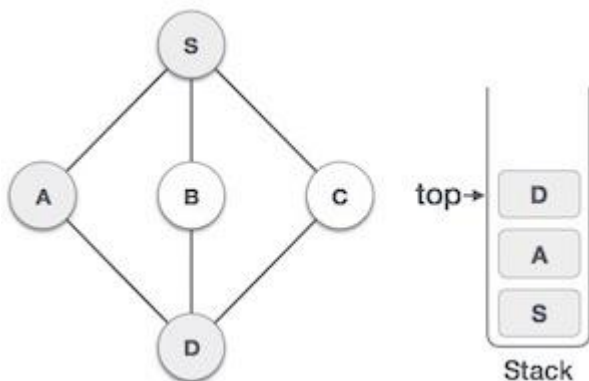
SET MID = (BEG + END)/2

CALL MERGE_SORT (ARR, BEG, MID)

	<div>CALL MERGE_SORT (ARR, MID + 1, END)</div> <div>MERGE (ARR, BEG, MID, END)</div> <div>[END OF IF]</div> <div>Step 2: END</div> <div>Complexity of merge sort : The running time of merge sort in the average case and the worst case can be given as $O(n \log n)$. Best time complexity is also same as worst and average time complexity.</div>							
Q.3	<div>a) Explain DFS and BFS algorithm with an example.</div> <div>ANS:</div> <div>Depth First Search:</div> <div>The general idea behind depth first traversal is that, starting from any random vertex, single path is traversed until a vertex is found whose all the neighbors are already been visited. The search then backtracks on the path until a vertex with unvisited adjacent vertices is found and then begin traversing a new path starting from that vertex, and so on. This process will continue until all the vertices of the graph are visited.</div> <div>//Algorithm for DFS()</div> <div>Step1. Initialize all the vertices to ready state (STATUS = 1)</div> <div>Step2. Put the starting vertex into STACK and change its status to waiting (STATUS = 2)</div> <div>Step 3: Repeat Step 4 and 5 until STACK is EMPTY</div> <div>Step 4: POP the top vertex from STACK, Process the vertex, Change its status to processed state (STATUS = 3)</div> <div>Step 5: PUSH all the neighbors in the ready state (STATUS = 1) to the STACK and change their status to waiting state (STATUS = 2)</div> <div>Step 6: Exit.</div> <div>Example:</div> <table><tr><td>S</td><td>Traversal</td><td>Description</td></tr><tr><td>te</td><td></td><td></td></tr></table>	S	Traversal	Description	te			[10 M]
S	Traversal	Description						
te								

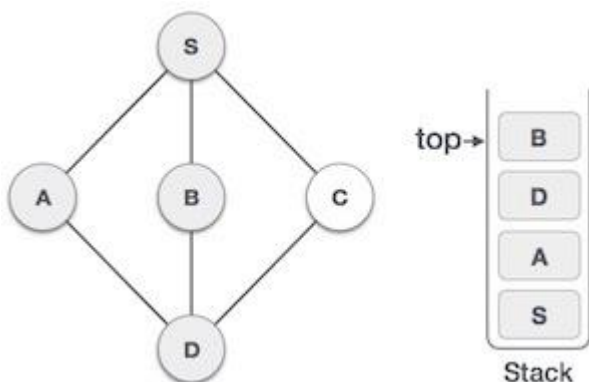
	p		
1			Initialize the stack.
2			Mark S as visited and put it onto the stack. Explore any unvisited adjacent node from S . We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3			Mark A as visited and put it onto the stack. Explore any unvisited adjacent node from A . Both S and D are adjacent to A but we are concerned for unvisited nodes only.

4



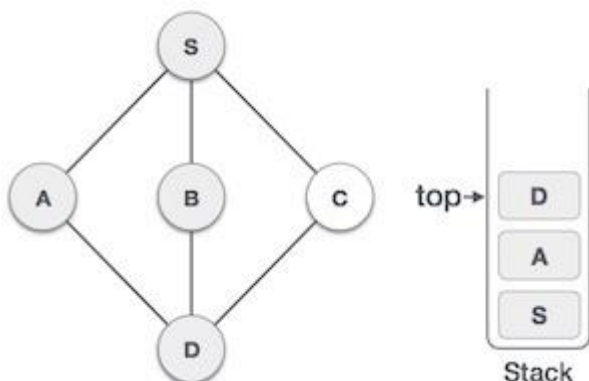
Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.

5



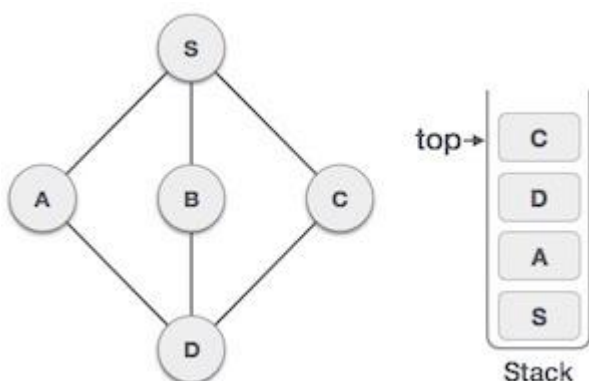
We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.

6



We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.

7



Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

Breadth First Search:

The general idea behind breadth first traversal is that, start at a random vertex, then visit all of its neighbors, the first vertex that we visit, say is at level '0' and the neighbors are at level '1'. After visiting all the vertices at level '1' we then pick one of these vertexes at level '1' and visit all its unvisited neighbors, we repeat this procedure for all other vertices at level '1'. Say neighbors of level 1 are in level 2, now we will visit the neighbors of all the vertices at level 2, and this procedure will continue.

// algorithm for BFS()

Step1. Initialize all the vertices to ready state (STATUS = 1)

Step2. Put the starting vertex into QUEUE and change its status to waiting (STATUS = 2)

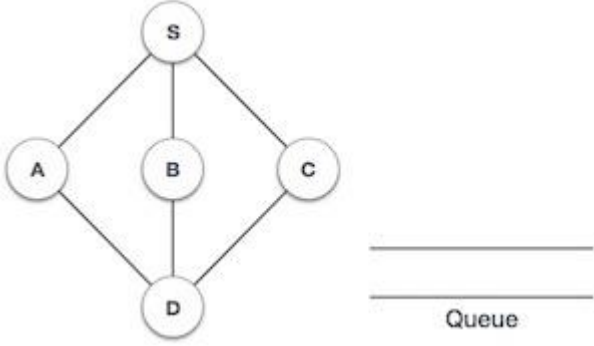
Step 3: Repeat Step 4 and 5 until QUEUE is EMPTY

Step 4: Remove the front vertex from QUEUE, Process the vertex, Change its status to processed state (STATUS = 3)

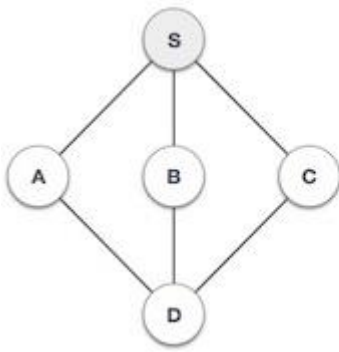
Step 5: ADD all the neighbors in the ready state (STATUS = 1) to the RARE of the QUEUE and change their status to waiting state (STATUS = 2)

Step 6: Exit.

Example:

Step	Traversal	Description
1		Initialize the queue.

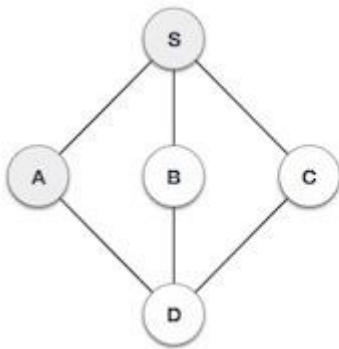
2



 Queue

We start from visiting **S** (starting node), and mark it as visited.

3

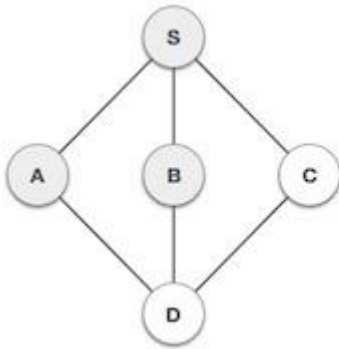


A _____

 Queue

We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.

4

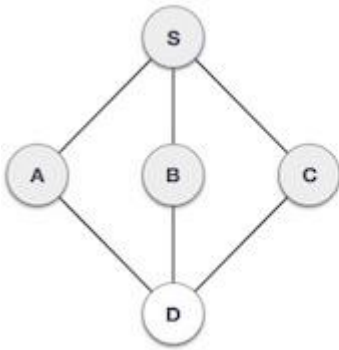


B **A** _____

 Queue

Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.

5



C **B** **A** _____

 Queue

Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.

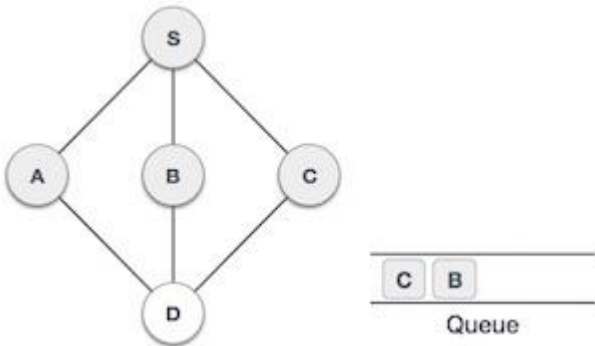
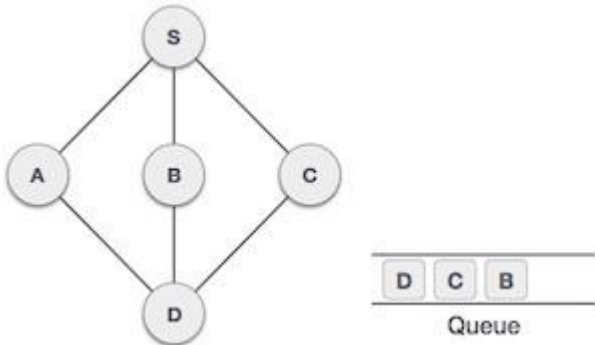

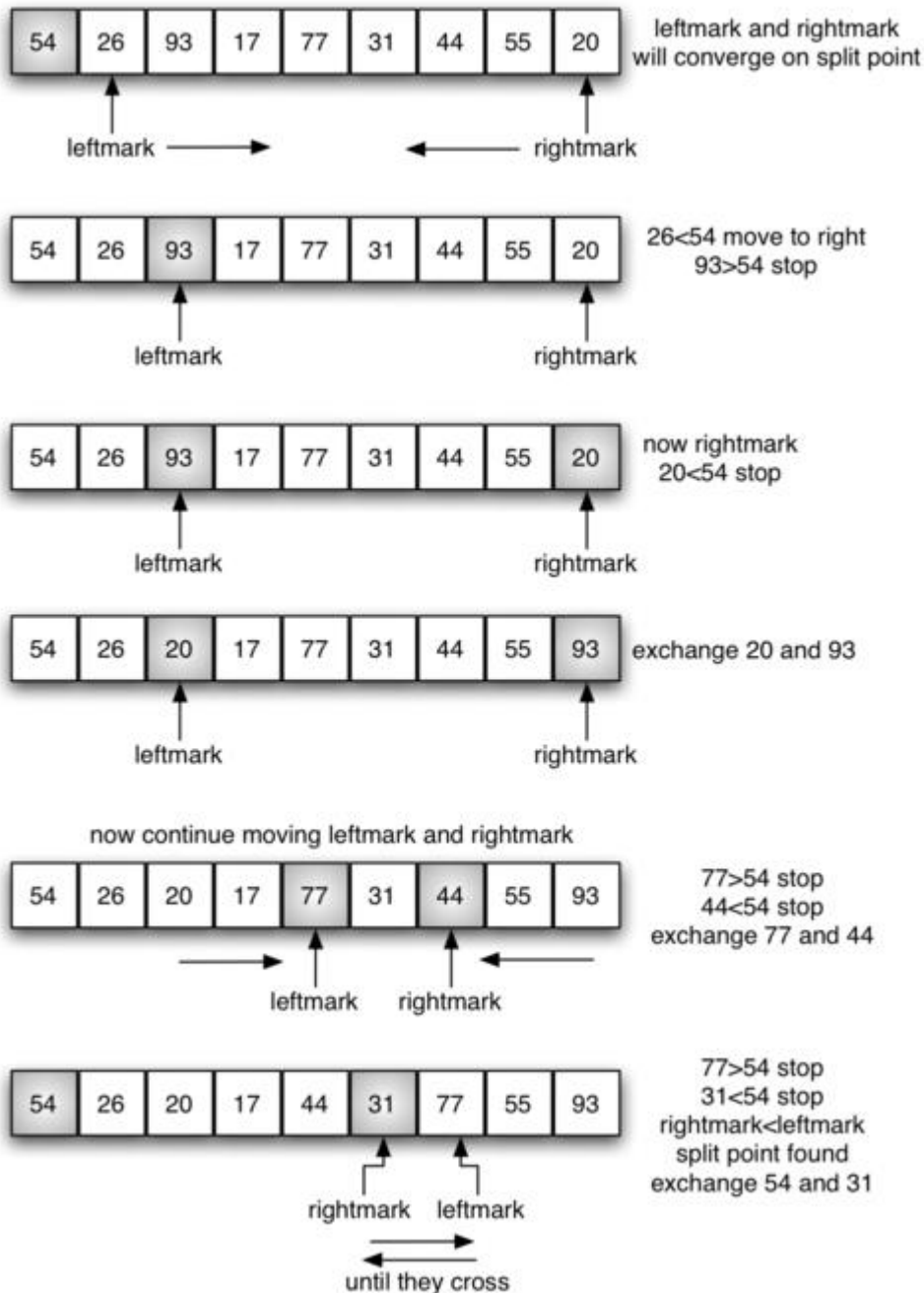
6		<p>Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A.</p>	
7		<p>From A we have D as unvisited adjacent node. We mark it as visited and enqueue it.</p>	
	<p>b) Explain quick sort using an example. Write an algorithm for it and comment on its complexity.</p> <p>ANS:</p> <p>The quick sort uses divide and conquer to gain the same advantages as the merge sort, while not using additional storage. As a trade-off, however, it is possible that the list may not be divided in half. When this happens, we will see that performance is diminished.</p> <p>A quick sort first selects a value, which is called the pivot value. Although there are many different ways to choose the pivot value, we will simply use the first item in the list. The role of the pivot value is to assist with splitting the list. The actual position where the pivot value belongs in the final sorted list, commonly called the split point, will be used to divide the list for subsequent calls to the quick sort.</p> <p>Figure shows that 54 will serve as our first pivot value. Since we have looked at this example a few times already, we know that 54 will eventually end up in the position currently holding 31. The partition process will happen next. It will find the split point and at the same time move other items to the appropriate side of the list, either less than or greater than the pivot value.</p> <div data-bbox="168 1730 964 1806">  </div>		<p>[10 M]</p>

Figure : The First Pivot Value for a Quick Sort

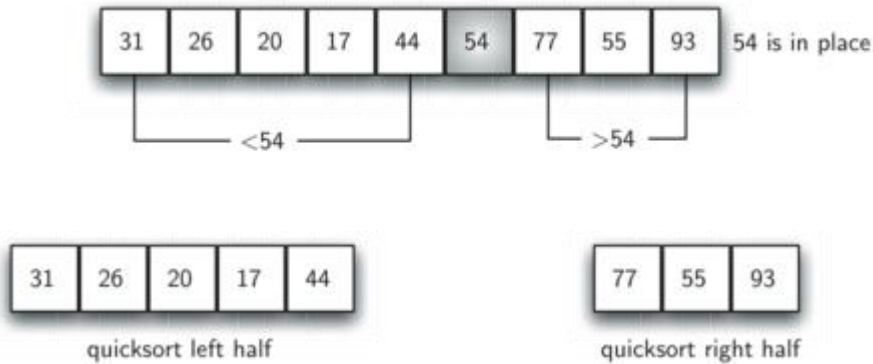
Partitioning begins by locating two position markers—let's call them leftmark and rightmark—at the beginning and end of the remaining items in the list (positions 1 and 8 in Figure). The goal of the partition process is to move items that are on the wrong side with respect to the pivot value while also converging on the split point. Figure shows this process as we locate the position of 54.



We begin by incrementing leftmark until we locate a value that is greater than the pivot value. We then decrement rightmark until we find a value that is less than the pivot value. At this point we have discovered two items that are out of place with respect to the eventual split point. For our example, this occurs at 93 and 20. Now we

can exchange these two items and then repeat the process again.

At the point where rightmark becomes less than leftmark, we stop. The position of rightmark is now the split point. The pivot value can be exchanged with the contents of the split point and the pivot value is now in place. In addition, all the items to the left of the split point are less than the pivot value, and all the items to the right of the split point are greater than the pivot value. The list can now be divided at the split point and the quick sort can be invoked recursively on the two halves.



Completing the Partition Process to Find the Split Point for 54

Algorithm:

QuickSort(array,begin,end)

i. If (begin<end)< p="">

a. divide(array,begin,end,pivot)

b. quickSort(array,begin,pivot-1)

c. quicksort(array,pivot+1,end)

d. END IF ii. STOP

Divide(array,begin,end,pivot)

i. Set Left=Begin , Right=End, pivot=begin

ii. Initialize flag to false.

iii. While Flag=false.

a. While array[pivot]<=array[right] and pivot≠right.

Decrement right by 1.

b. If pivot=right

Set Flag=True.

c. Else if $\text{array}[\text{pivot}] > \text{array}[\text{right}]$

Swap $\text{array}[\text{pivot}]$ and $\text{array}[\text{right}]$.

Set right pointer as pivot.

d. If flag=false

1. While $\text{array}[\text{pivot}] \geq \text{array}[\text{left}]$ and $\text{pivot} \neq \text{left}$

Increment left pointer by 1.

1. IF $\text{pivot} = \text{left}$

Set Flag=false.

2. Else if $\text{array}[\text{pivot}] < \text{array}[\text{left}] < \text{p} = "" >$

Swap $\text{array}[\text{pivot}]$ and $\text{array}[\text{left}]$.

Set left pointer as pivot.

iv. Return pivot position.

- Let us assume that $T(n)$ be the complexity and that all elements are distinct. Recurrence for $T(n)$ depends only on two subproblem sizes which depend on partition element. If pivot is i th smallest element, then exactly $(i-1)$ items will be in the left part and $(n-i)$ in the right part. Since each element has equal probability of being selected as pivot, the probability of selecting i th element is $1/n$.
- In best case: each partition splits array in halves.

$$T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n) \quad T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$$

- In worst case: Each partition gives unbalanced spilt.

$$T(n) = T(n-1) + \Theta(n) = \Theta(n^2) \quad T(n) = T(n-1) + \Theta(n) = \Theta(n^2)$$

- In average case: $T(n) = O(n \log n) \quad T(n) = O(n \log n)$

So, quicksort ranges from $O(n \log n)$ with the best pivots, to $O(n^2)$ with the worst pivots, where n is the number of elements in the array.