

Text Book

http://ressources.unisciel.fr/algoprogram/s00aaroot/aa00module1/res/%5BCormen-AL2011%5DIntroduction_To_Algorithms-A3.pdf

Table Of Contents:

TOPIC	PAGE
Merge-Sort:	2 - 4
Heap-Sort:	5 - 6
Counting-Sort:	7
Radix-Sort:	8
Bucket-Sort:	9 - 10
Breadth-First-Search:	11 - 13
depth-first-search (Graph)	14 - 15
Kruskal (Minimum Spanning Trees)	16 - 18
Prim's (Minimum Spanning Trees)	19 - 21
Strongly Connected Components	22
Dijkstra	23 - 24
Bellman-Ford	25 - 27
Knapsack Problem(Dynamic)	28 - 29
Floyd-Warshall	30 - 31
 Data structures:	
Hash Tables	32
Binary Search Trees	32 - 33
Red-Black Trees	34 - 38
AVL Trees	39 - 41
2-3 Trees	42 - 44

Algorithms:

Merge-Sort (divide and conquer)

- SERIAL Merge-Sort - Page 34

MERGE(A, p, q, r)

```
1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  let  $L[1 \dots n_1 + 1]$  and  $R[1 \dots n_2 + 1]$  be new arrays
4  for  $i = 1$  to  $n_1$ 
5       $L[i] = A[p + i - 1]$ 
6  for  $j = 1$  to  $n_2$ 
7       $R[j] = A[q + j]$ 
8   $L[n_1 + 1] = \infty$ 
9   $R[n_2 + 1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13     if  $L[i] \leq R[j]$ 
14          $A[k] = L[i]$ 
15          $i = i + 1$ 
16     else  $A[k] = R[j]$ 
17          $j = j + 1$ 
```

MERGE-SORT(A, p, r)

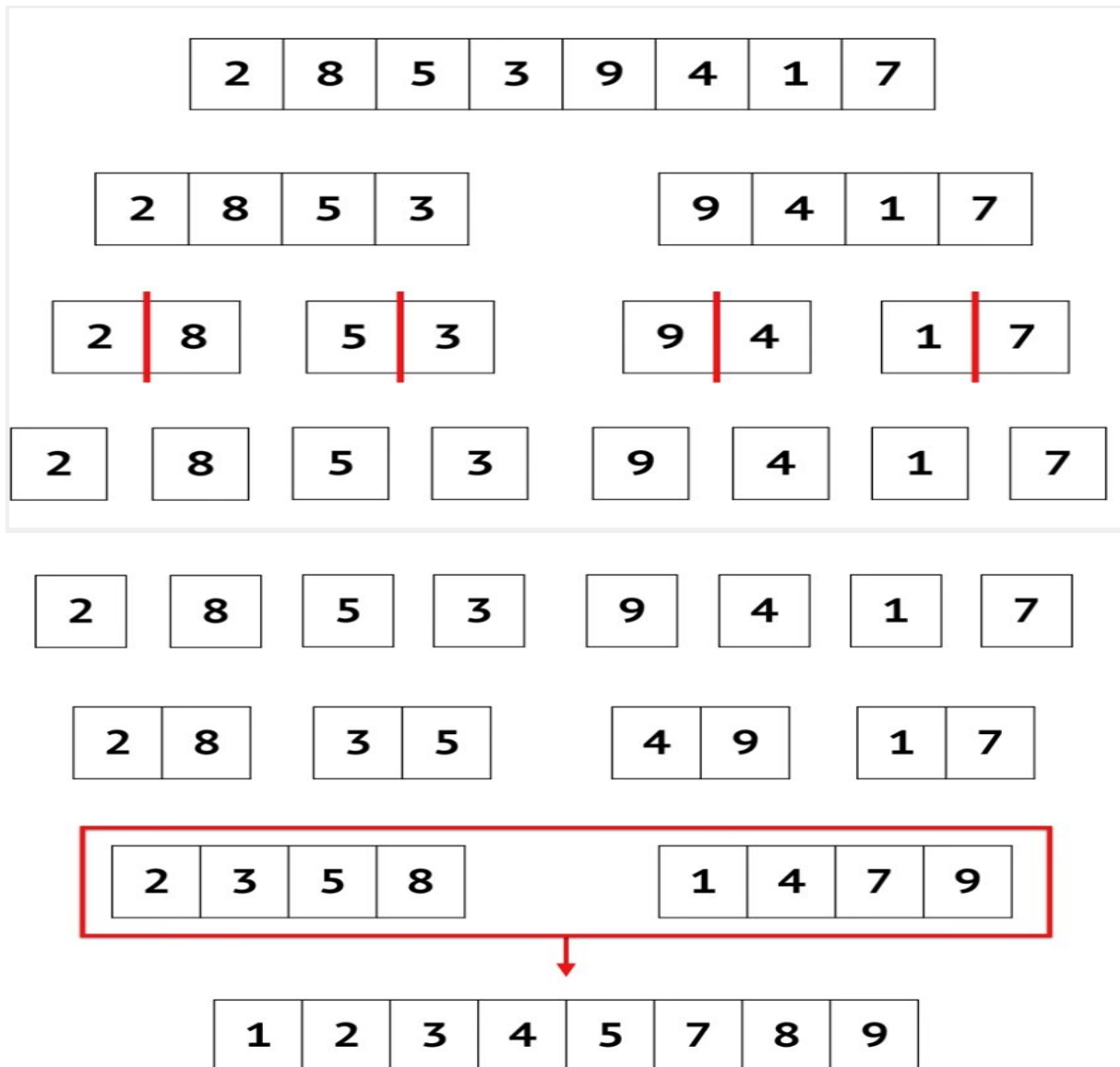
```
1  if  $p < r$ 
2       $q = \lfloor (p + r) / 2 \rfloor$ 
3      MERGE-SORT( $A, p, q$ )
4      MERGE-SORT( $A, q + 1, r$ )
5      MERGE( $A, p, q, r$ )
```

- Usually done recursively, divide and conquer algorithm.

STEPS

1. Continuously split in half until left with individual items.

2. Compare individual item and their neighbor and add the two into temporary arrays.
3. Merge smaller arrays into larger one, inserting in correct order until we have one full array rather than broken ones.



MERGE-SORT'(A, p, r)

1 **if** $p < r$

2 $q = \lfloor (p + r)/2 \rfloor$

3 **spawn** **MERGE-SORT'**(A, p, q)

4 **MERGE-SORT'**($A, q + 1, r$)

5 **sync**

6 **MERGE**(A, p, q, r)

Heap-Sort

- 6.1 Heaps - Page 151

Heap - Ordered binary tree.

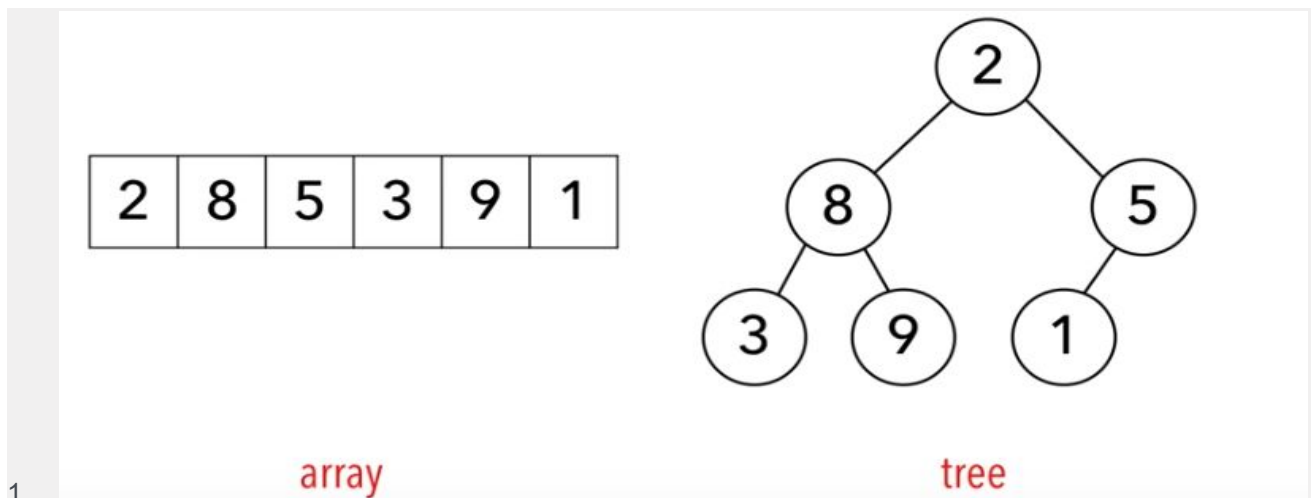
Max heap - Parent > Child

BuildMaxHeap() -> Creates a max heap from an unsorted array

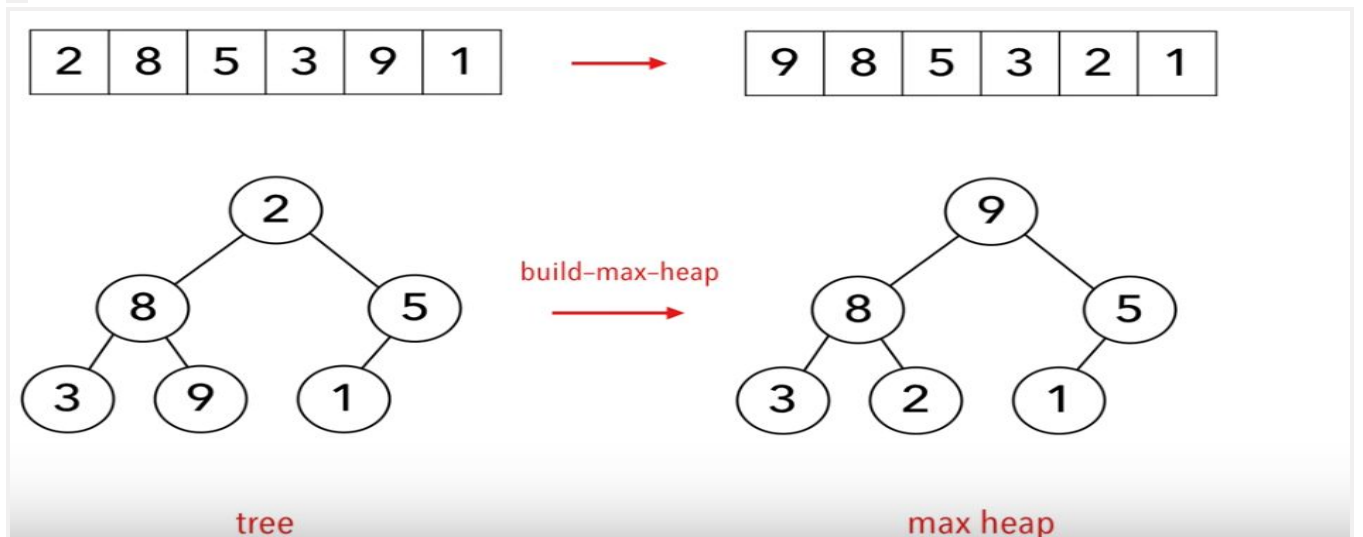
Heapify() -> Similar as BMH but assumes part of array is already sorted (faster)

STEPS

1. Add Array into a tree.
2. Create a max heap
3. Swap the largest item with the smallest and remove the largest item from the max heap
4. Place this item in a sorted partition
5. Run 1-4 again, this time with heapify because the tree is nearly sorted.

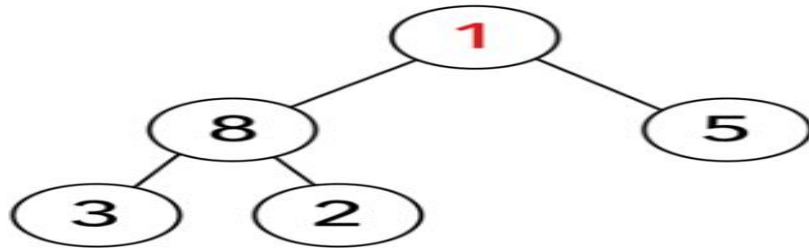


2.



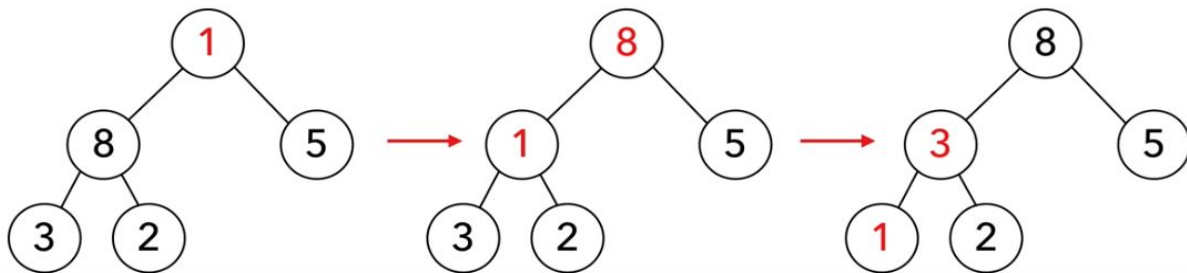
3 & 4.

1	8	5	3	2	9
---	---	---	---	---	---



5.

1	8	5	3	2	9
---	---	---	---	---	---



Counting Sort

- 8.2 Counting sort - Page 194
- Sorting technique based on keys between a specific range
- Counts number of objects with distinct keys, does arithmetic to calculate the position of each object in the output sequence.

STEPS:

1. Create a count array to count the number of instances of each number in the array.

For simplicity, consider data in range of 0 to 9

1	4	1	2	7	5	2
---	---	---	---	---	---	---

Create a count array to store the count of each unique object

For simplicity, consider data in range of 0 to 9

1	4	1	2	7	5	2
---	---	---	---	---	---	---

Index :	0	1	2	3	4	5	6	7	8	9
	0	2	2	0	1	1	0	1	0	0

2. Modify the count array by adding the previous counts.

Index :	0	1	2	3	4	5	6	7	8	9
	0	2	4	0	1	1	0	1	0	0

$2 + 2$

Index :	0	1	2	3	4	5	6	7	8	9
	0	2	4	4	5	6	6	7	7	7

3. Place the array objects in their correct positions and decrease their count by 1.

	1	4	1	2	7	5	2
--	---	---	---	---	---	---	---

Index :	0	1	2	3	4	5	6	7	8	9
	0	2	4	4	5	6	6	7	7	7

Places :	1	2	3	4	5	6	7
		1					

	1	4	1	2	7	5	2
--	---	---	---	---	---	---	---

Index :	0	1	2	3	4	5	6	7	8	9
	0	0	3	4	4	5	6	6	7	7

Places :	1	2	3	4	5	6	7
	1	1	2	2	4	5	7

Radix Sort

- 8.3 Radix sort - Page 197
- Sort numbers from least significant digit to most significant digit.
- Use counting sort as a subroutine to sort these numbers.

STEPS:

Consider this input array

170	45	75	90	802	24	2	66
-----	----	----	----	-----	----	---	----

1. First consider the ones place

170	45	75	90	802	24	2	66
-----	----	----	----	-----	----	---	----

170	90	802	2	24	45	75	66
-----	----	-----	---	----	----	----	----

2. Then consider the tens place

170	90	802	2	24	45	75	66
-----	----	-----	---	----	----	----	----

802	2	24	45	66	170	75	90
-----	---	----	----	----	-----	----	----

3. Then consider the hundreds place

802	2	24	45	66	170	75	90
-----	---	----	----	----	-----	----	----

2	24	45	66	75	90	170	802
---	----	----	----	----	----	-----	-----

4. And so on if larger numbers.

Bucket Sort

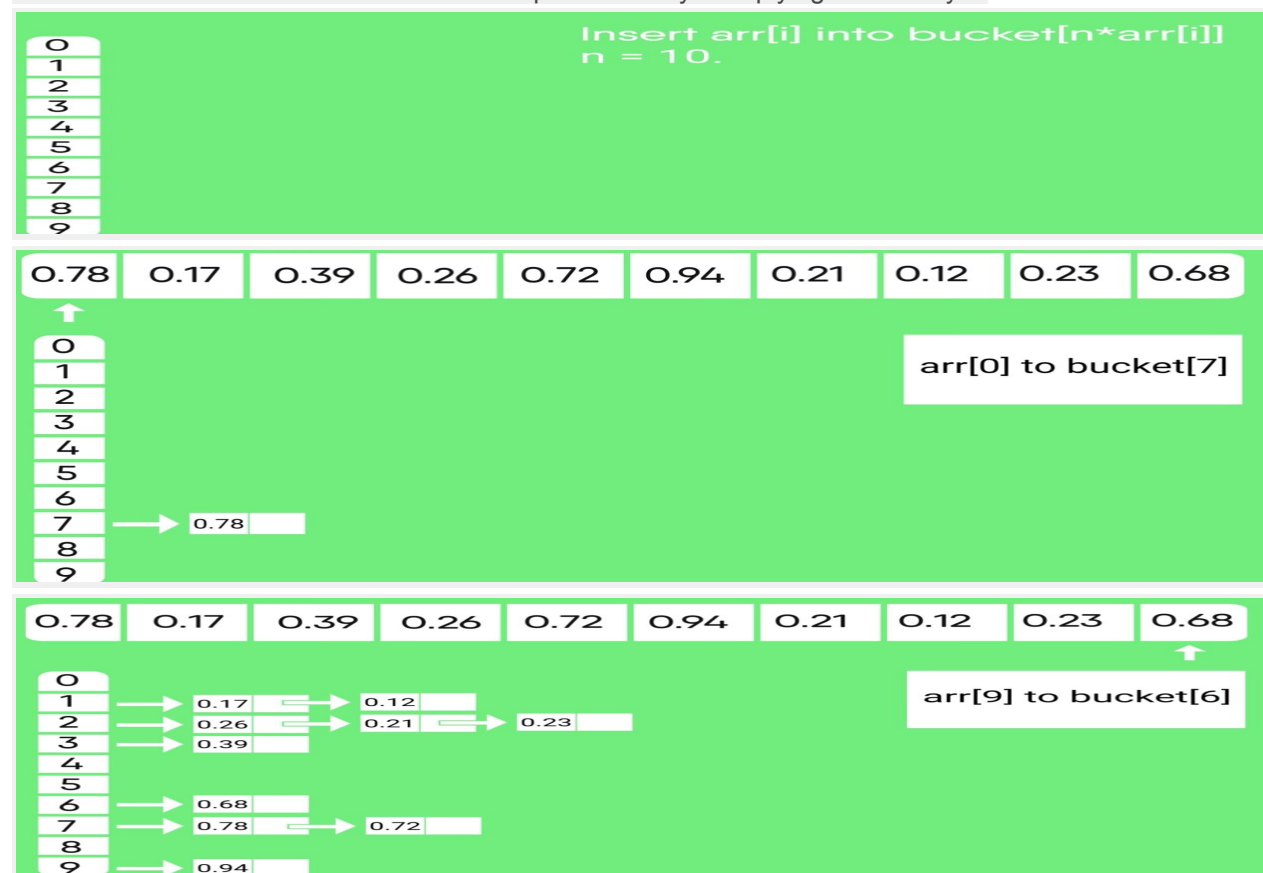
- 8.4 Bucket sort - Page 200
- Useful when sorting decimal values.
- Create buckets to put elements in. Sort the elements in each bucket (using insertion sort). Join all buckets to get sorted array.

STEPS:

1. Create buckets based on total number of elements



2. Insert elements into buckets. Find bucket placement by multiplying number by n .



3. Sort Buckets using Insertion Sort



4. Add buckets, in order, into output array

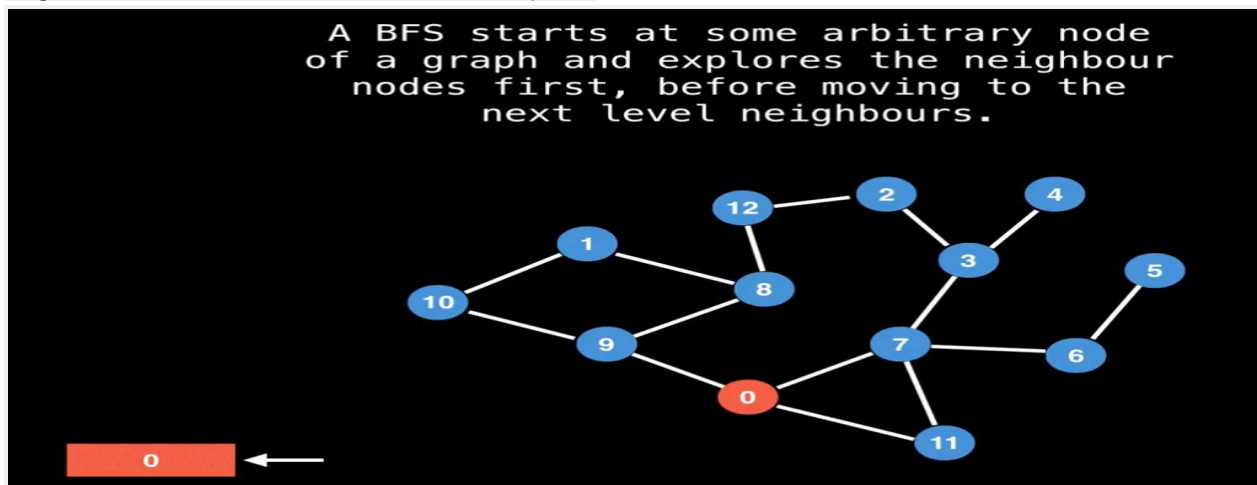
0.12	0.17	0.21	0.23	0.26	0.39	0.68	0.72	0.78	0.94
------	------	------	------	------	------	------	------	------	------

Breadth-First-Search (Graph)

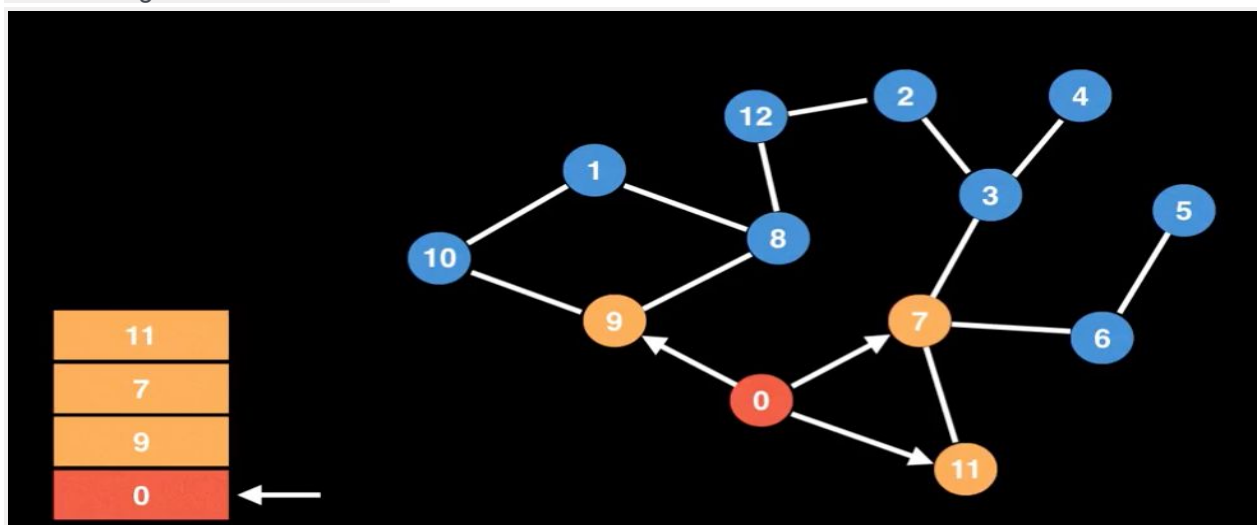
- 22.2 Breadth-first search - Page 594
- BFS is particularly useful in finding shortest paths on unweighted graphs.
- BFS starts at a node s in a graph and explores all its neighbor nodes before moving to the next level (neighbors of neighbors). (Explores nodes in "layers".)
- Uses Queue
- Each node has adjacency list with pointers to neighboring nodes.
- BFS is used for :
 - Used to explore nodes and edges of graph.
 - FINDING THE SHORTEST PATH ON UNWEIGHTED GRAPHS
 - Testing whether graph is connected.
 - Computing a spanning forest of graph.
 - Computing, for every vertex in graph, a path with the minimum number of edges between start vertex and current vertex or reporting that no such path exists. •
 - Computing a cycle in graph or reporting that no such
- COMPLEXITY = $O(V+E) \rightarrow O(\text{Vertices} + \text{Edges})$

STEPS:

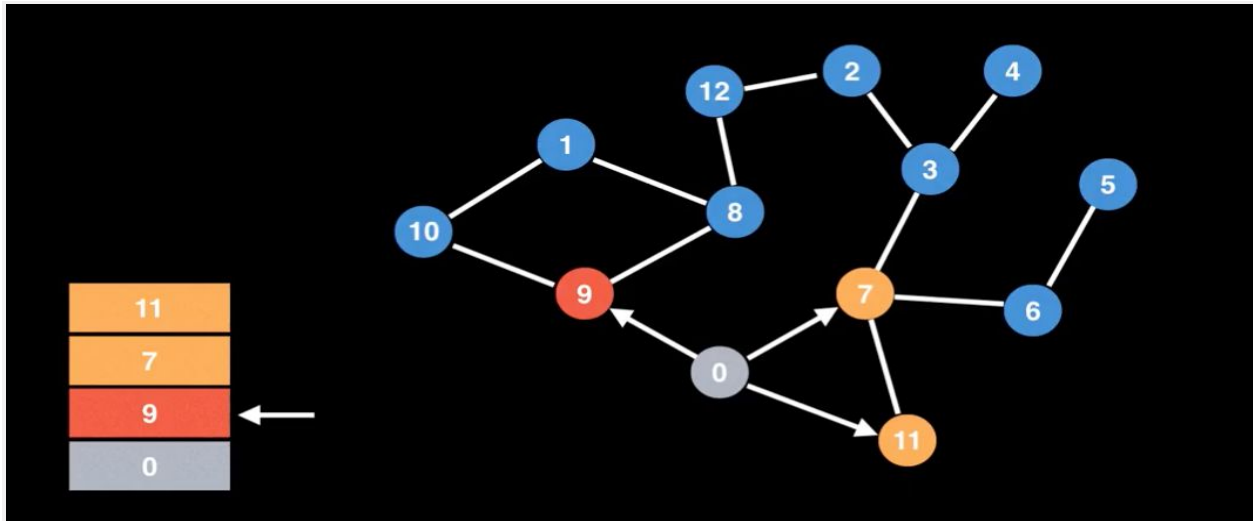
Begin With a chosen node and add it to the queue



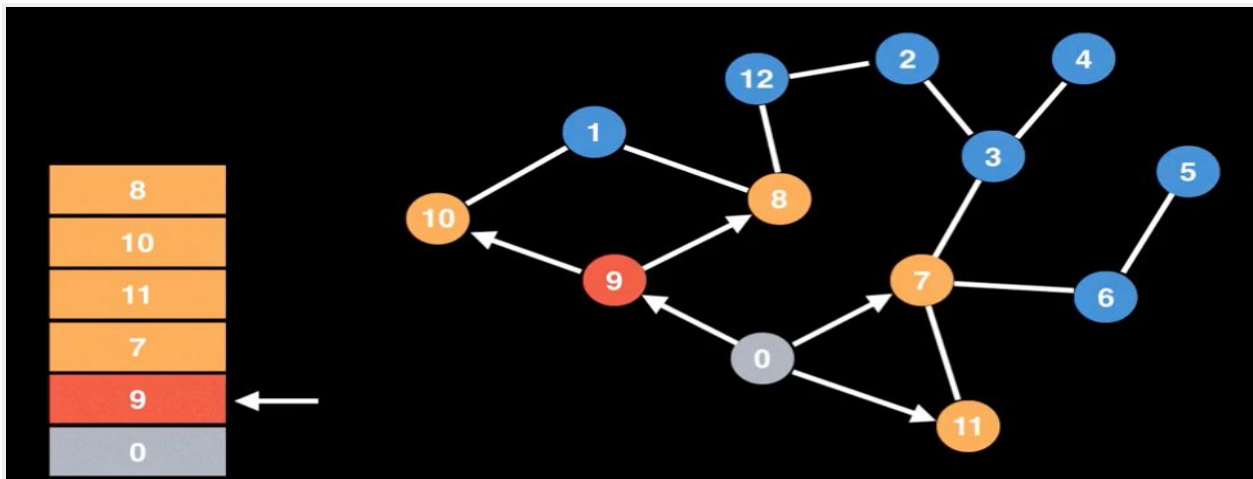
Add it's Neighbors To the Queue



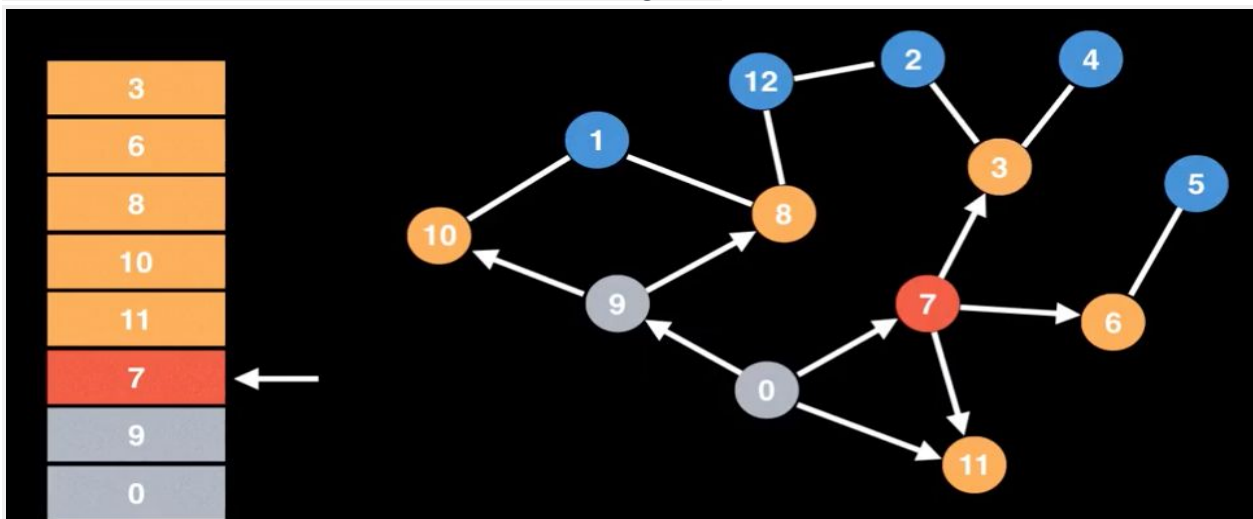
Go To The Next Element in the Queue



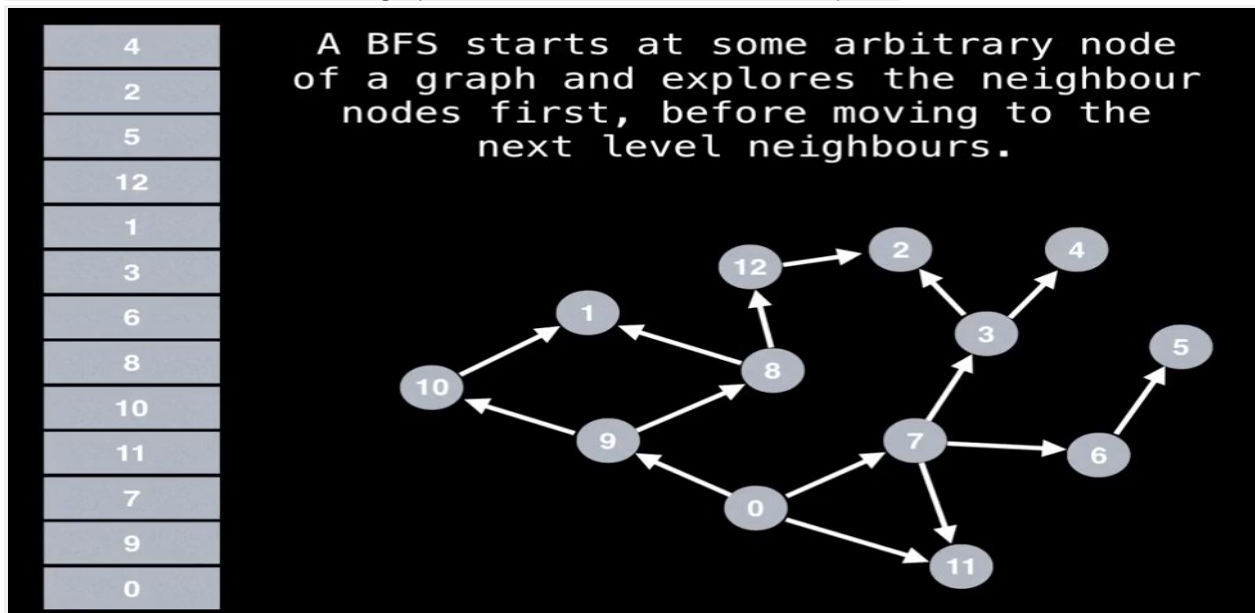
Add It's Neighbors to the Top of the Queue



Go To The Next Element in the Queue and Add its neighbors



Continue until each node in the graph has been visited in order of the queue



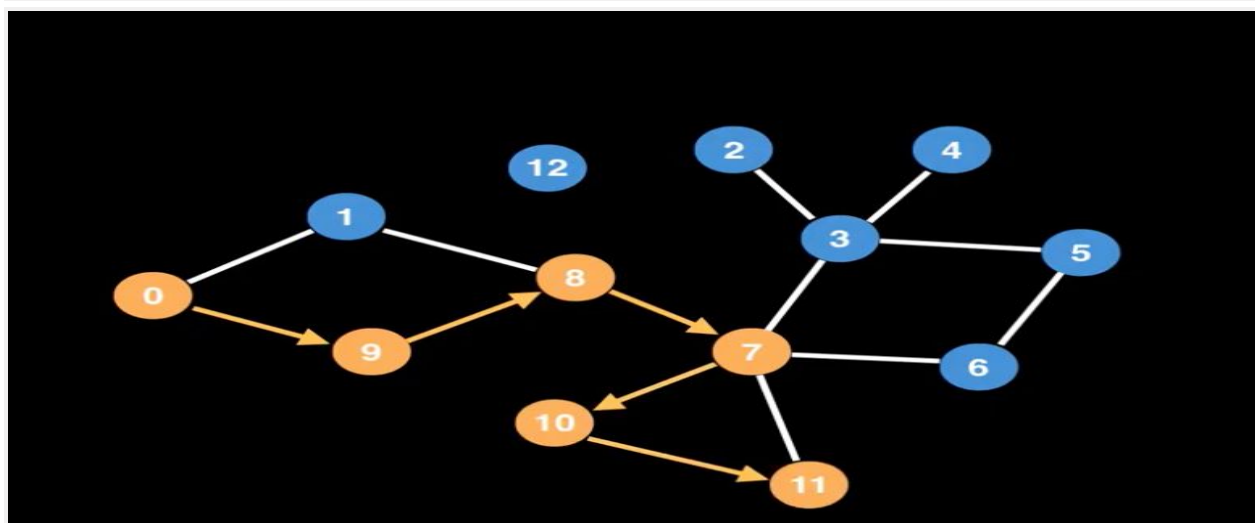
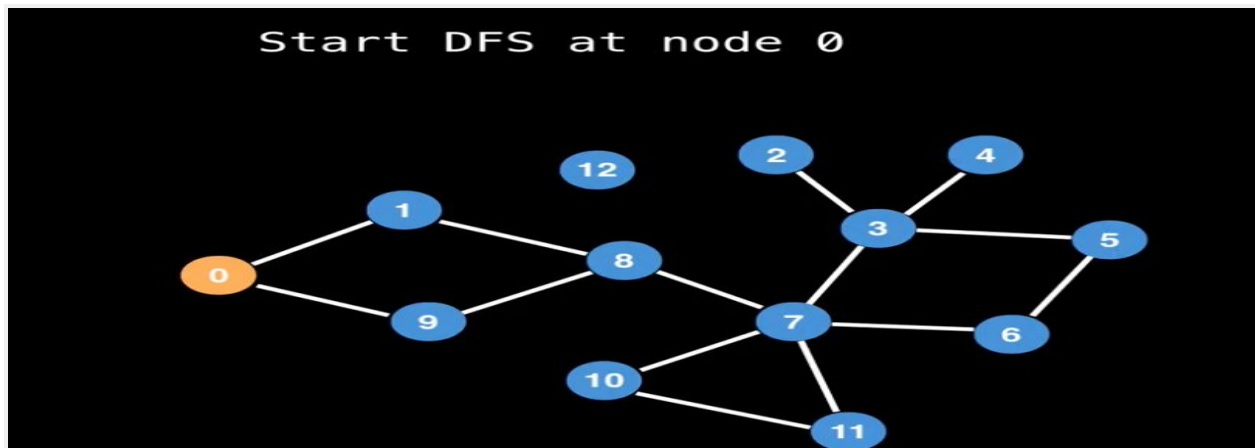
When Using Queue -> When add a node to the queue we Enqueue(node). After visiting this node we Dequeue(Node).

Depth-First-Search (Graph)

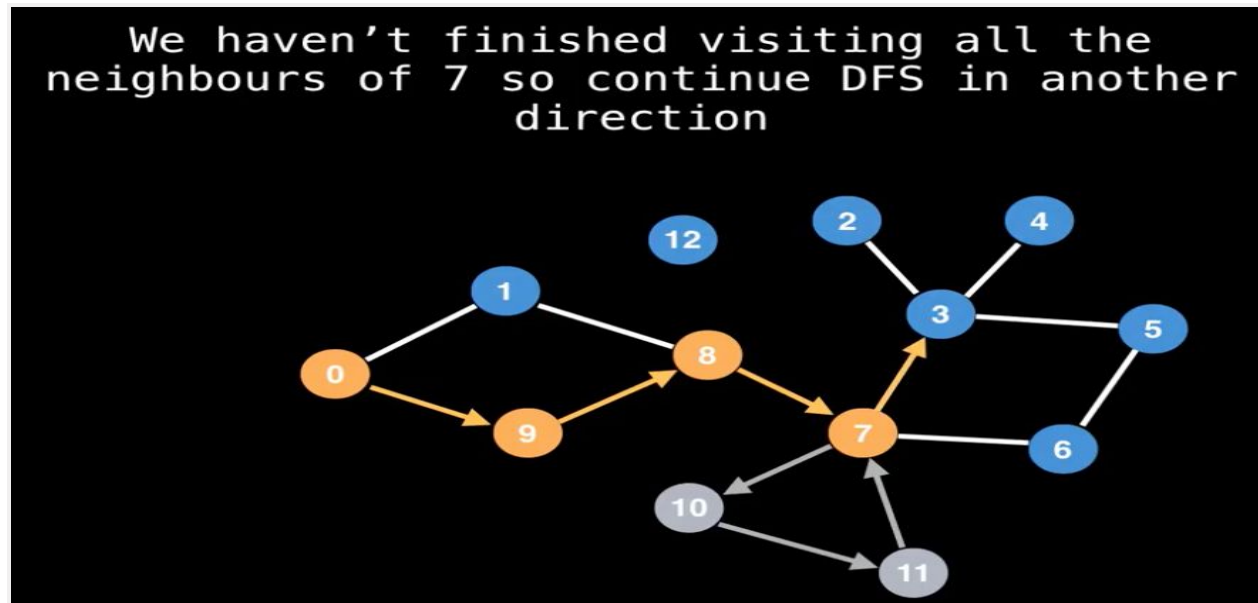
- 22.3 Depth-first search - Page 603
- 2 versions: Stack and recursion
- Tree edges: Edges included in depth-first forest. Edge (u,v) is a tree edge if v was first discovered by edge (u,v) .
- Back edges: Edge (u,v) connects a vertex u to an ancestor (non-parent) v in a depth-first tree.
- Forward edges: Edge (u,v) connects a vertex u to a descendant (non-child) v in a depth-first tree.
- Cross edges: All other edges, i.e., between sibling nodes (e.g., nodes on different branches) of the same depth-first tree or between nodes in different depth-first trees.
- Used for Finding Strongly Connected Components of a Digraph
- topological sort (Graph)
- 22.4 Topological sort - Page 612
- IDEA: go forward, in depth until you cannot any more, then backtrack until you can go forward again.
- COMPLEXITY: $O(V + E) \rightarrow O(\text{Vertices} + \text{Edges})$
- USED FOR: Count connected components, connectivity between nodes, etc.

STEPS:

Begin at a node and arbitrarily choose a neighbor to travel to and continue this process until reach a deadend



Once We reach a dead end, we back track until there are unvisited node options



Continue this process until all nodes have been visited

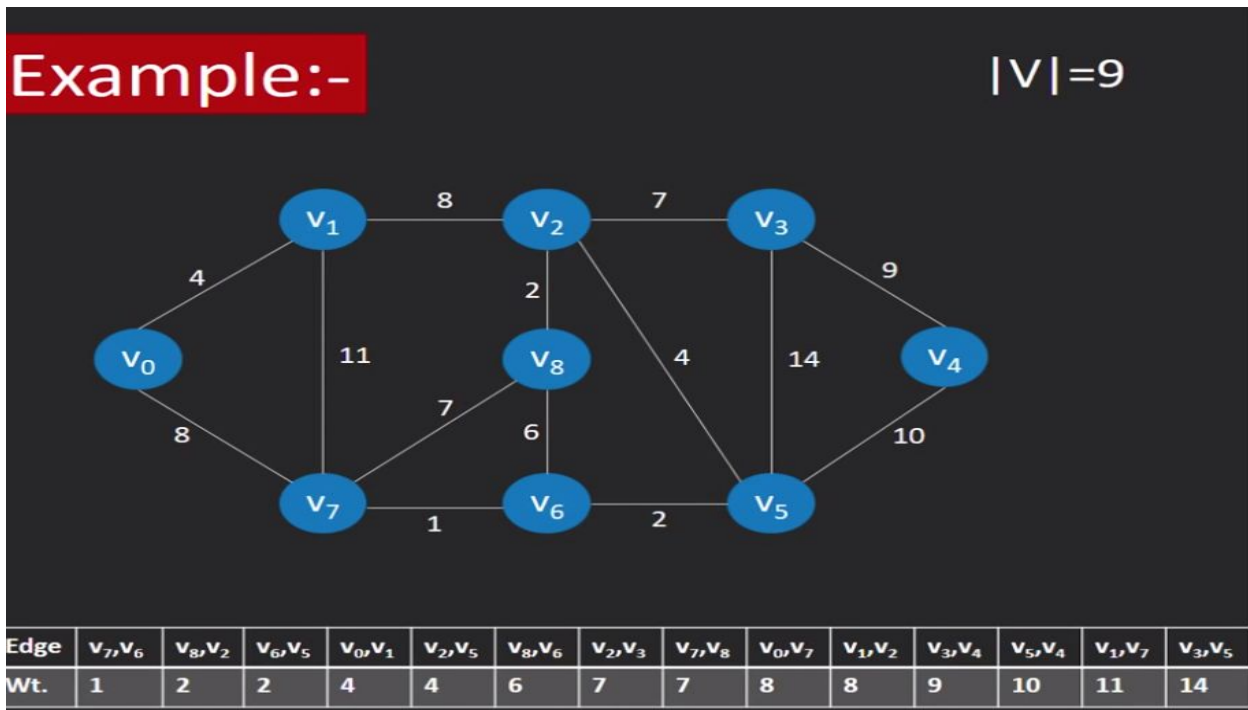
Kruskal (Minimum Spanning Trees)

- Kruskal's algorithm - Page 631
- Outputs a Minimal Spanning tree:
 - Properties of Minimal Spanning Trees:
 - has no cycles (i.e., is a tree),
 - connects all nodes in V , and
 - has a sum of edge weights that is minimum over all possible spanning trees of G .
- Runs in $O(E \cdot \log(E))$ time, or equivalently, $O(E \cdot \log(V))$ time
 - These times are equal since
- Use a disjoint-set data structure to keep track of which vertices are in which components
 - Make-Set(x): $O(1)$
 - Find-Set(x): $O(n)$
 - Union(x, y): $O(1)$
- The running time for m disjoint-set operations on n elements is: $O(m \cdot \lg(n))$
- **Spanning Tree**: Given connected and undirected graph, a spanning tree is a subgraph that is a tree and connects all the vertices together.
- **Minimum Spanning Tree**: The spanning tree of the graph whose sum of weights of edges is minimum.

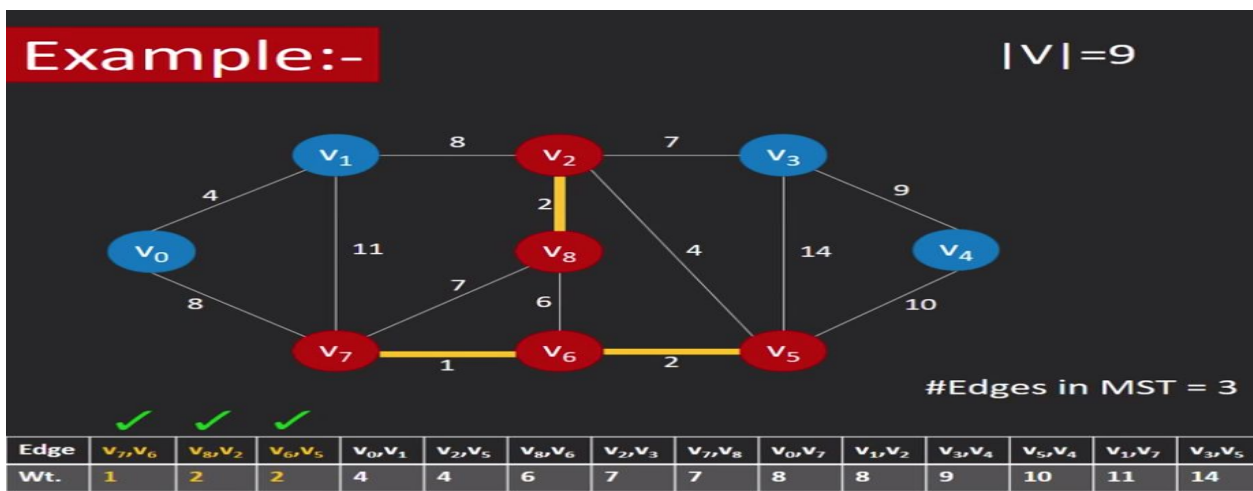
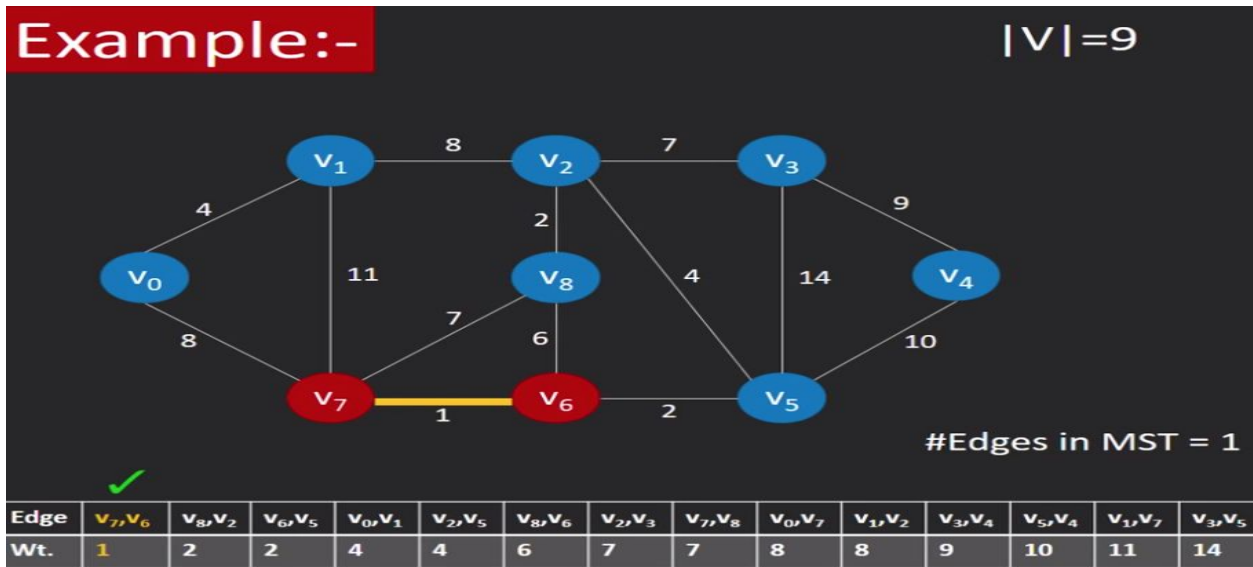
STEPS:

1. Sort all edges in non-decreasing order of their weight
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree thus far.
 - a. If cycle is formed \rightarrow discard it
 - b. Else include this edge
3. Repeat step 2 until there are $V - 1$ edges in the spanning tree

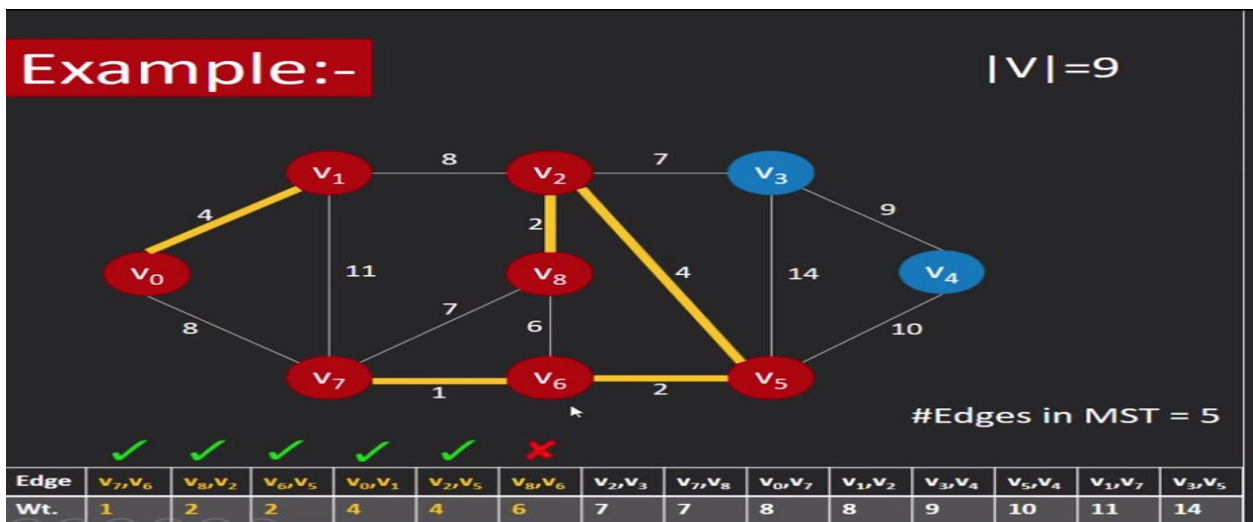
1. Sort Edges by weight



2. Choose edges in order on the condition that they do not form a cycle



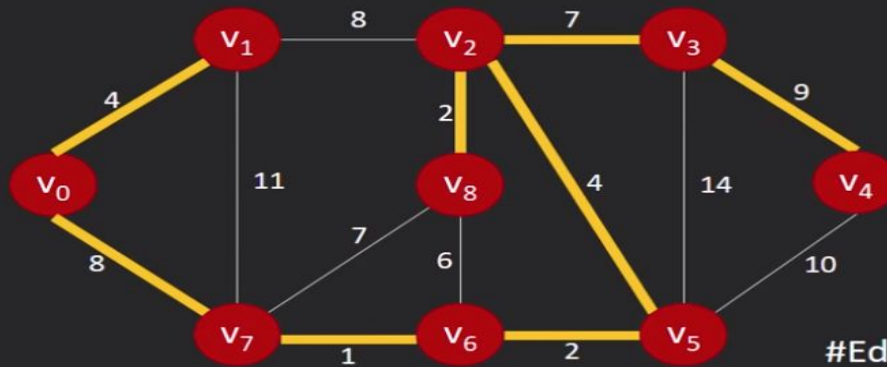
Discard edges that form a cycle



Terminate once we have $(V - 1)$ Edges

Example:-

$|V|=9$



#Edges in MST = 8
Terminate!

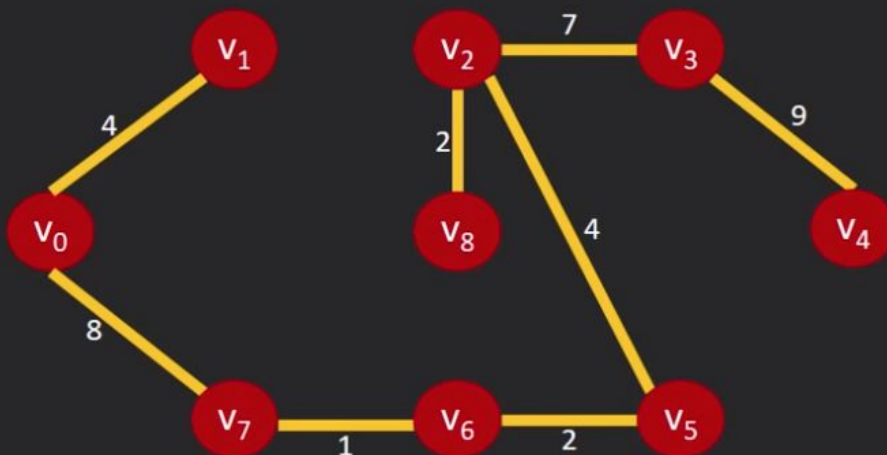
	✓	✓	✓	✓	✓	✗	✓	✗	✓	✗	✓			
Edge	V_7, V_6	V_8, V_2	V_6, V_5	V_0, V_1	V_2, V_5	V_8, V_6	V_2, V_3	V_7, V_8	V_0, V_7	V_1, V_2	V_3, V_4	V_5, V_4	V_1, V_7	V_3, V_5
Wt.	1	2	2	4	4	6	7	7	8	8	9	10	11	14

Calculate the weight

Example:-

We now have our MST !

Weight of Minimum Spanning Tree = **37**



Prim's (Minimum Spanning Trees)

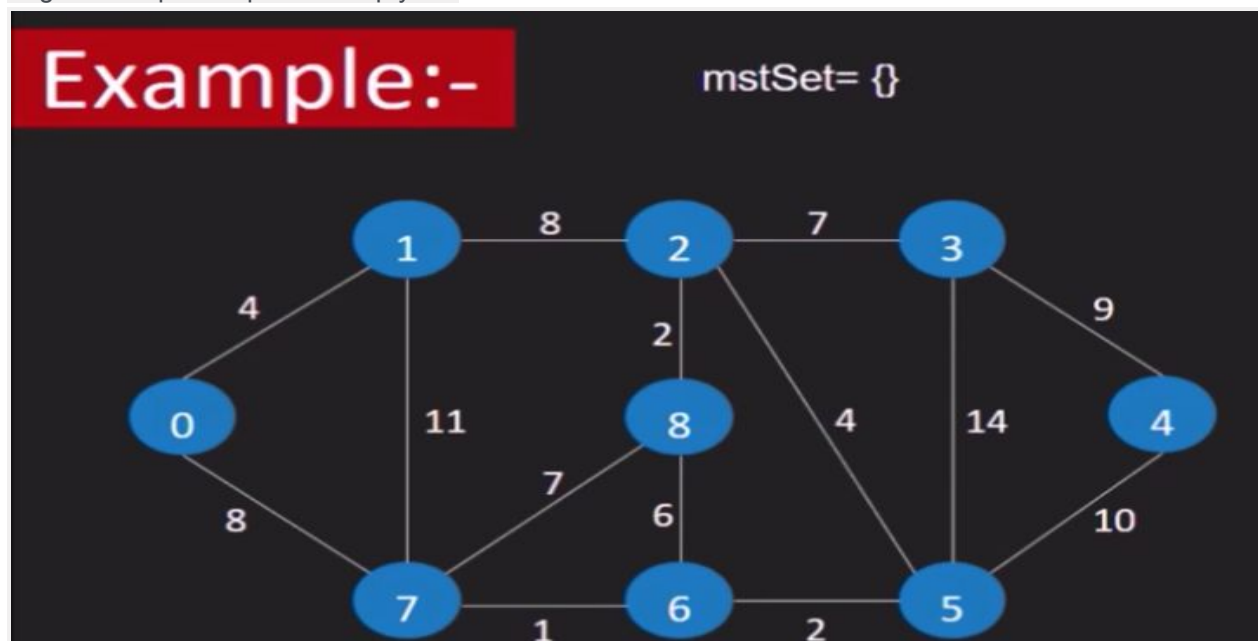
- Prim's Algorithm - Page 634
- Outputs a Minimal Spanning tree:
 - Properties of Minimal Spanning Trees:
 - has no cycles (i.e., is a tree),
 - connects all nodes in V , and
 - has a sum of edge weights that is minimum over all possible spanning trees of G .
- Run Time for different data structures because adding a new edge means that we need to sort the data structure :
 - **adjacency matrix**, searching: $O(|V|^2)$
 - **Binary Heap** (stores input & sorts) and **Adjacency list** (Stores path output):
 - $O((|V| + |E|) * \log(|V|)) = O(|E| * \log(|V|))$
 - **Fibonacci Heap** (stores input & sorts) and **adjacency list** (Stores path output):
 - $O(|E| + |V| * \log(|V|))$
 -

STEPS:

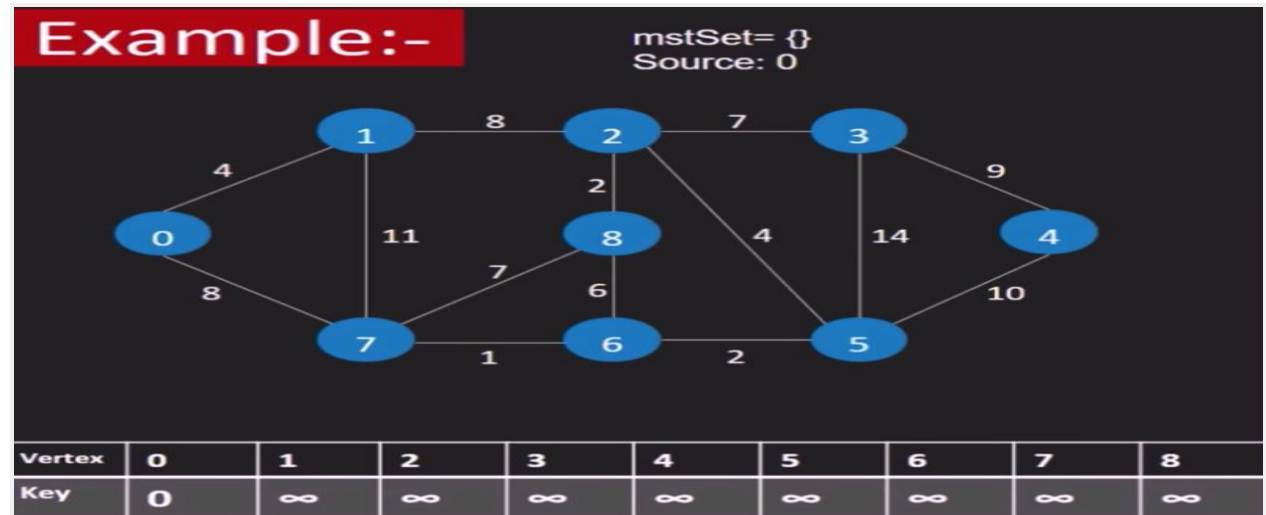
1. Create a set that keeps track of the vertices already included in the Minimum Spanning Tree
2. Assign a key value to all vertices on the input graph. Initialize all key values to be INFINITY. Assign key value of 0 for the first vertex so it is picked first.
3. While(SET does not include all vertices):
 - a. Pick a vertex U which is not in the set and has a minimum key value.
 - b. Add U to Set
 - c. Update key value of all adjacent vertices of U

UPDATING* -> When updating key value, iterate through adjacent vertices. For all adjacent vertices, is weight of edge (U - Adjacent) is less than previous key value, update key value to be (U - V).

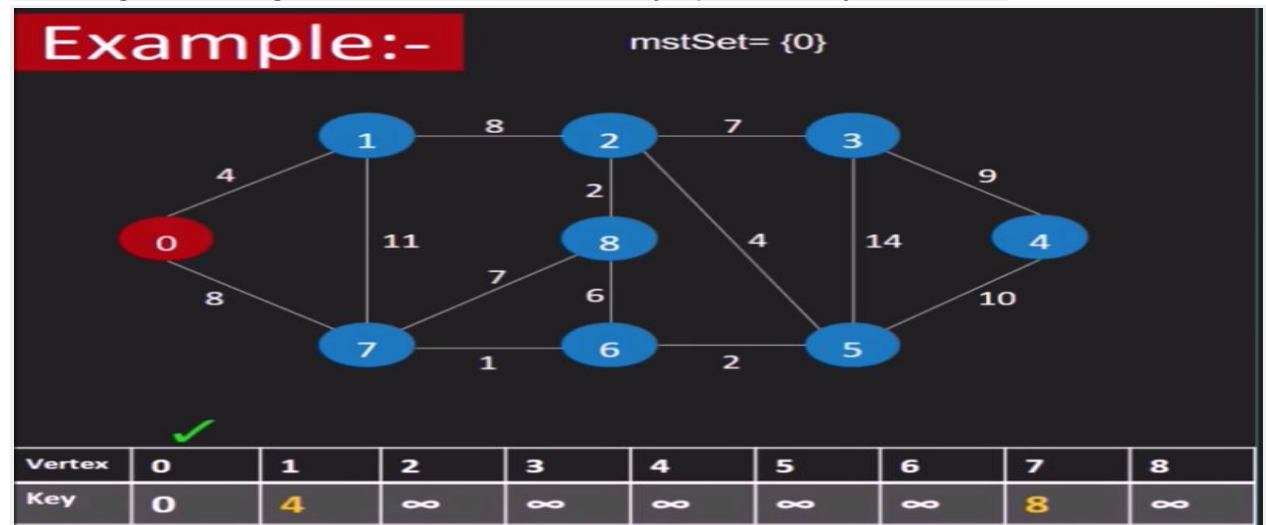
Begin with input Graph and Empty Set



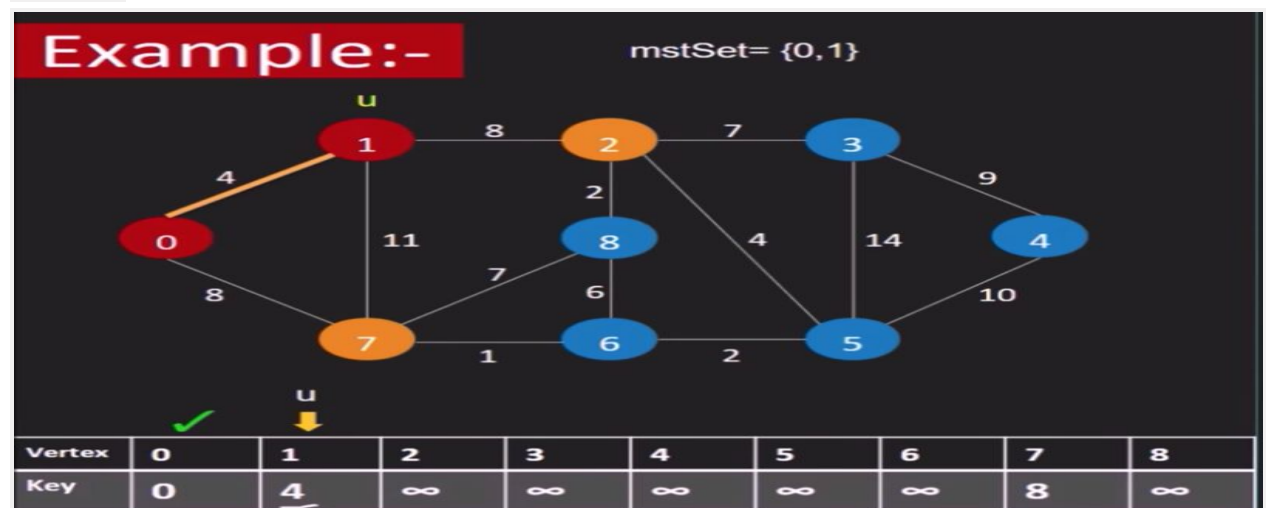
Initialize the key of all vertices but the source to infinity



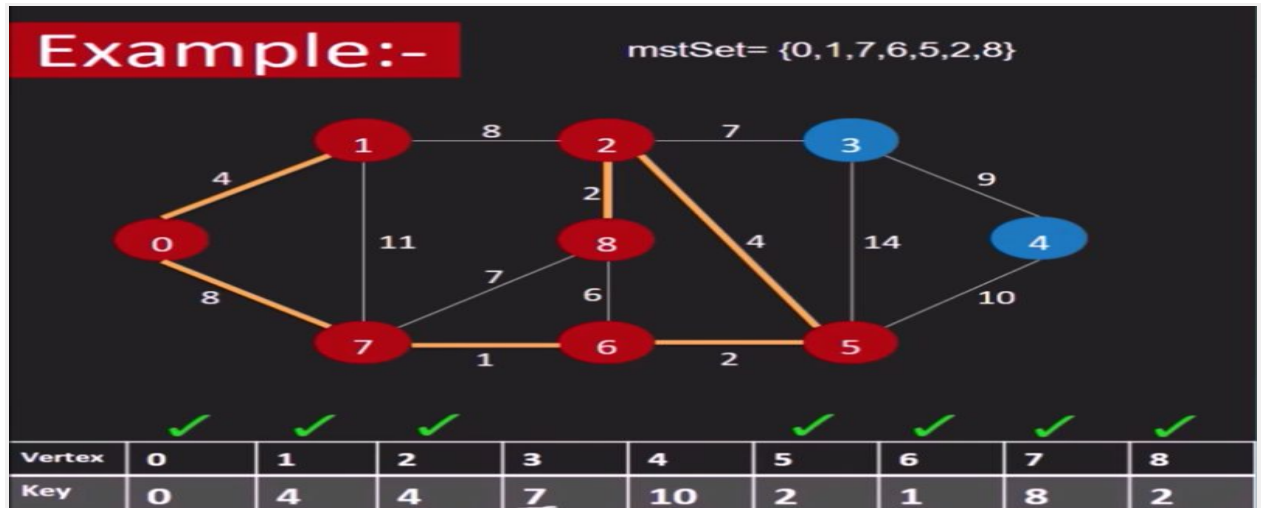
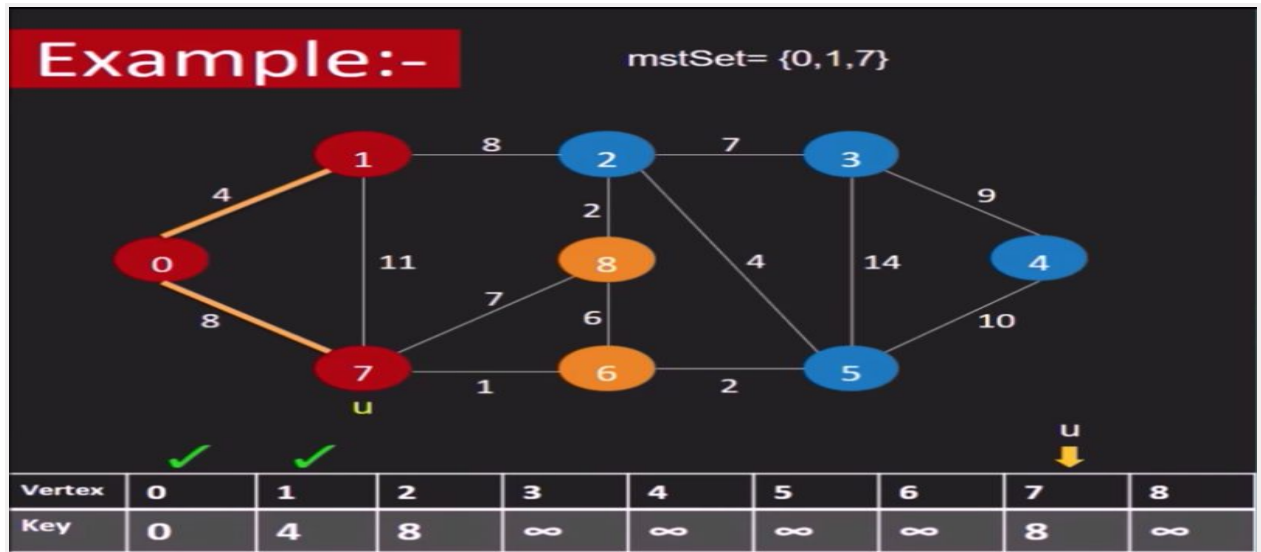
If the weight of the neighbors is less than its current key, update the key to that value



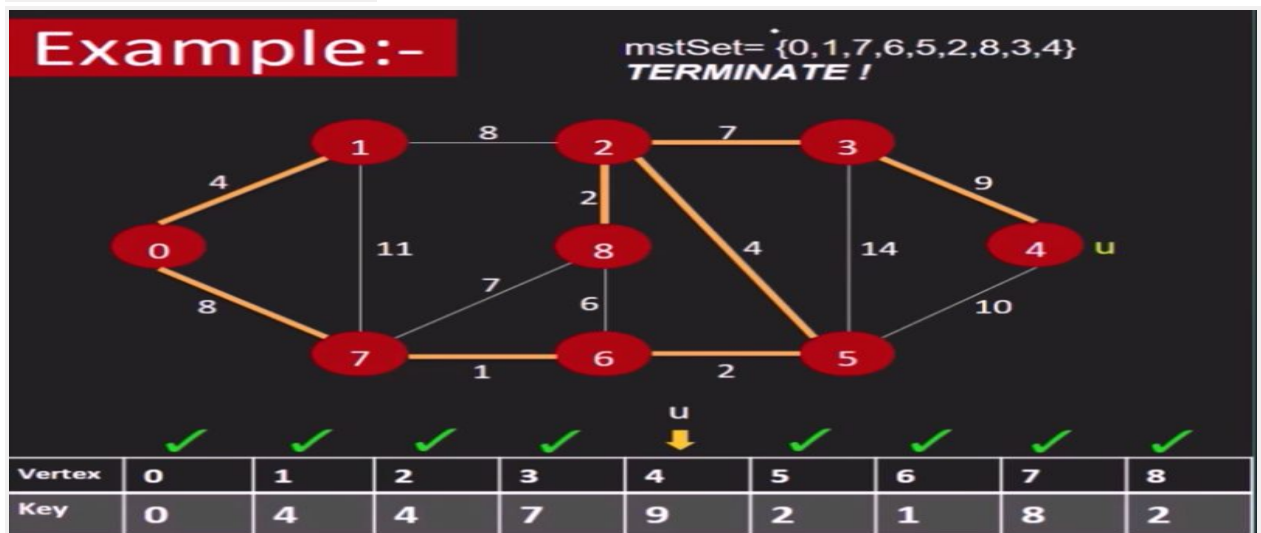
Pick the Vertex with the minimum key value (that is not in the Set) and add it to the Set and create connection



Repeat. Update the key if the path is less than its current Key. Otherwise keep it and choose Min.



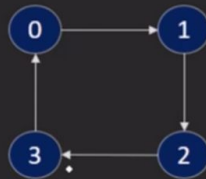
When All are Added. Terminate



Strongly Connected Components

- Page 1170
- A directed graph is strongly connected if every two vertices are reachable from each other. The strongly connected components of a directed graph are the equivalence classes of vertices under the “are mutually reachable” relation. A directed graph is strongly connected if it has only one strongly connected component.
- A directed graph is strong connected if there is a path between all sets of vertices.
- A strongly connected component of a directed graph is a maximal strongly connected subgraph.

1. A directed graph is **strongly connected** if there is a path between all pairs of vertices.



2. A **strongly connected component (SCC)** of a directed graph is a maximal strongly connected subgraph.



Strongly Connected Components

1. 1-0-2
2. 3
3. 4

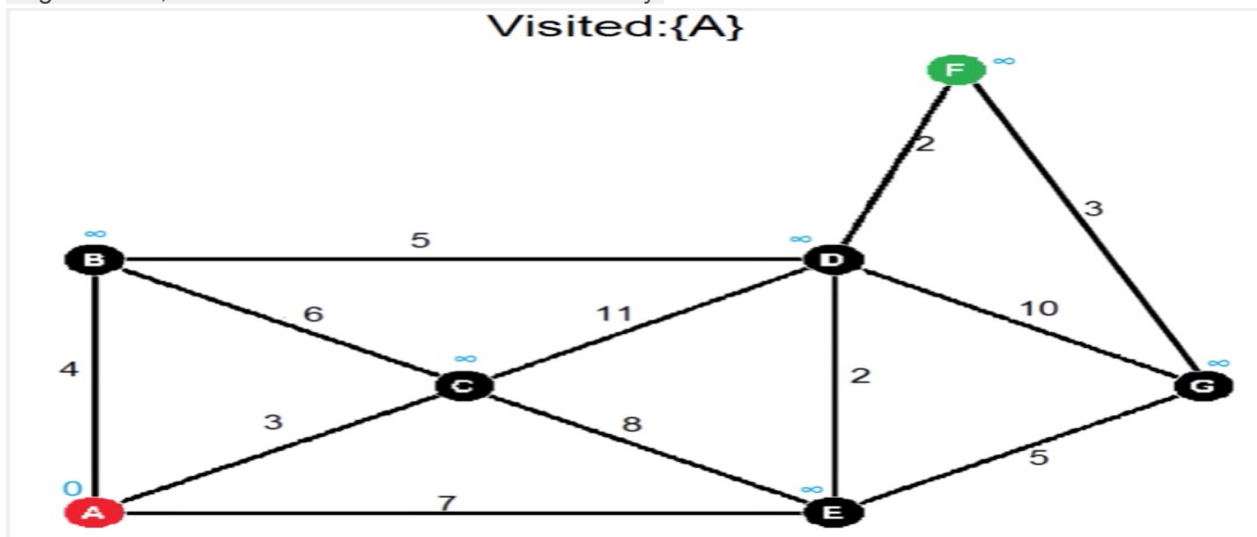
Dijkstra

- 24.3 Dijkstra's algorithm - Page 658
- Finds the shortest path on a weighted graph. Takes in an initial node and a goal node and finds min path between them.

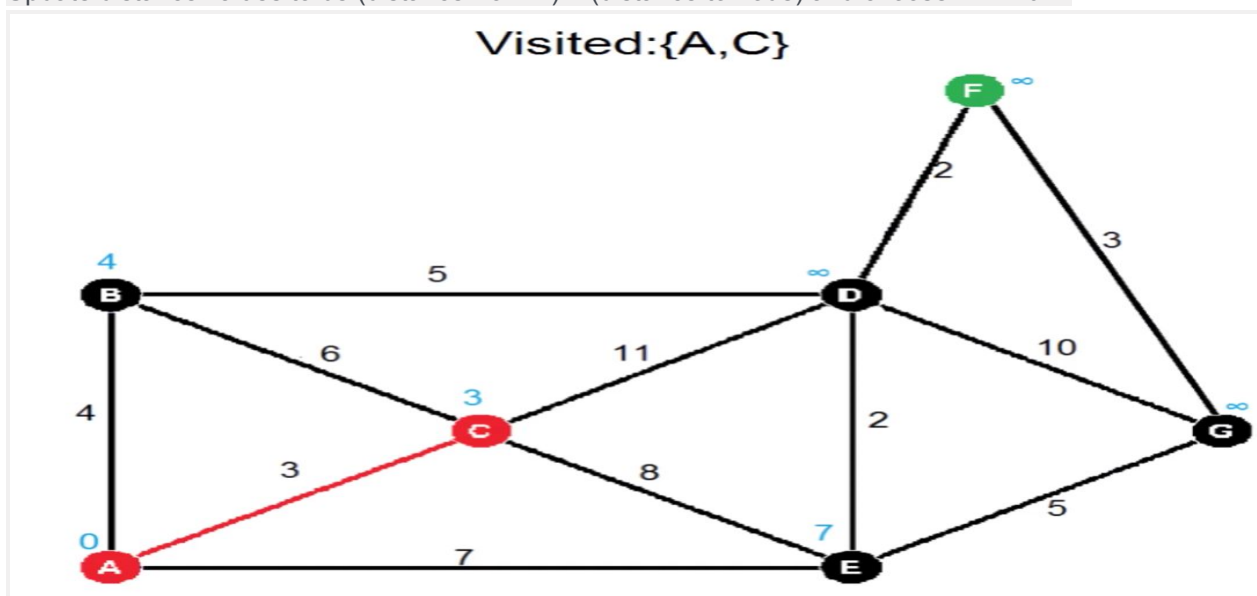
STEPS:

- Assign tentative distance value. Set it to 0 for our initial node and INFINITY for all other nodes.
- Keep a set of visited nodes, initially just the origin node.
- From current node, calculate the distance to all neighbors.
 - (distance to current node) + (distance from current to neighbor)
 - If this is less than current tentative distance, replace it with this value.
- When we have visited all neighbors of current node, make the node as visited and remove it from unvisited set. (Add it to the visited set?)
- If destination node has been marked visited, we are done.

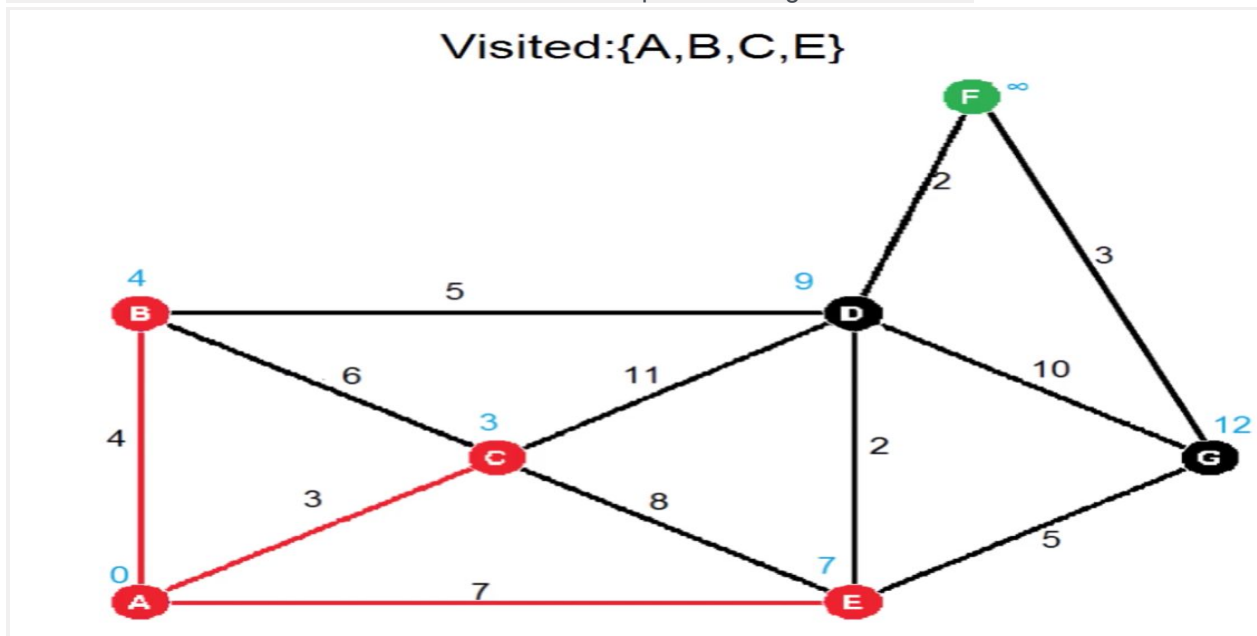
Begin With A, Go to F. Set all distance values to infinity.



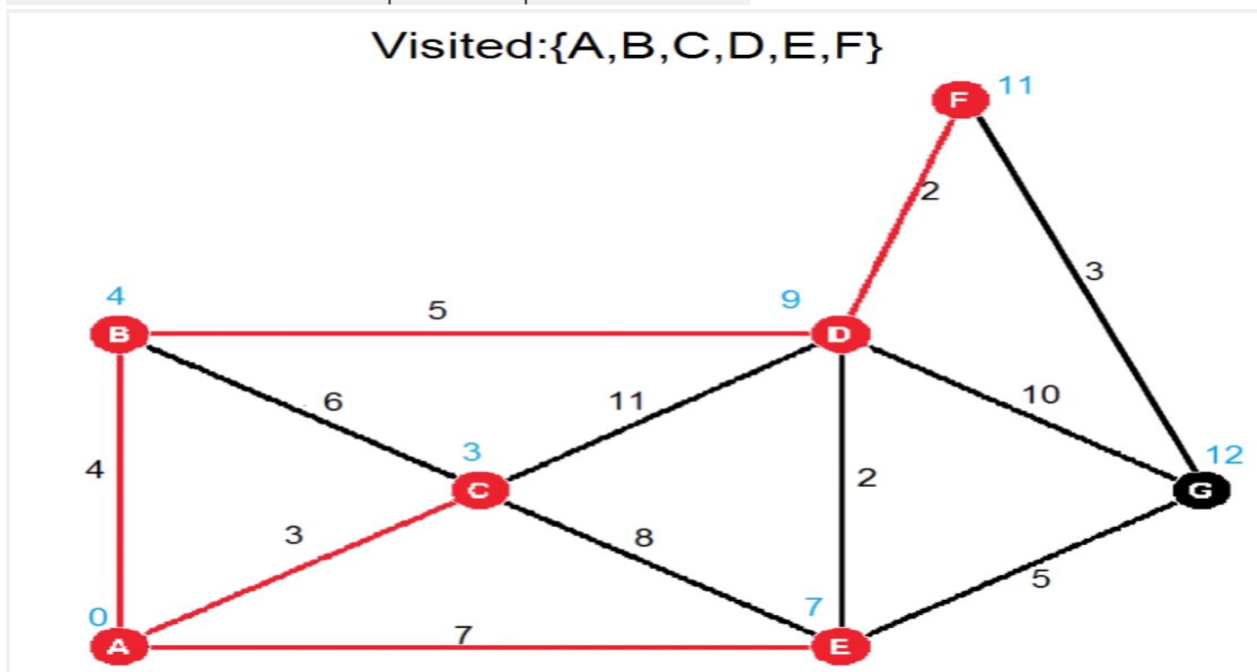
Update distance values to be (distance from A) + (distance to node) and choose minimum.



Continue to travel to the smallest distance node and update its neighbors distance



Have reached destination so stop. Shortest path = A->B->D->F



TIME COMPLEXITY

With **Binary Heap** -> $O((|E| + |V|)\log(|V|))$

SPACE COMPLEXITY

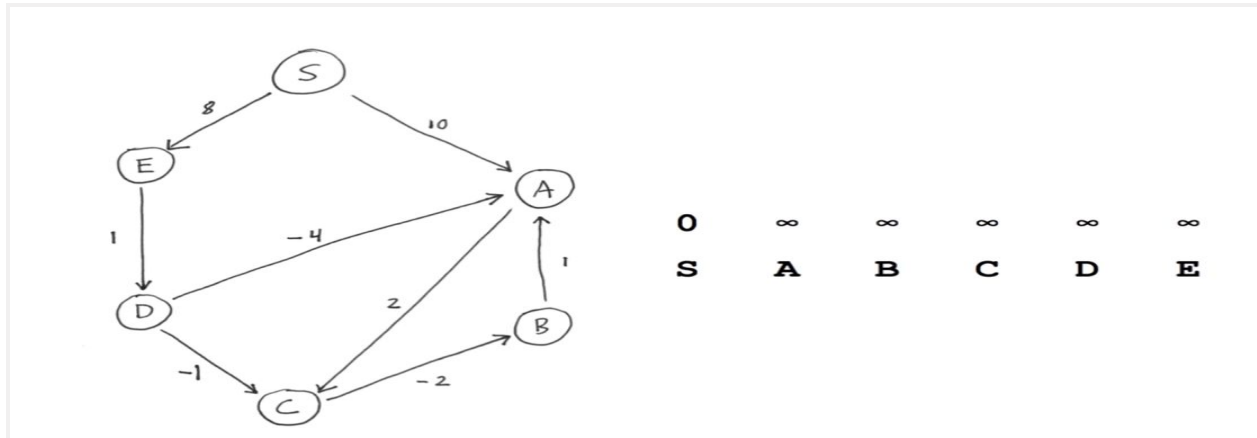
With **Binary Heap** -> $O(|V|)$ (Up to $|V|$ vertices may have to be stored)

Bellman-Ford

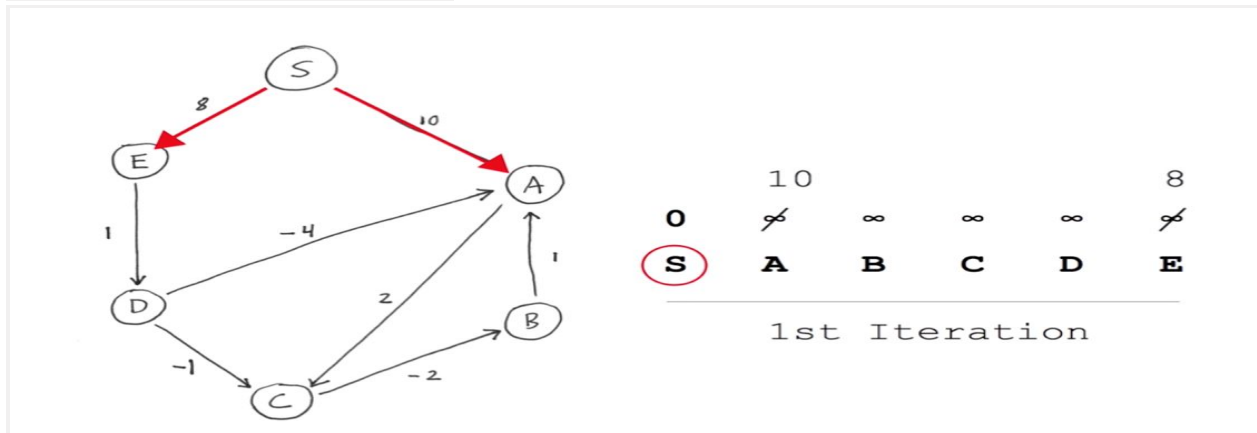
- 24.1 The Bellman-Ford algorithm - Page 651
- We will need to do Vertices - 1 iterations

STEPS:

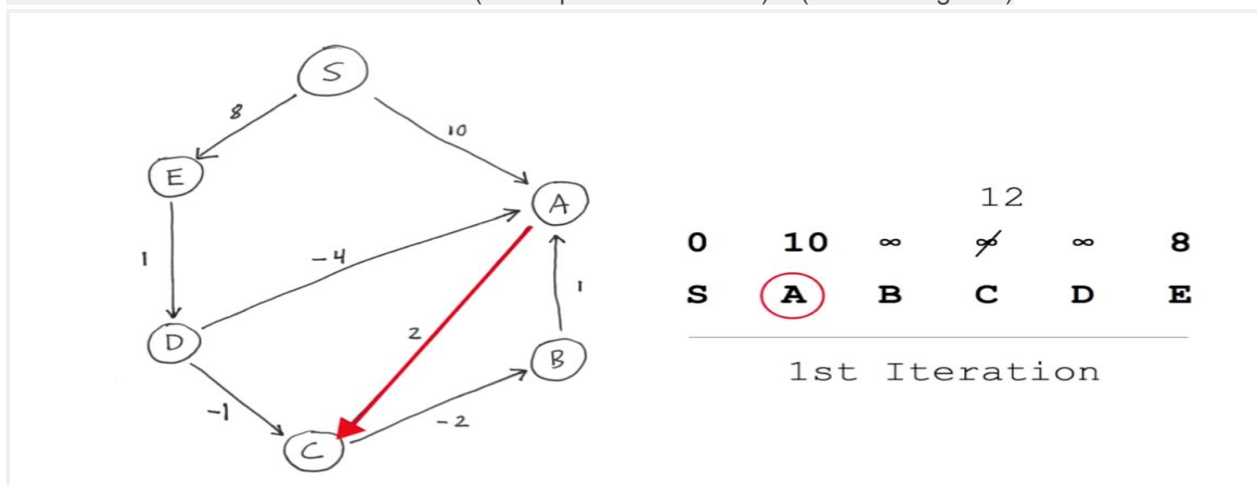
S is starting node, set all other distance values to INFINITY in list and starting node distance to 0



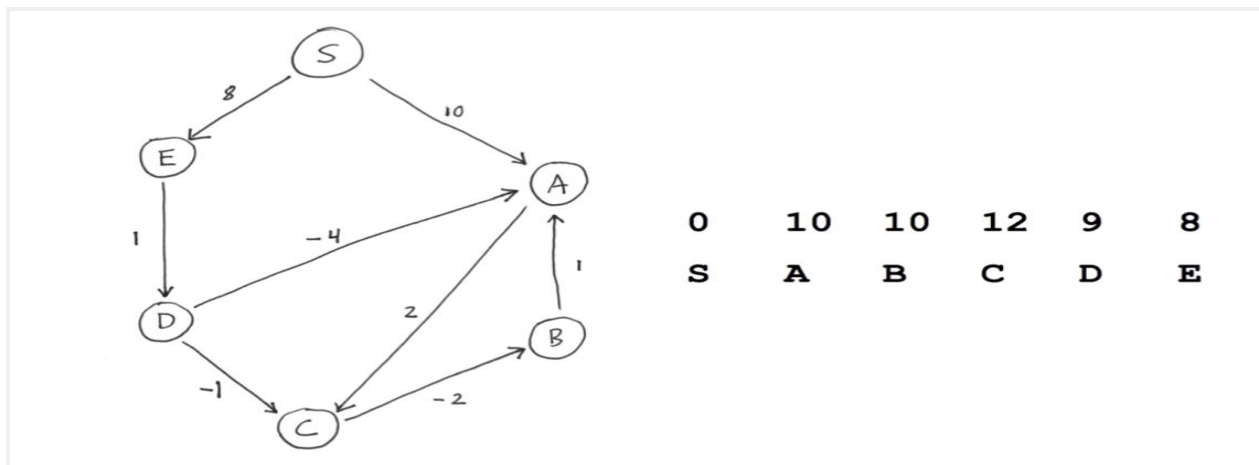
At each iteration, we must examine all of the edges. Look at outgoing edges of each node. Calculate the cost to reach each node



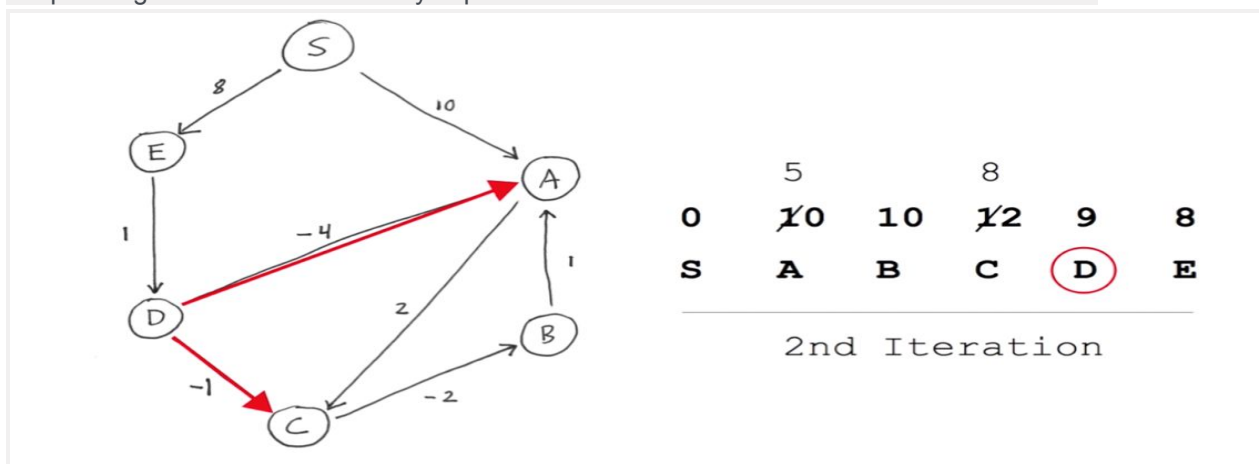
Calculate total cost to each node. TC = (Cost up to current node) + (Cost to neighbor)



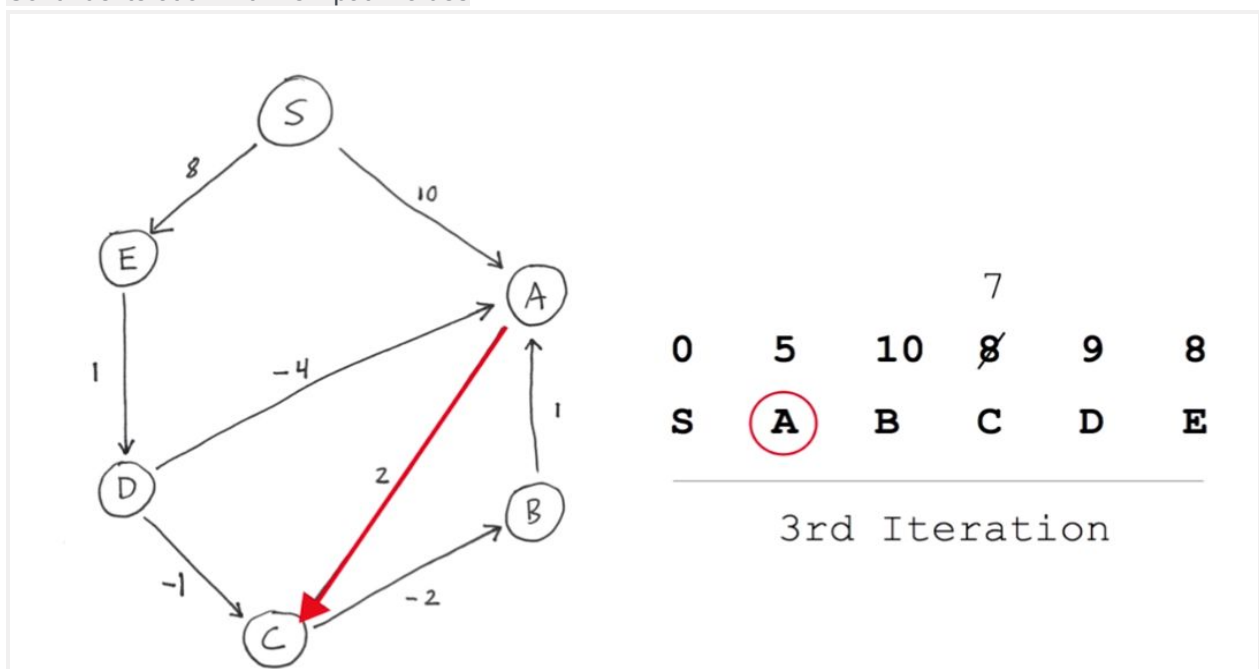
End of iteration 1... All total costs from S

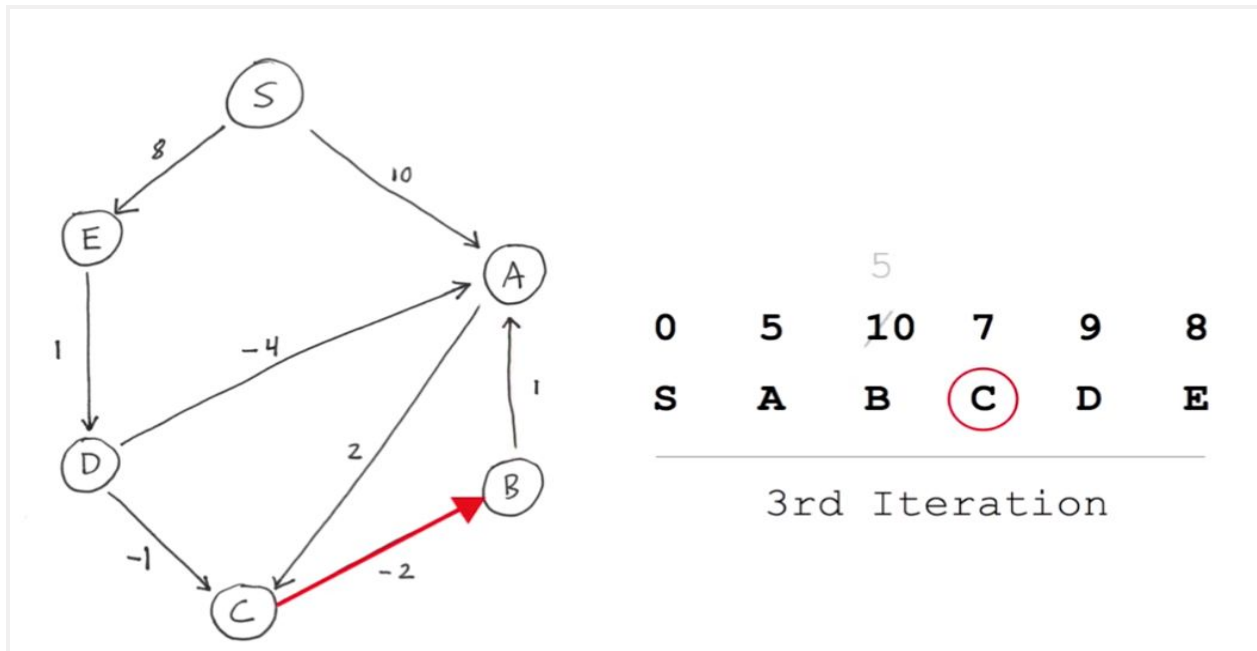


Loop through list until we find a way improve our total cost to a Node. D can shorten A and C

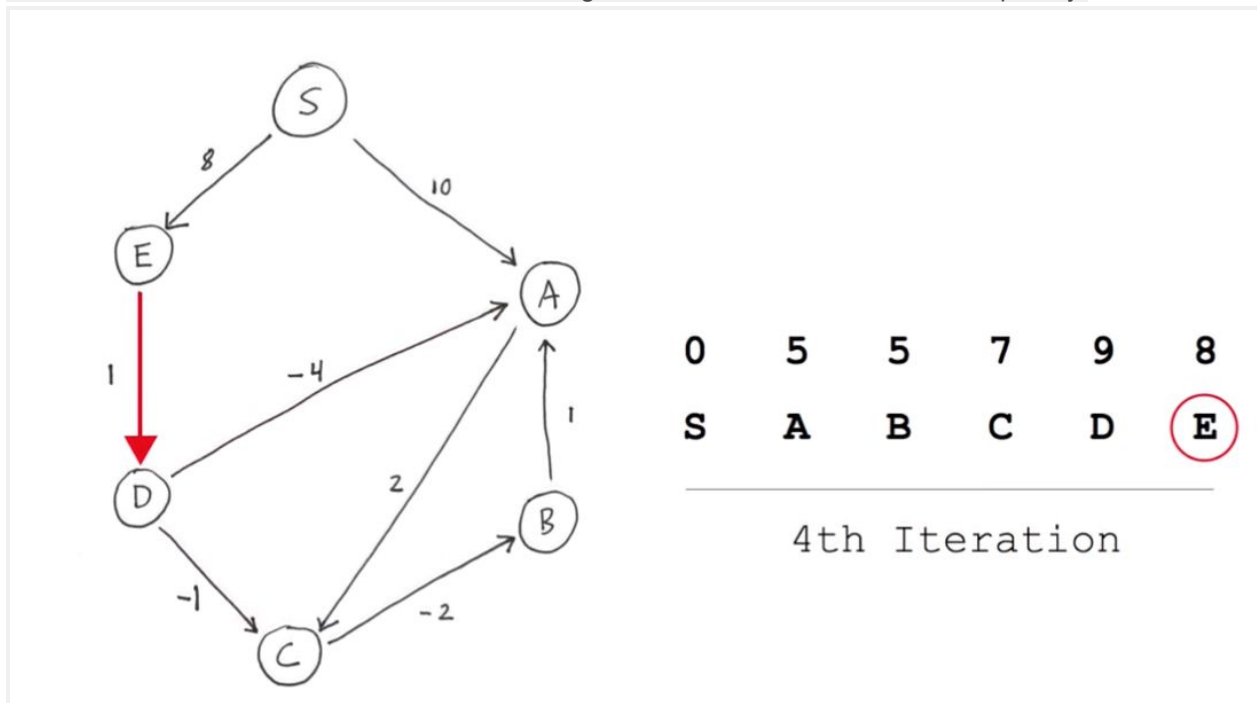


Continue iteration with new path values





Can take #Nodes - 1 iterations max. But no changes in iteration four thus we can stop early.



Time Complexity = $O(|V| * |E|)$

Knapsack Problem(Dynamic)

- The 0-1 knapsack problem is the following. A thief robbing a store finds n items. The i th item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack, for some integer W . Page 425
- 0 - 1 knapsack problem. 0-1 means you may either pick or not pick an item
 - We have a total weight.
 - We then have a list of items with a value and a weight.
 - We want to maximize the items value, while not crossing total weight

0/1 Knapsack Dynamic Programming
Total wt = 7

val	wt	0	1	2	3	4	5	6	7
(1)	1	0							
(4)	3	0							
(5)	4	0							
(7)	5	0							

items	
wt	val
1	1
3	4
4	5
5	7

- Number of columns = total weight + 1
 - Number of rows = # of items in increasing order of weight
 - ONLY have one of each item
 - Go through and fill out how many of each item we can have at varying total weight
- Total weight for item with val and weight 1

val	wt	0	1	2	3	4	5	6	7
(1)	1	0	1	1	1	1	1	1	1
(4)	3	0							
(5)	4	0							
(7)	5	0							

Go through and check if value at each spot is greater with past value or new value
Total value after introducing weight 3

val	wt	0	1	2	3	4	5	6	7
(1)	1	0	1	1	1	1	1	1	1
(4)	3	0	1	1	4	5	5	5	5
(5)	4	0							
	5	0							

Continue steps with next weights

val	wt	0	1	2	3	4	5	6	7
(1)	1	0	1	1	1	1	1	1	1
(4)	3	0	1	1	4	5	5	5	5
(5)	4	0	1	1	4	5	6	6	9
(7)	5	0							

val	wt	0	1	2	3	4	5	6	7
(1)	1	0	1	1	1	1	1	1	1
(4)	3	0	1	1	4	5	5	5	5
(5)	4	0	1	1	4	5	6	6	9
(7)	5	0	1	1	4	5	7	8	9

Max value with total weight of 7 is 9.

- Choose the items to give this max value
 - Start with answer value, and retrace its path.
 - Remove rows who do not affect the value.
 - Values(5, 4) -> weights(4, 3 = 7)

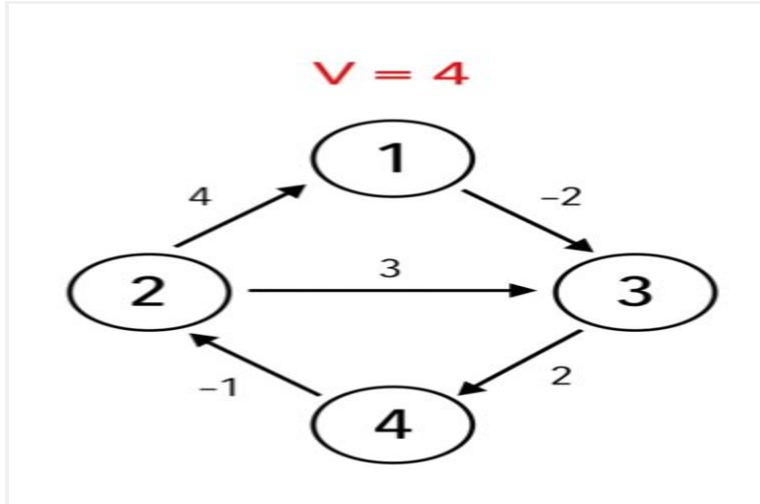
Formula

i is row, j is column

$$\begin{aligned}
 &\text{if } (j < wt[i]) \{ \\
 &\quad T[i][j] = T[i-1][j] \} \\
 &\text{else} \{ \\
 &\quad T[i][j] = \max \left\{ \begin{array}{l} val[i] + T[i-1][j - wt[i]] \\ T[i-1][j] \end{array} \right.
 \end{aligned}$$

Floyd-Warshalls

- 25.2 The Floyd-Warshall algorithm - 693
- Shortest path between all pairs for vertices, negative edges are allowed.



Create a distance array that keeps track of the shortest path between nodes

Initialized to infinity

Diagonal is 0 because each nodes path to itself is 0

	1	2	3	4
1				
2				
3				[3,4]
4				

dist

Loop through and add weights to table

	1	2	3	4
1	0		-2	
2	4	0	3	
3			0	2
4		-1		0

No update

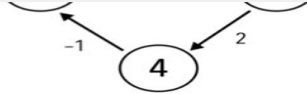
k = 1 2 3 4
i = 1 2 3 4
j = 1 2 3 4

dist [i][j] > dist [i][k] + dist [k][j]
dist [1][1] > dist [1][1] + dist [1][1]
0 > 0 + 0
X 0 > 0

Condition met, update path with the smaller weight

k = 1 2 3 4
i = 1 2 3 4
j = 1 2 3 4

dist [i][j] > dist [i][k] + dist [k][j]
dist [2][3] > dist [2][1] + dist [1][3]
3 > 4 + -2
3 > 2



	1	2	3	4
1	0		-2	
2	4	0	2	
3			0	2
4		-1		0

Continue until have looped through k completely and finished

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

Time Complexity -> $O(V^3)$

Data structures:

Hash Tables

- Hash Tables - Page 253
- Key value lookup. Given a key, associate a value with it for quick lookups.
- Key \Rightarrow value
- Map from key \rightarrow hash code \rightarrow index in array
- Collision, when two strings have the same hash codes.
 - Chaining.
 - When there is a collision, store them in a linked list
- Run Time $\rightarrow O(1)$ for well distributed hash table
 - $O(n)$ if lots of collisions

Binary Search Trees

- Binary Search Trees - Page 286
- Complexity**

Complexity of Binary Search Trees

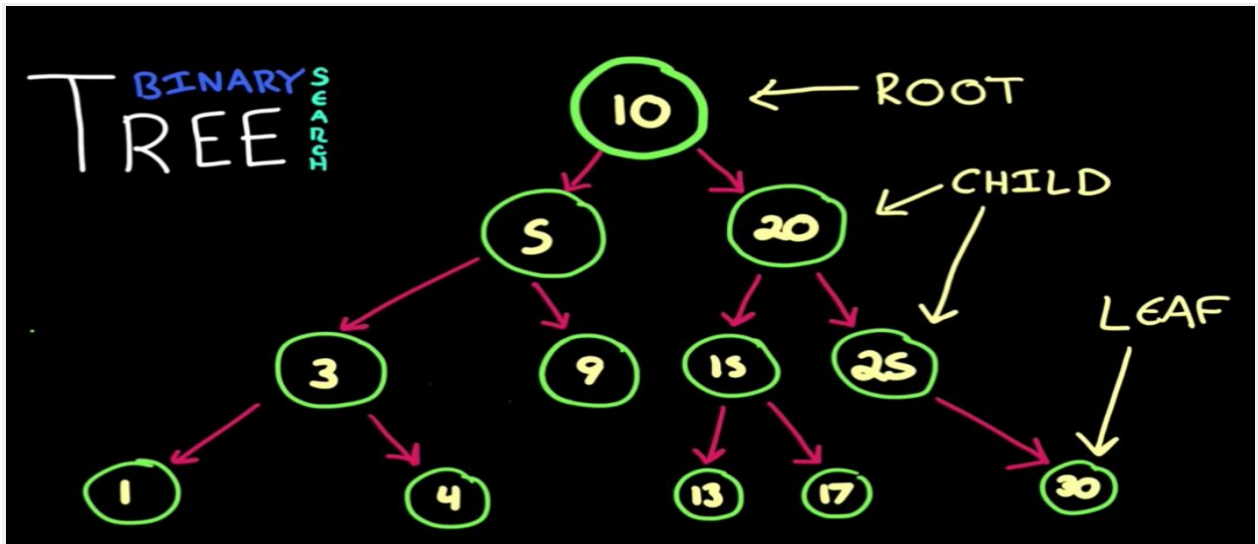
Operation	Average	Worst
Insert	$O(\log(n))$	$O(n)$
Delete	$O(\log(n))$	$O(n)$
Remove	$O(\log(n))$	$O(n)$
Search	$O(\log(n))$	$O(n)$

But When balanced!!!

Complexity of Balanced Binary Search Trees

Operation	Average	Worst
Insert	$O(\log(n))$	$O(\log(n))$
Delete	$O(\log(n))$	$O(\log(n))$
Remove	$O(\log(n))$	$O(\log(n))$
Search	$O(\log(n))$	$O(\log(n))$

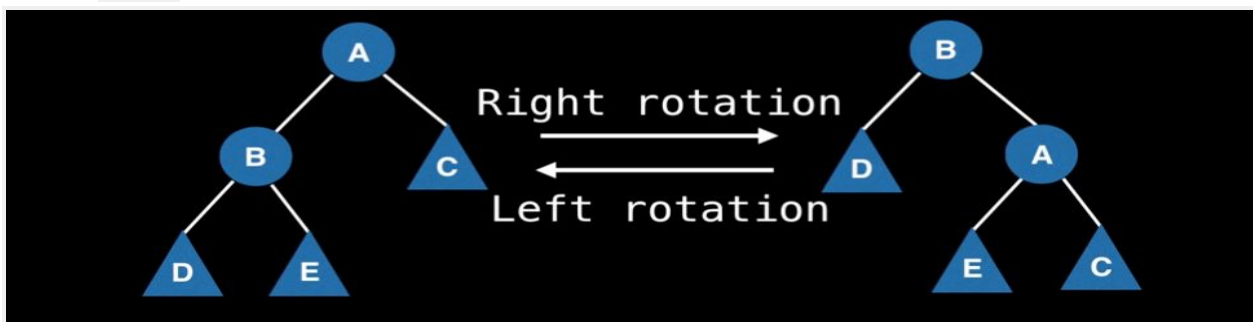
- Ordered tree, right children are greater than root, left are less. Same for all subtrees



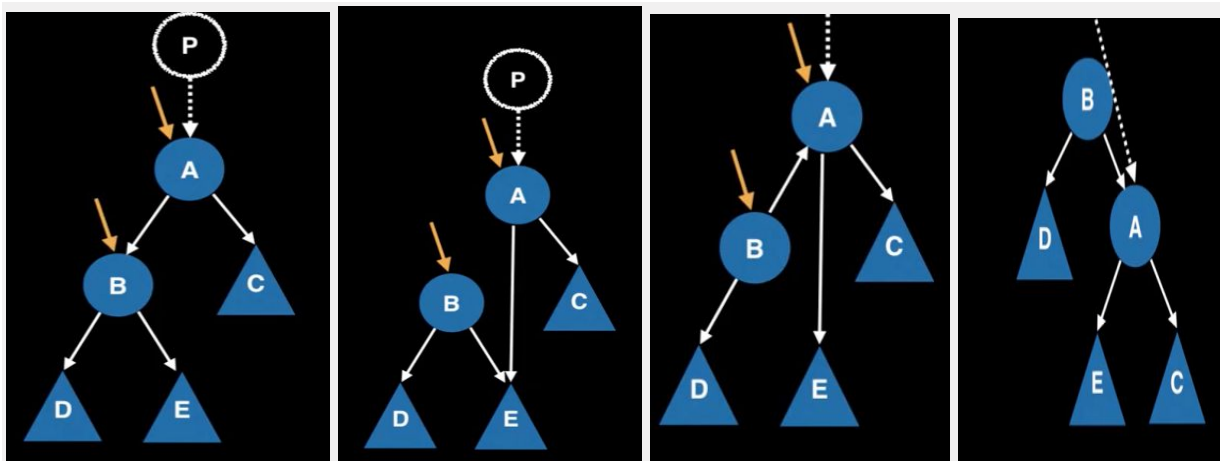
Insert

- Continue until find an open space
- But must check for unbalanced!!

Rotate

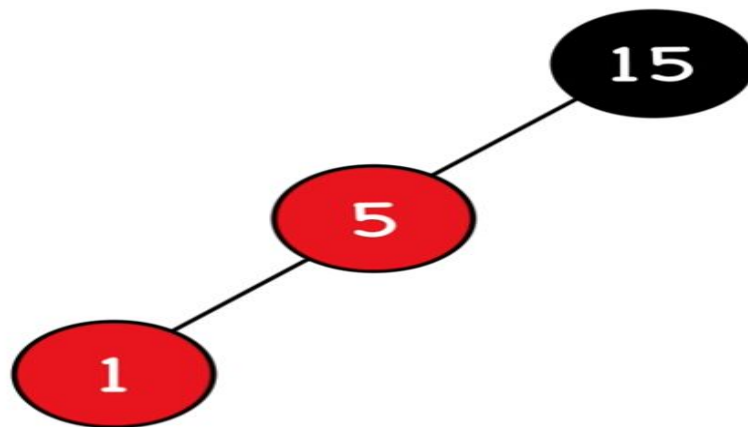


STEPS

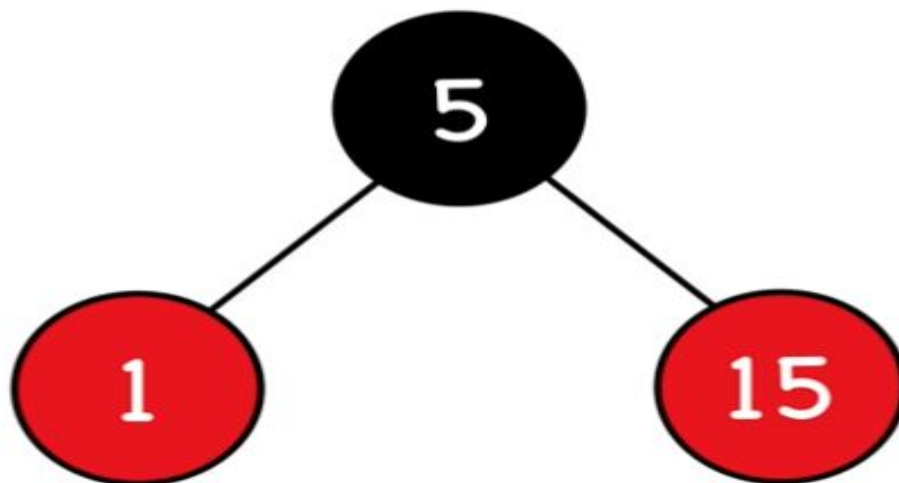


Red-Black Trees (Balanced Binary Search Trees)

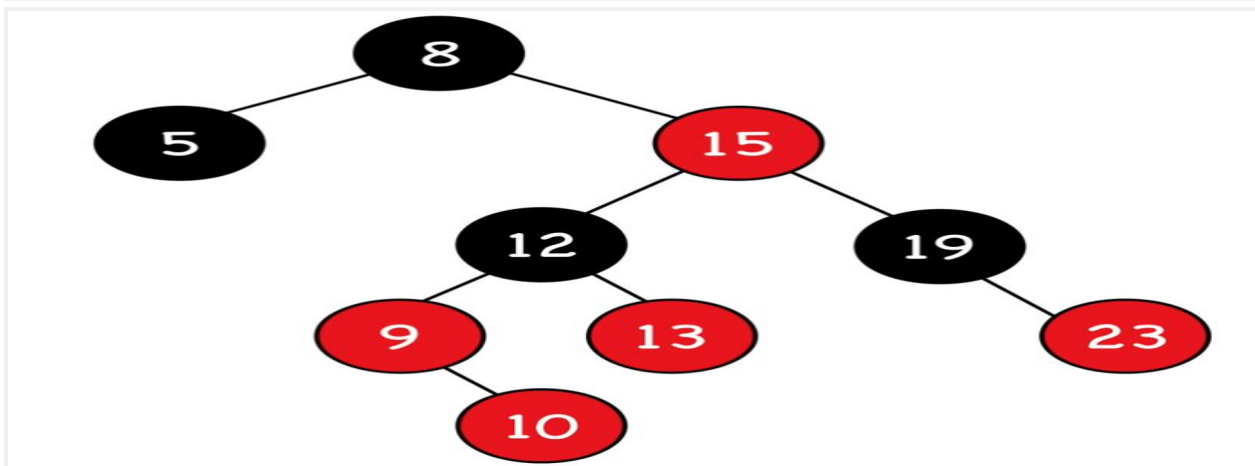
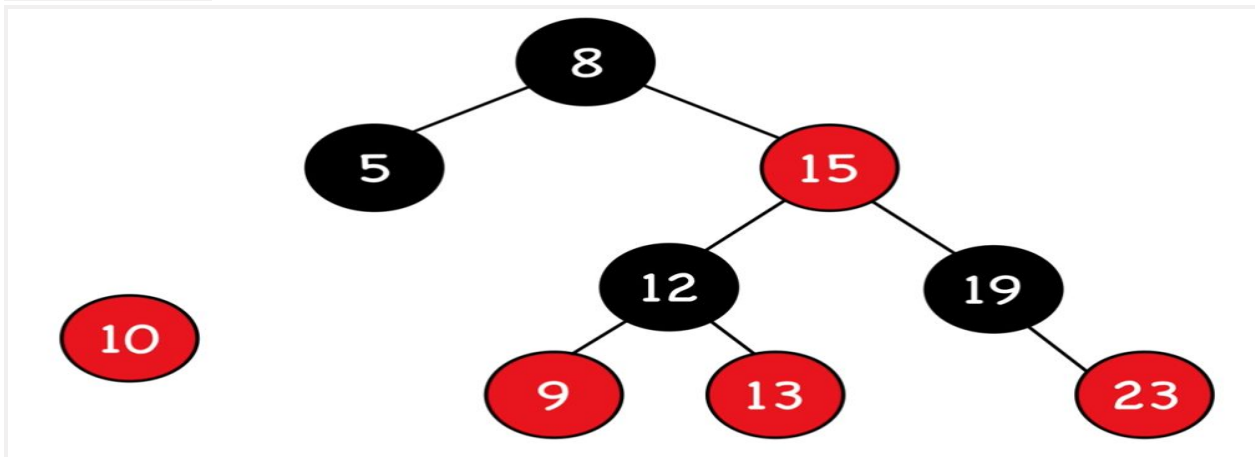
- Red-Black Trees - Page 308
 - A balanced search tree.
 - Guaranteed height of $O(\log(n))$
 - Space Complexity $\rightarrow O(n)$
 - RULES:
 - A node is either red or black
 - The root and leaves (NILL) nodes are black
 - If a node is red, then its children are black
 - All paths from a node to its NILL descendants contain the same number of black nodes.
 - Don't count starting node when counting black nodes in path
 - SEARCH, INSERT, REMOVE. All $O(\log(n))$
- INSERTION** \rightarrow done because self balancing
- Nodes are inserted as red
- INSERT 1, violation occurs



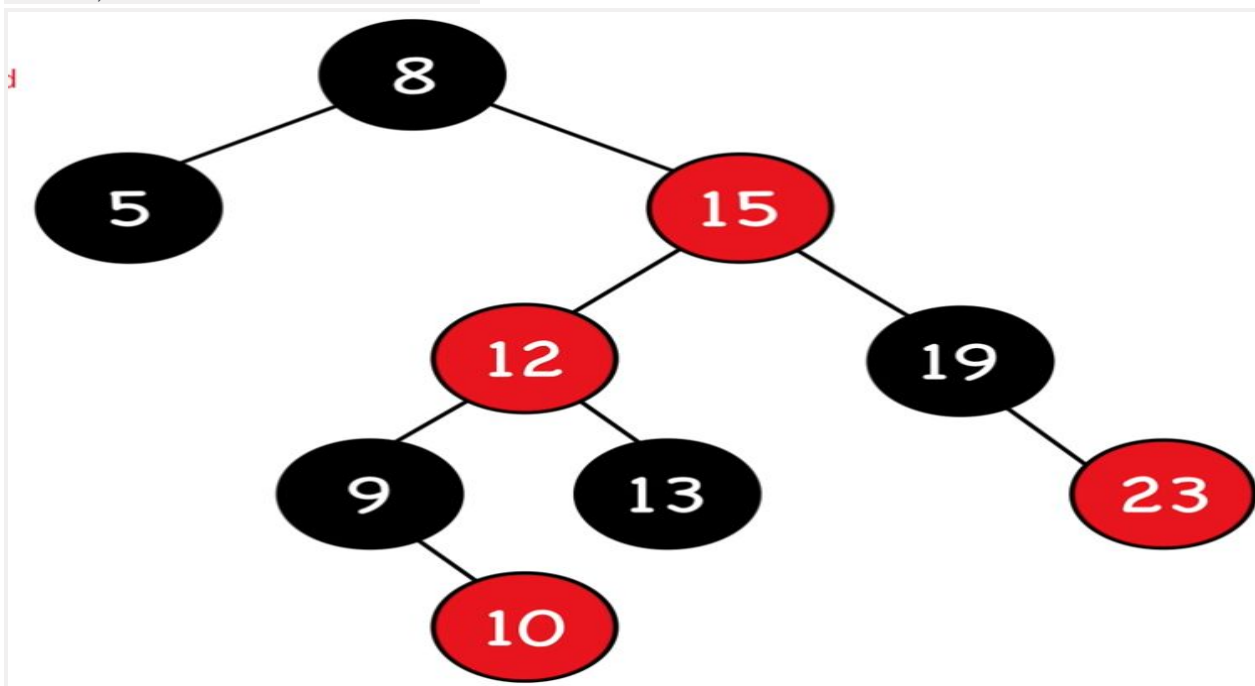
Right rotate



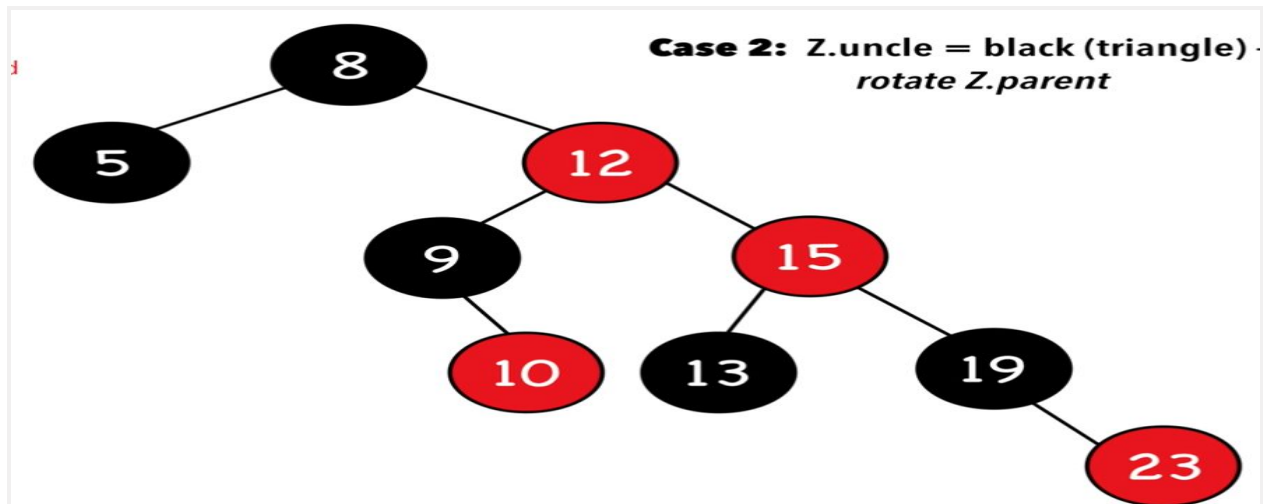
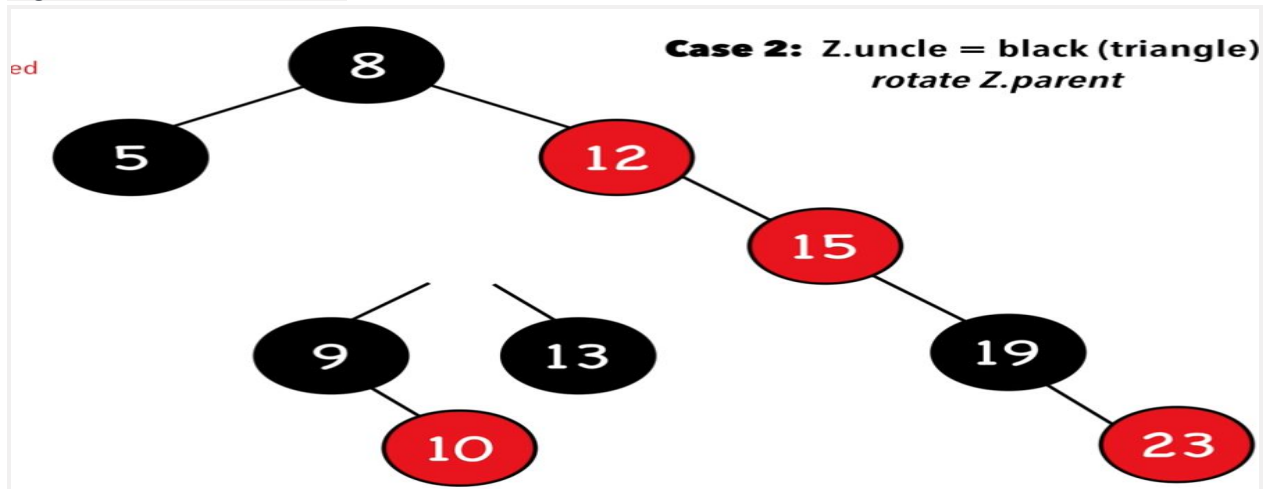
Insert 10 into tree



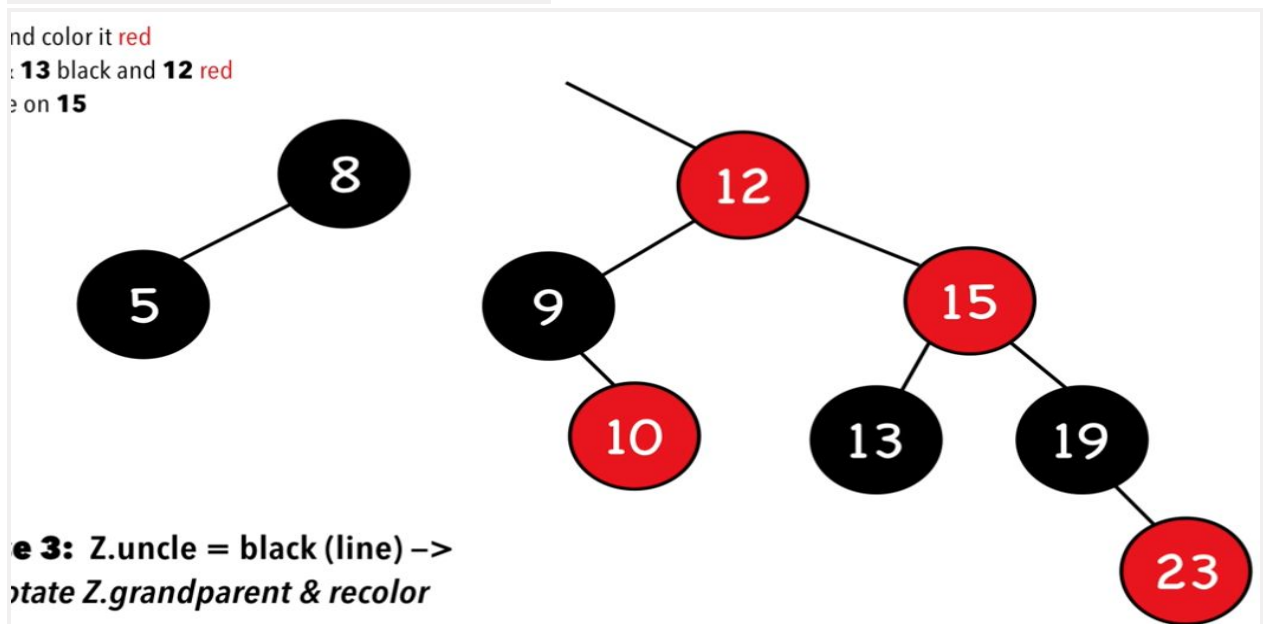
Recolor, but still violation with 12 and 15



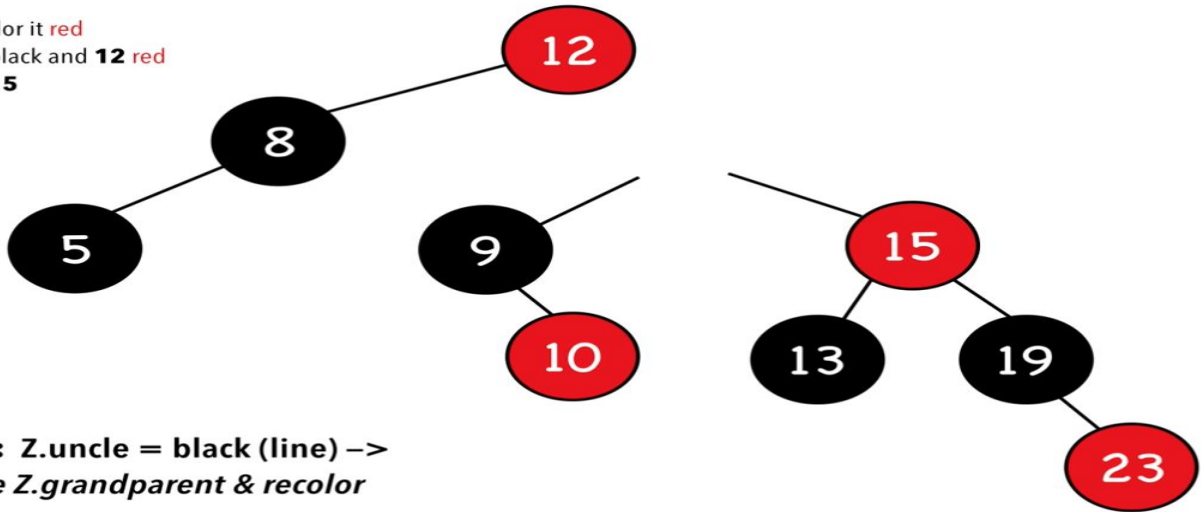
Right rotation with 12 and 15



Still violation with 12 and 15. Left rotation on 8

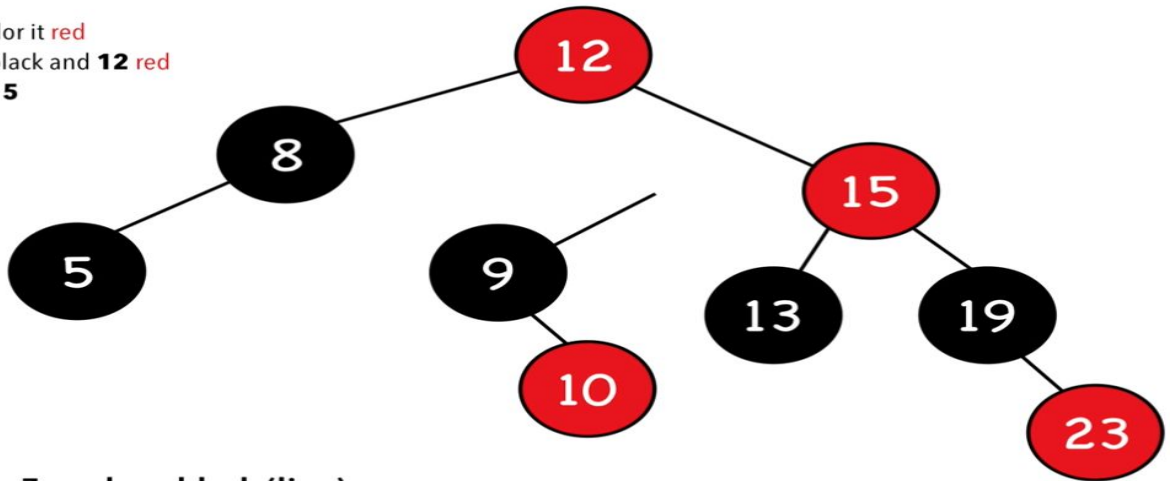


color it red
 13 black and 12 red
 on 15



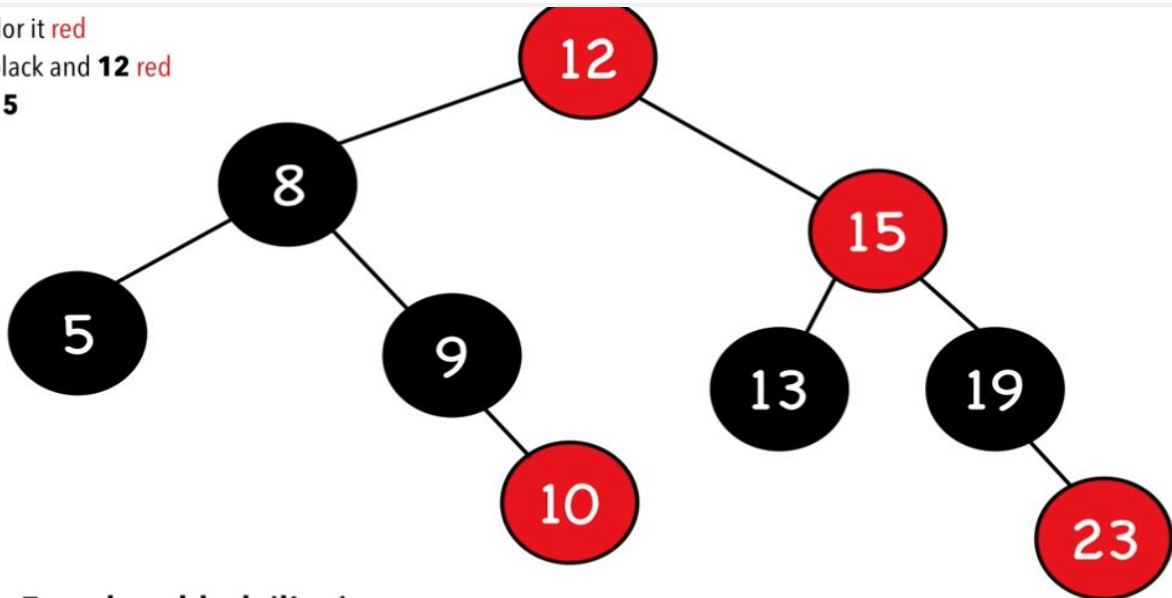
3: Z.uncle = black (line) ->
ate Z.grandparent & recolor

nd color it red
 & 13 black and 12 red
 e on 15



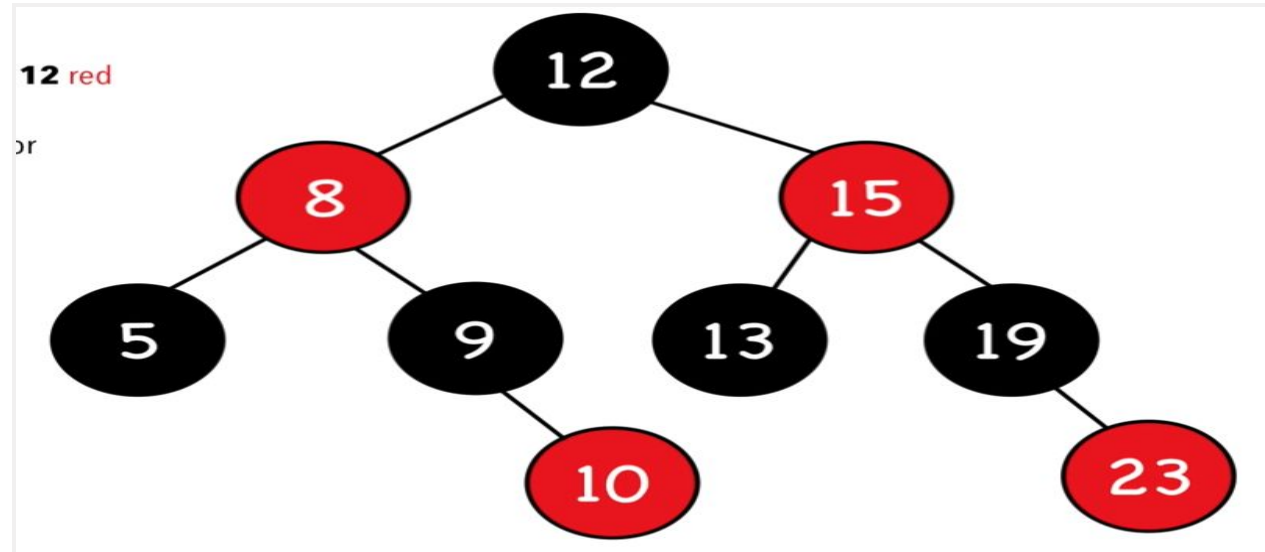
e 3: Z.uncle = black (line) ->

color it red
 black and 12 red
 15



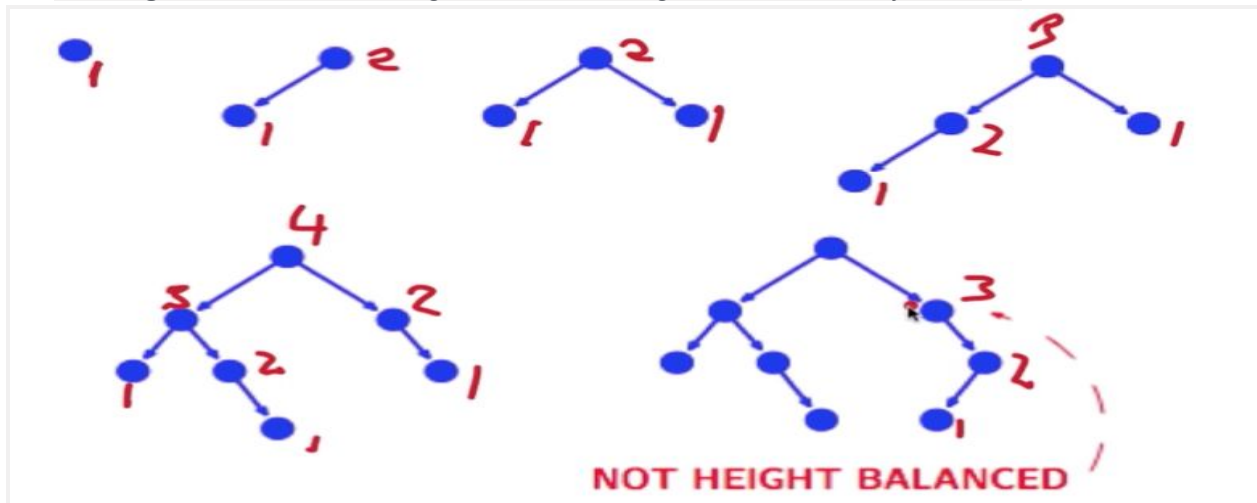
3: Z.uncle = black (line) ->

Now balanced, just need to recolor



AVL Trees

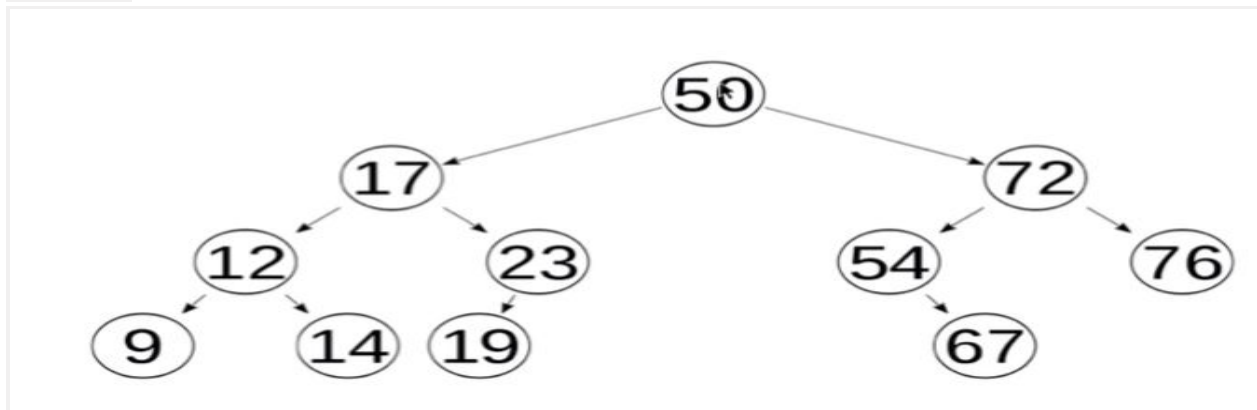
- 13-3 AVL trees - 333
- Height balanced** -> the height of the left and right subtrees differ by at most 1



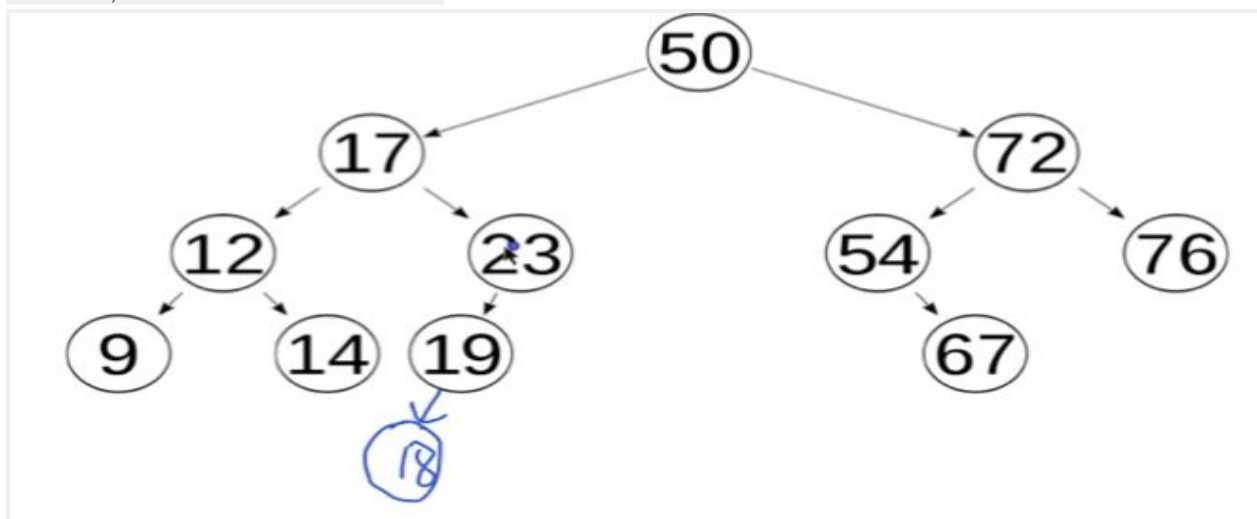
AVL is height balanced binary search tree

Complexity -> $O(\log(n))$

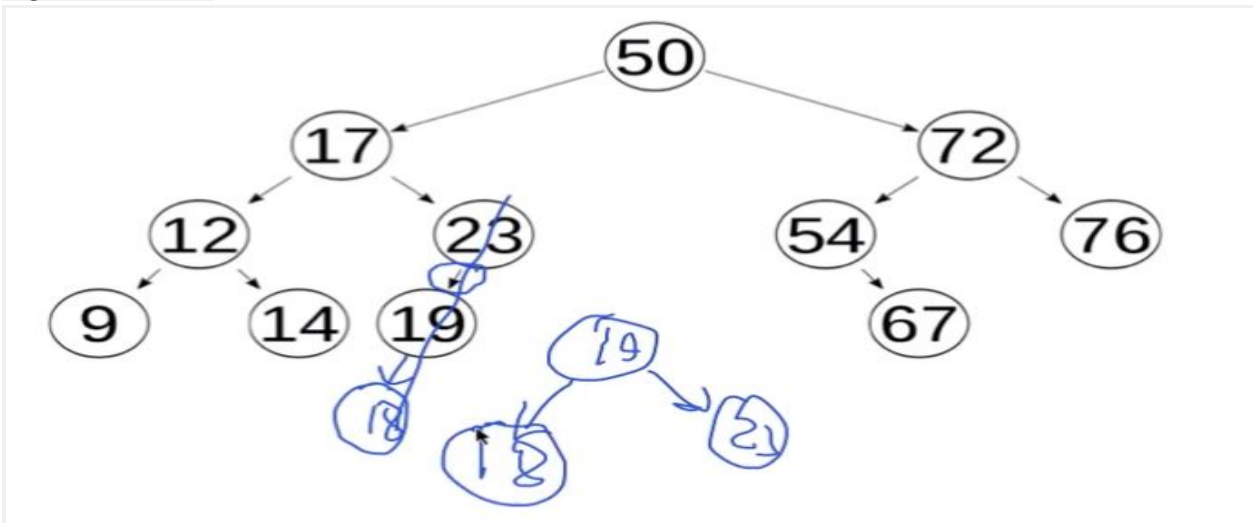
EXAMPLE



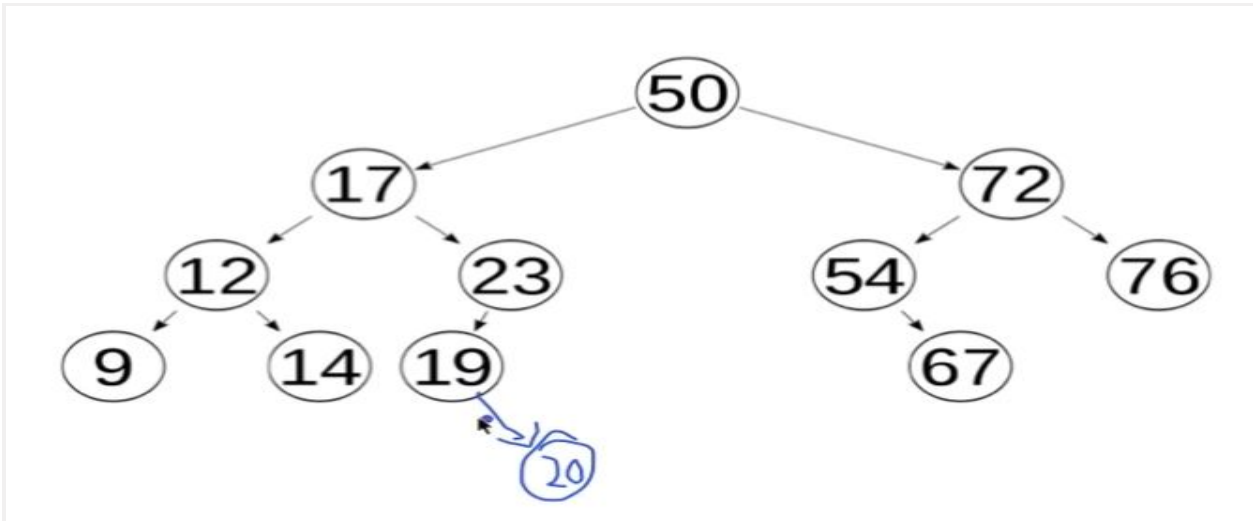
Insert 18, becomes unbalanced at 23



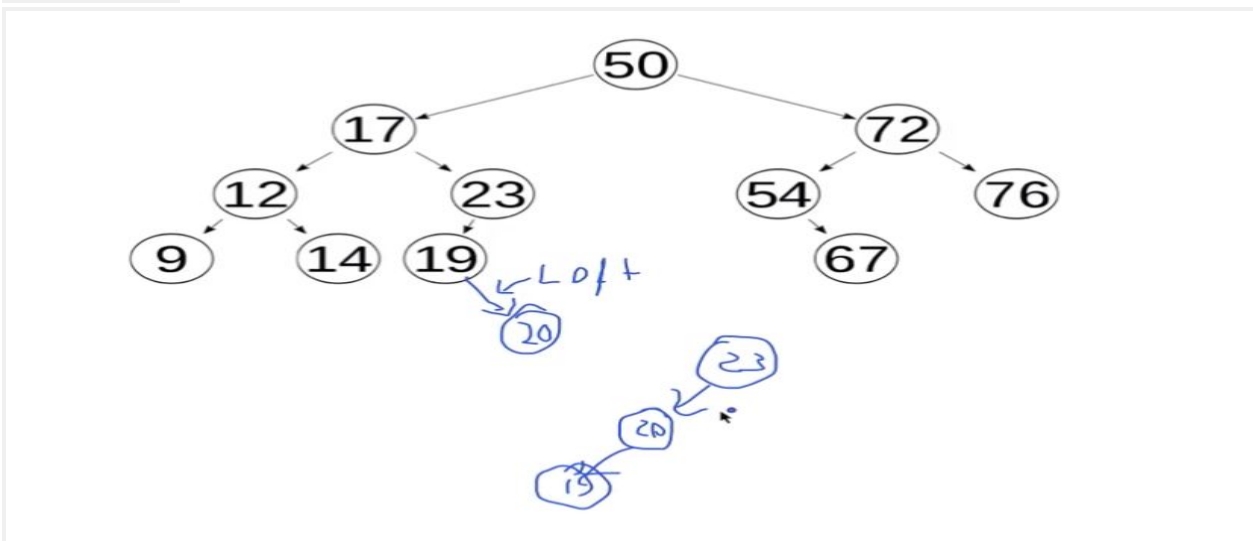
Right rotate on 23



New example. Insert 20, unbalanced at 23



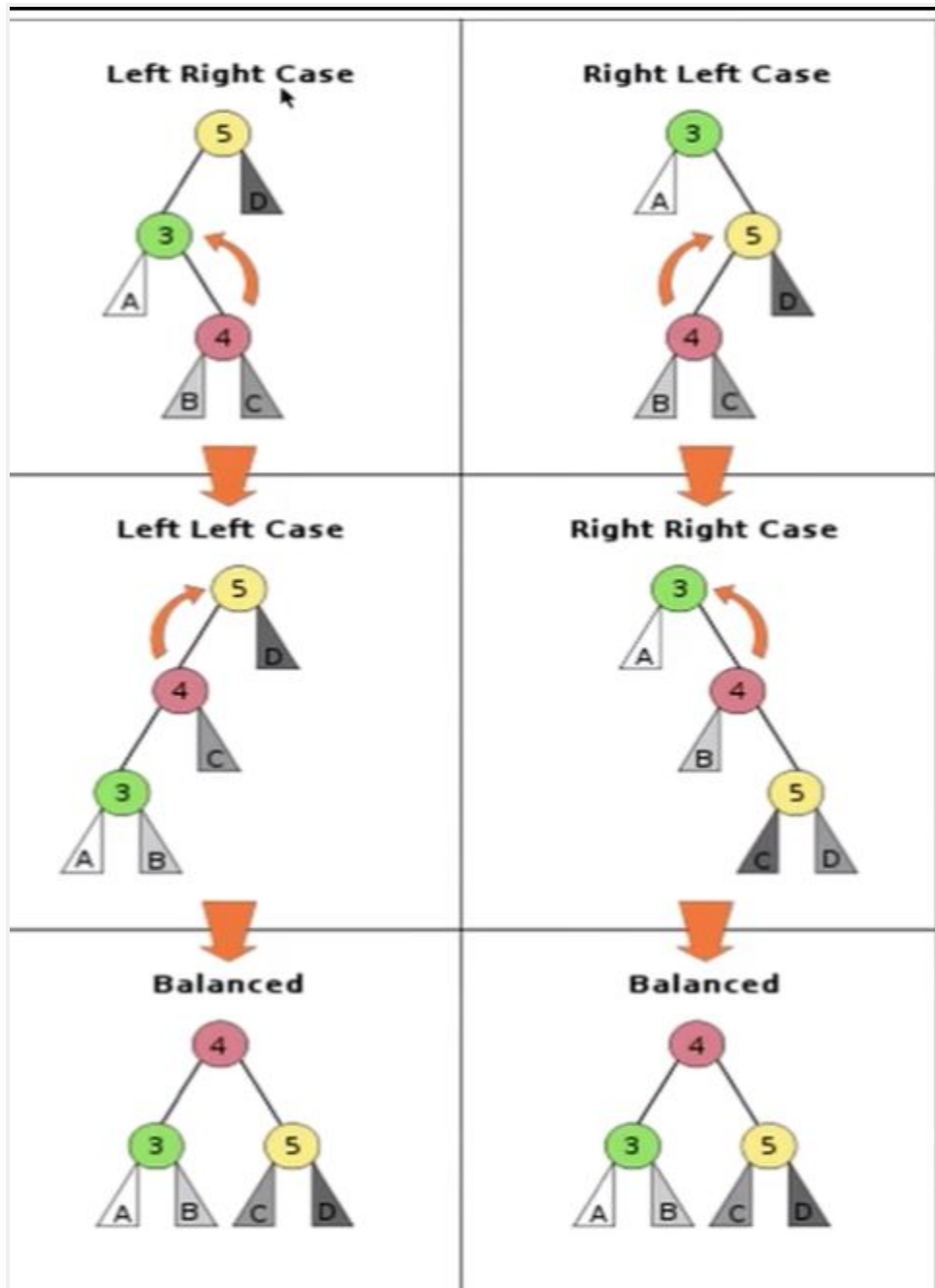
First left rotate.



Then right rotate on 20



***** Important *****



2-3 Trees

• Page 504

EXAMPLE

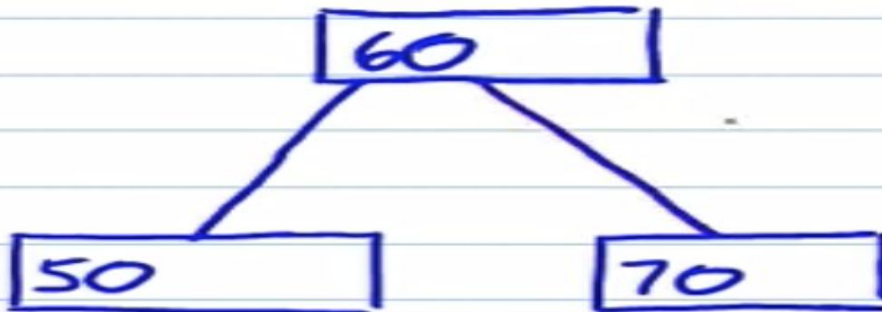
Insert 50, then 60

50 60 70 40 30 20 10 80 90 100

50 60 .

Insert 70

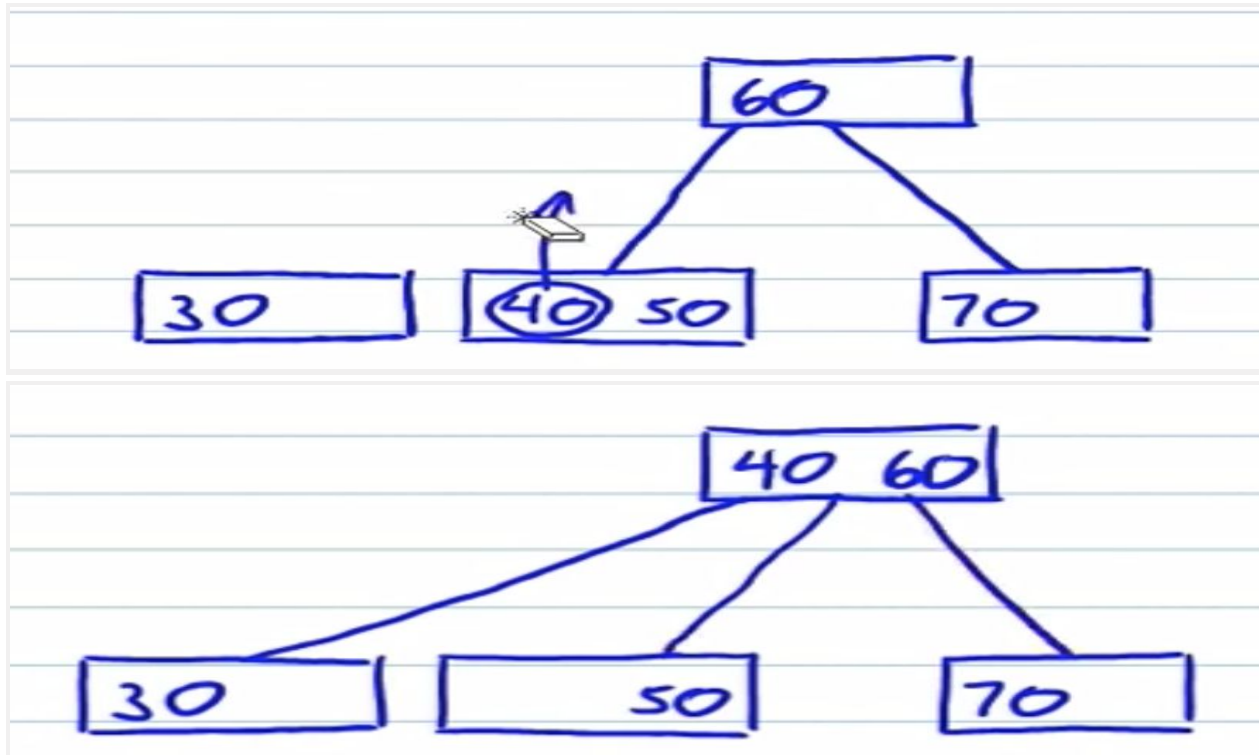
60



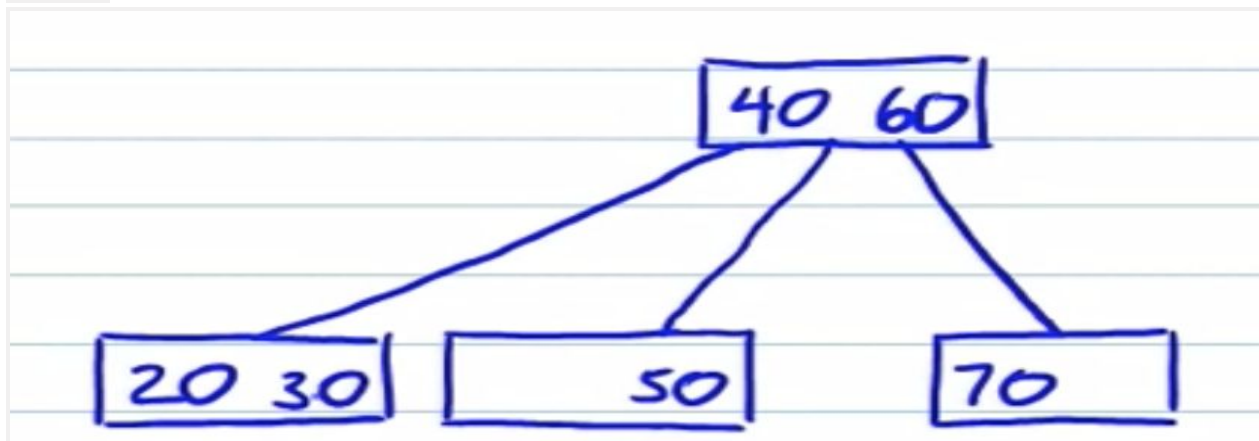
Insert 40



Insert 30



Insert 20



Insert 10. 2 Steps needed

