Part A

1	A stack data structure cannot be used for a) Implementation of Recursive Function b) **Allocation Resources and Scheduling** c) Reversing string d) Evaluation of string in postfix form
2	Postfix form of following expression. D + (E * F^G) a) DFG^E * + b) **DEFG^* +** c) GFED^ *+ d) GF^E* D+
3	If the elements “A”, “B”, “C” and “D” are placed in a queue and are deleted one at a time, in what order will they be removed? a) **ABCD** b) DCBA c) DCAB d) ABDC
4	A B-tree of order 4 and of height 3 will have a maximum of _____ keys. a) **255** b) 63 c) 127 d) 188
5	What is a hash table? a) A structure that maps values to keys b) **A structure that maps keys to values** c) A structure used for storage d) A structure used to implement stack and queue
6	If several elements are competing for the same bucket in the hash table, what is it called? a) Diffusion b) Replication c) **Collision** d) Duplication
7	Which of the following statements is/are TRUE for an undirected graph? P: The number of odd-degree vertices is even Q: Sum of degrees of all vertices is even a) P Only b) Q Only **c) Both P and Q** d) Neither P nor Q
8	Consider an undirected unweighted graph G. Let a breadth-first traversal of G be done starting from a node r. Let d(r, u) and d(r, v) be the lengths of the shortest paths from r

	to u and v respectively, in G. lf u is visited before v during the breadth-first traversal, which of the following statements is correct? a) d(r, u) < d (r, v) b) d(r, u) > d(r, v) **c) d(r, u) <= d (r, v)** d) None of the above

---

PART B

9. Some Practical Applications of Stack:

- They are the building blocks of function calls and recursive functions. Yes, this is a common application that you maybe aware of, but think about it — right now there are hundreds (if not thousands) of functions existing on your call stack in memory maintained by your OS. Every time a function is called, some memory is reserved (PUSH) for it on the call stack, and when it returns, the memory is deallocated (POP).

- The undo/redo function that has become a part of your muscle memory now, uses the stack pattern. All your actions are pushed onto a stack, and whenever you 'undo' something, the most recent action is popped off. The number of undos you can do is determined by the size of this stack.

- The stack pattern is also used to keep track of the 'most recently used' feature. I am sure you have come across the most recently seen files, items, tools etc.. across different applications. Your browser uses the stack pattern to maintain the recently closed tabs, and reopen them.

- Your code editor uses a stack to check if you have closed all your parentheses properly, and even to prettify your code.
  This ties to the more formally described use case of stacks — expression evaluation and syntax parsing. They are used to convert one notation of expression into another (like infix to postfix etc..), this is actually used in

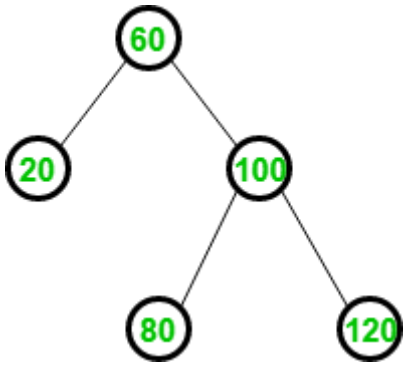
calculators. If you have prepared for coding interviews, you know the famous parentheses matching problem is solved using stack.

- They are used to implement back tracking algorithm, which is basically an algorithm with a goal, and if it takes a wrong path it simply ‘back tracks’ back to a previous state. These states are maintained using stacks. Simple games like tic-tac-toe can be solved using this approach.
  A similar concept is used in the Depth First Search algorithm.

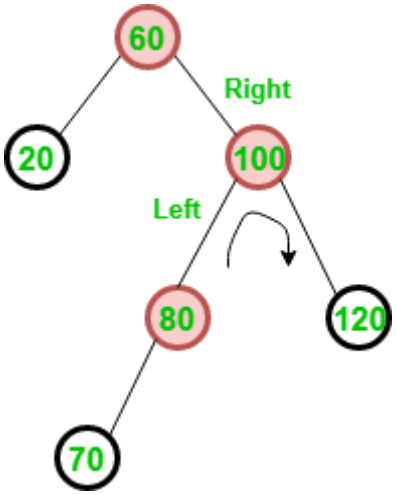
- Increase efficiency of algorithms — Several algorithms make use of the stack data structure and its properties. Examples are Graham Scan — which is used to find the convex hull, and the problem of finding the nearest smaller or larger value in an array.

---

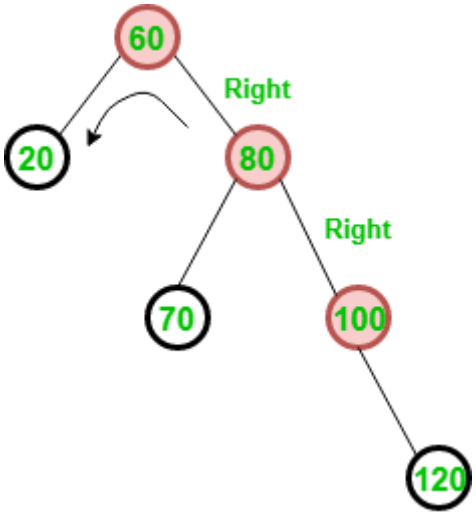
10. Consider the following AVL tree.



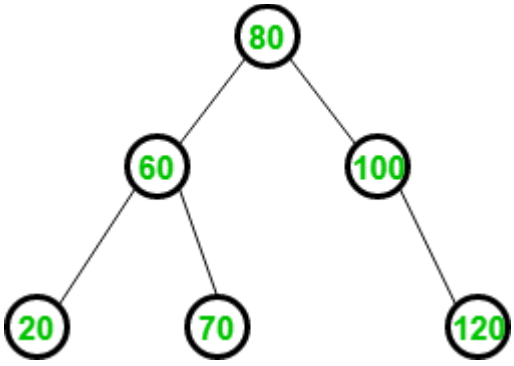
Solution: The element is first inserted in the same way as BST. Therefore after insertion of 70, BST can be shown as:



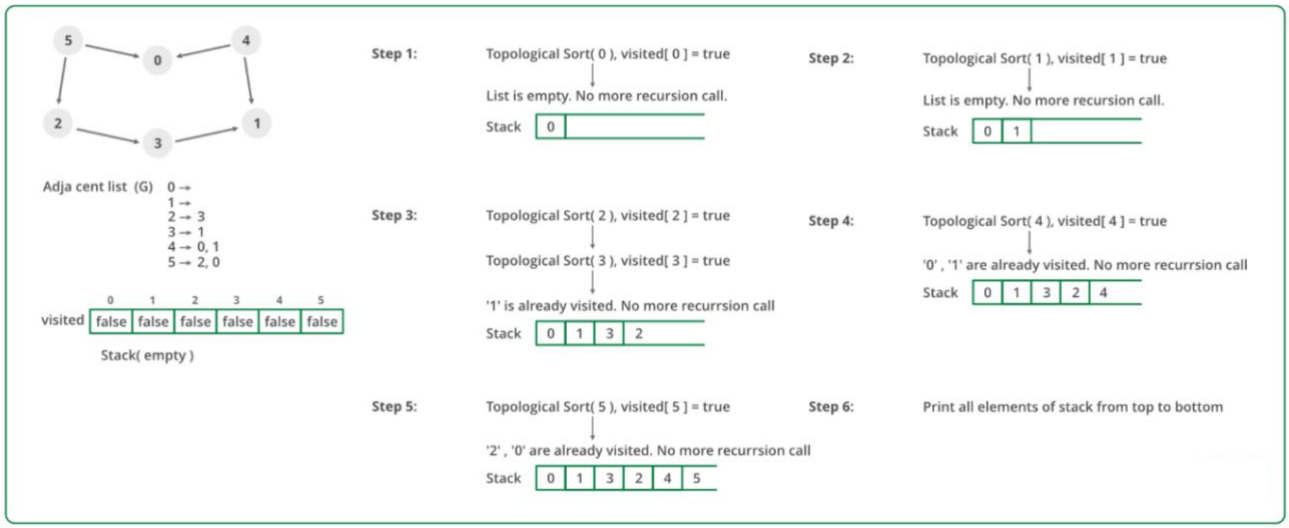
However, balance factor is disturbed requiring RL rotation. To remove RL rotation, it is first converted into RR rotation as:



After removal of RR rotation,



11. Topological sorting



PART C

12.A. To convert the given expression into its corresponding postfix expression, we can use the concept of the stack. Here is the pseudocode for converting the expression into its postfix form:

```
Function IsOperator(character)
    If character is '+' or '-' or '*' or '/' Then
        Return True
    Else
        Return False

Function Precedence(operator)
    If operator is '*' or '/' Then
        Return 2
    Else If operator is '+' or '-' Then
        Return 1
    Else
        Return 0

Function ConvertToPostfix(expression)
    Initialize an empty stack
    Initialize an empty postfix string
    For each character in the expression
        If the character is a number
            Append it to the postfix string
        Else If the character is '('
            Push it to the stack
```

```
        Else If the character is ')'
            Pop and append all the operators from the stack to the postfix string until '(' is
encountered
            Pop '(' from the stack
        Else If the character is an operator
            While the stack is not empty and the precedence of the top of the stack is
greater than or equal to the precedence of the current operator
                Pop and append the top of the stack to the postfix string
            Push the current operator to the stack
    While the stack is not empty
        Pop and append the top of the stack to the postfix string
    Return the postfix string
```

Using the above pseudocode, let's convert the given expression "10 + ((7 – 5) + 10)/2" into its corresponding postfix expression.

1. Convert the expression:

Input: 10 + ((7 – 5) + 10)/2

Step 1: Apply the pseudocode to convert the infix expression to postfix:
10 7 5 - 10 + 2 / +

Resulting postfix expression: "10 7 5 - 10 + 2 / +"

2. Evaluate the postfix expression:

```
Initialize an empty stack
For each token in the postfix expression
    If the token is a number
        Push it to the stack
    Else If the token is an operator
        Pop the top two elements from the stack and perform the operation
        Push the result back onto the stack
```

Final result: 15.0

So, the postfix expression "10 7 5 - 10 + 2 / +" evaluates to 15.0.

12.B. Here is the pseudocode for inserting an element into a linked queue:

```
Structure Node
    value
    next

Function Enqueue(Queue, value)
    newNode = new Node
    newNode.value = value
    newNode.next = null

    If Queue is empty
        Queue.front = newNode
        Queue.rear = newNode
    Else
        Queue.rear.next = newNode
        Queue.rear = newNode
```

And here is the pseudocode for deleting an element from a linked queue:

```
Function Dequeue(Queue)
    If Queue is empty
        Return "Underflow"

    If Queue.front = Queue.rear
        value = Queue.front.value
        Queue.front = null
        Queue.rear = null
    Else
        value = Queue.front.value
        Queue.front = Queue.front.next

    Return value
```

Now, let's illustrate the process with an example:
Create an empty linked queue Queue

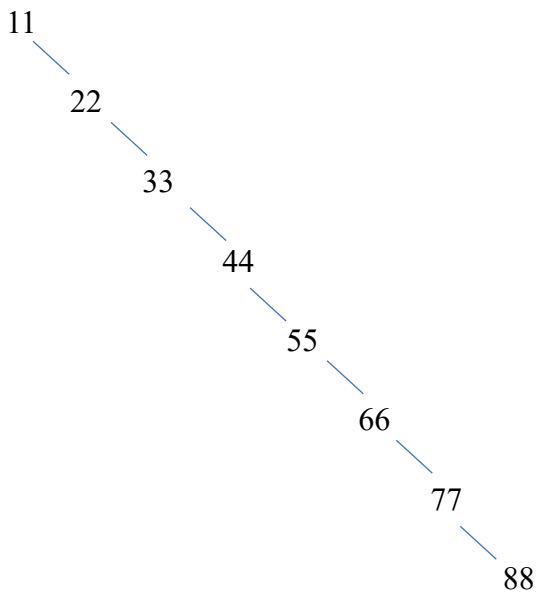
```
Enqueue(Queue, 5)
Enqueue(Queue, 10)
Enqueue(Queue, 15)

Dequeue(Queue) // This should return 5
Dequeue(Queue) // This should return 10
Dequeue(Queue) // This should return 15
Dequeue(Queue) // This should return "Underflow" as the queue is empty
```

Using the pseudocode, the elements 5, 10, and 15 are inserted into the linked queue. Then, the elements are dequeued one by one, resulting in 5, 10, and 15 being returned successively. Finally, attempting to dequeue an element from an empty queue results in "Underflow" being returned.

13.A. To construct a binary search tree (BST) with the given data {11, 22, 33, 44, 55, 66, 77, 88}, we can follow these steps: (5 marks)

1. Start by creating a root node with the first element, which is 11.
2. Then, for each subsequent element, check whether it is greater or less than the current node's value to determine where it should be placed in the tree.
3. Repeat this process until all the elements are inserted.

The resulting binary search tree will look like the following:



b. The type of the binary tree is a Right skewed Binary Search Tree. (2 marks)

c. The height of the BST is 7. This is the number of edges on the longest downward path from the root node to a leaf node. (2 marks)

13.B. Primary clustering occurs when multiple keys hash to the same index, causing longer chains at specific positions in the hash table. One common method to avoid primary clustering is to utilize a technique called open addressing, which includes linear probing, quadratic probing, or double hashing. Let's consider an example using linear probing:

Assuming a hash table of size 11, we can illustrate how primary clustering can be avoided through linear probing. Suppose we have the following keys to insert: 12, 22, 32, 42, 52, 62, 72, 82, 92, 102.

We will use the hash function `h(key) = key % 11` for simplicity.

1. Initially, the hash table is empty.
2. Insert the keys one by one:

Key: 12
Index: 12 % 11 = 1 (Hashed to index 1)

Key: 22
Index: 22 % 11 = 0 (Hashed to index 0)

Key: 32
Index: 32 % 11 = 10 (Hashed to index 10)

Key: 42
Index: 42 % 11 = 9 (Hashed to index 9)

Key: 52
Index: 52 % 11 = 8 (Hashed to index 8)

Key: 62
Index: 62 % 11 = 7 (Hashed to index 7)

Key: 72
Index: 72 % 11 = 6 (Hashed to index 6)

Key: 82
Index: 82 % 11 = 4 (Hashed to index 4)

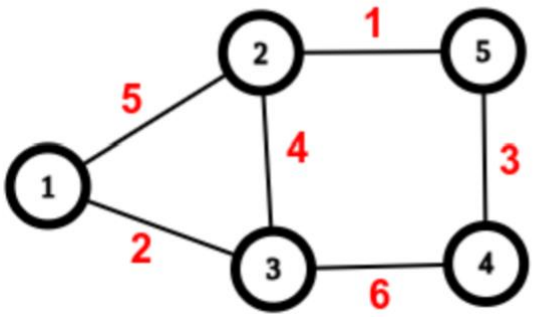
Key: 92
Index: 92 % 11 = 3 (Hashed to index 3)

Key: 102
Index: 102 % 11 = 10 (Collision, will be placed in next available slot, 10 + 1 % 11 = 0)
After inserting the keys, the hash table will look like this:

```
-------------------------------------------------
Index: 0  1  2  3  4  5  6  7  8  9  10
Key:   22 12     92 82 52 72 62 32 42 102
-------------------------------------------------
```

In this example, we can see that with the use of linear probing, even though there was a collision at index 10, the key 102 was placed at the next available slot, which is index 0. This effectively avoids primary clustering and spreads out the keys more evenly within the hash table.

14.a Draw the spanning tees of the given Graph G. (4 marks)

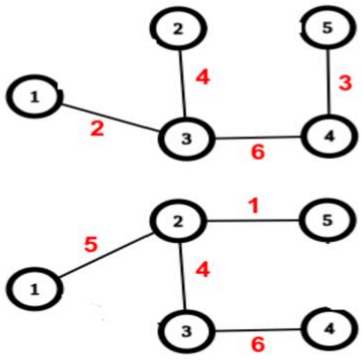


a. MST using Prim's algorithm

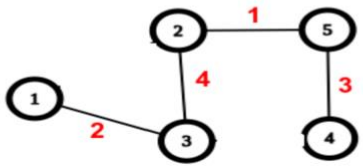
Solution:

a.

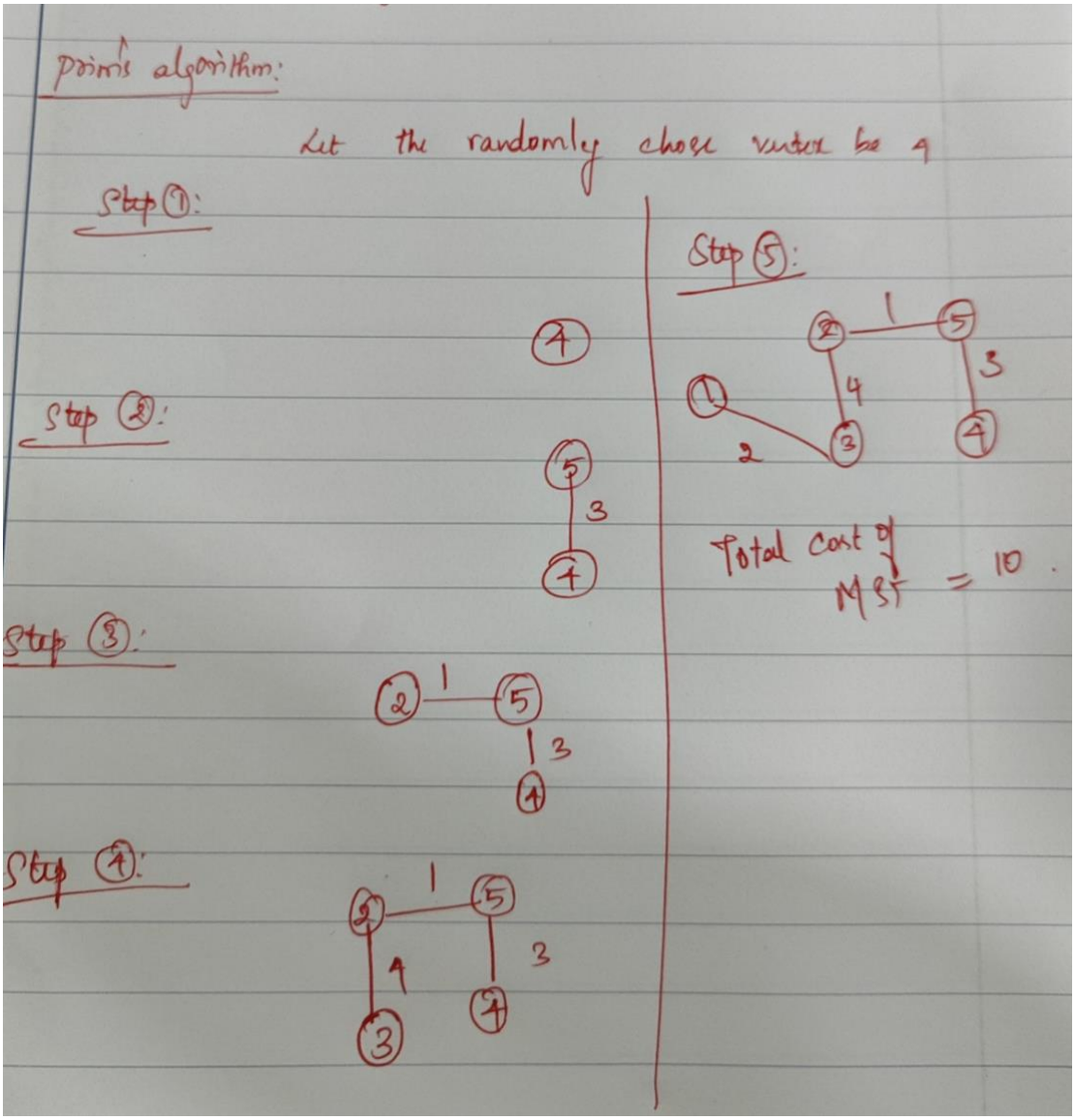
Some possible spanning trees are:



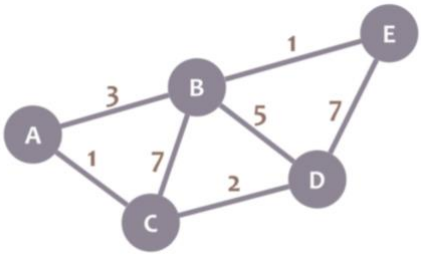
While the weights of the above spanning trees are 15 and 16 respectively, the minimum-weight spanning tree (minimum spanning tree) is 10.



b.



14.B. Construct a single source shortest path algorithm to find the shortest distance from the

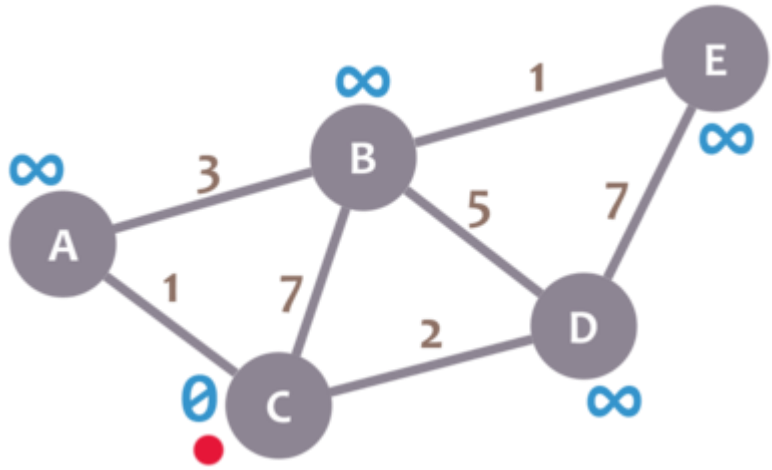


source node 'C' in the following graph.

Algorithm – 6 marks

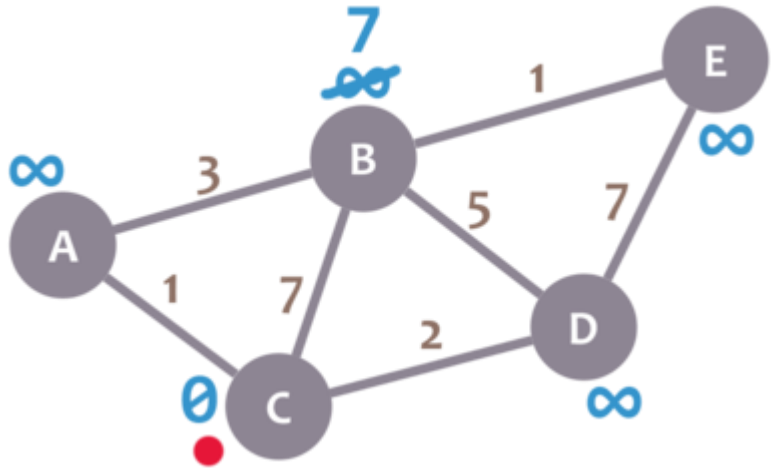
Distance computation – 3 marks.

Solution: During the algorithm execution, we'll mark every node with its *minimum distance* to node C (our selected node). For node C, this distance is 0. For the rest of nodes, as we still don't know that minimum distance, it starts being infinity (∞):

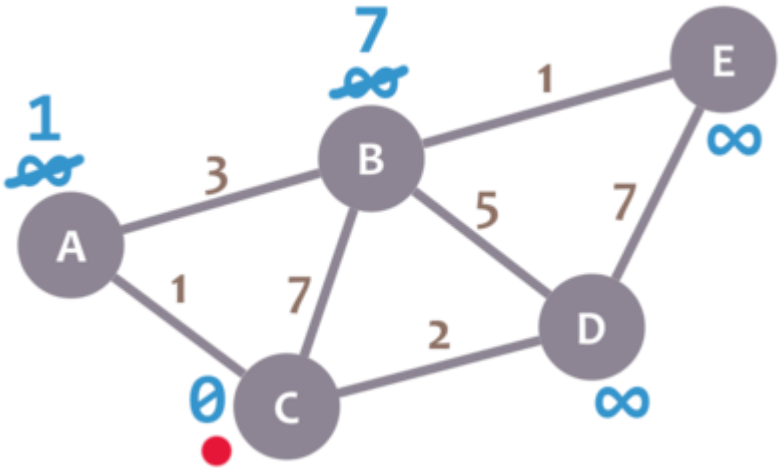


We'll also have a *current node*. Initially, we set it to C (our selected node). In the image, we mark the current node with a red dot.

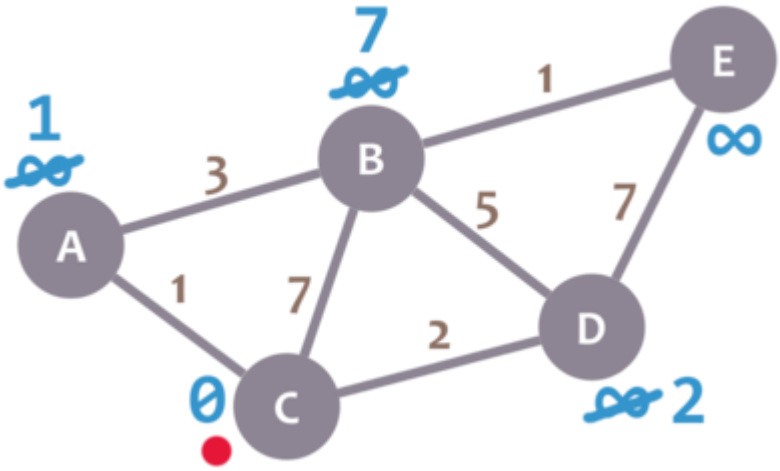
Now, we check the neighbours of our current node (A, B and D) in no specific order. Let's begin with B. We add the minimum distance of the current node (in this case, 0) with the weight of the edge that connects our current node with B (in this case, 7), and we obtain 0 + 7 = 7. We compare that value with the minimum distance of B (infinity); the lowest value is the one that remains as the minimum distance of B (in this case, 7 is less than infinity):



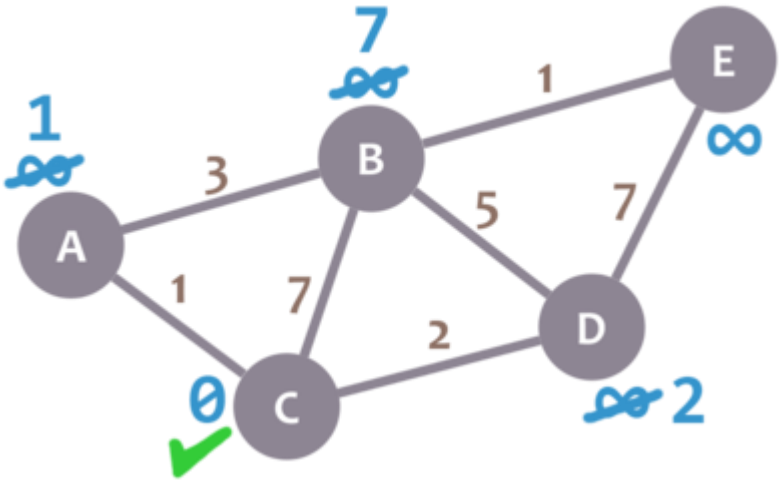
So far, so good. Now, let's check neighbour A. We add 0 (the minimum distance of C, our current node) with 1 (the weight of the edge connecting our current node with A) to obtain 1. We compare that 1 with the minimum distance of A (infinity), and leave the smallest value:



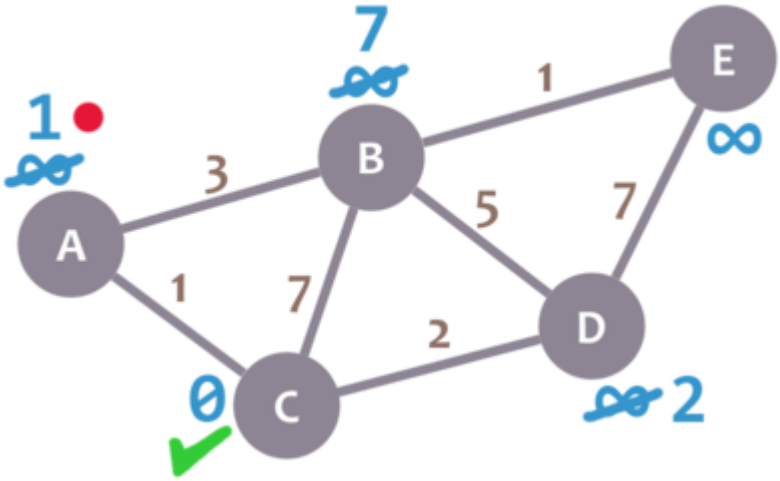
OK. Repeat the same procedure for D:



Great. We have checked all the neighbours of C. Because of that, we mark it as *visited*. Let's represent visited nodes with a green check mark:

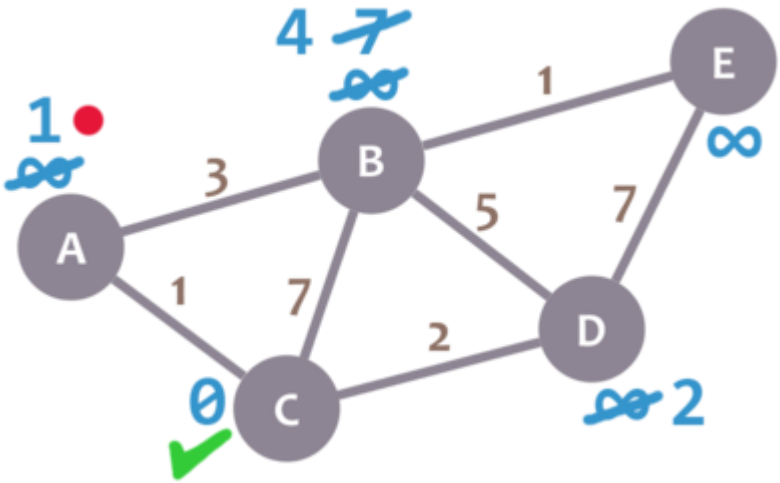


We now need to pick a new *current node*. That node must be the unvisited node with the smallest minimum distance (so, the node with the smallest number and no check mark). That's A. Let's mark it with the red dot:

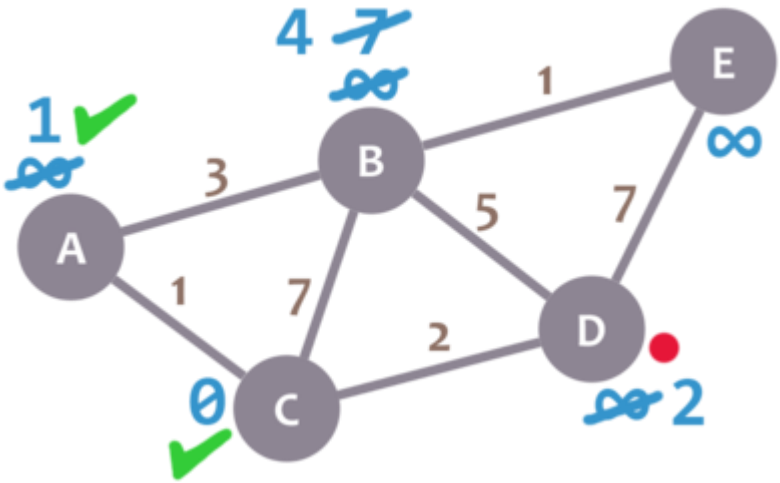


And now we repeat the algorithm. We check the neighbours of our current node, ignoring the visited nodes. This means we only check B.

For B, we add 1 (the minimum distance of A, our current node) with 3 (the weight of the edge connecting A and B) to obtain 4. We compare that 4 with the minimum distance of B (7) and leave the smallest value: 4.



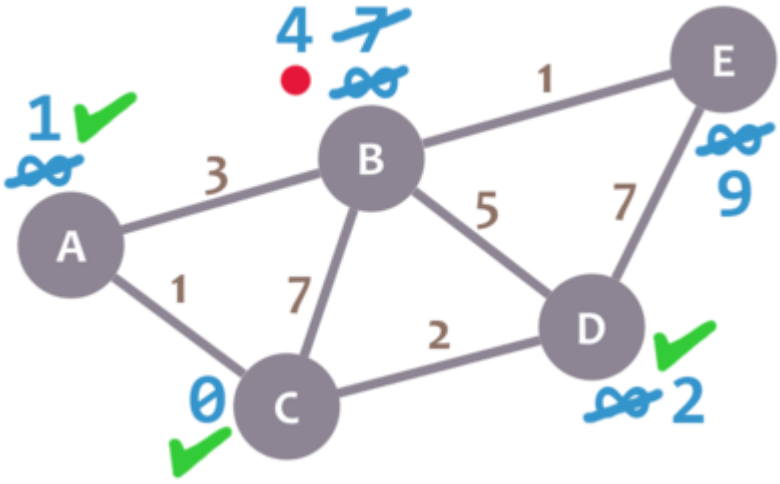
Afterwards, we mark A as visited and pick a new current node: D, which is the non-visited node with the smallest current distance.



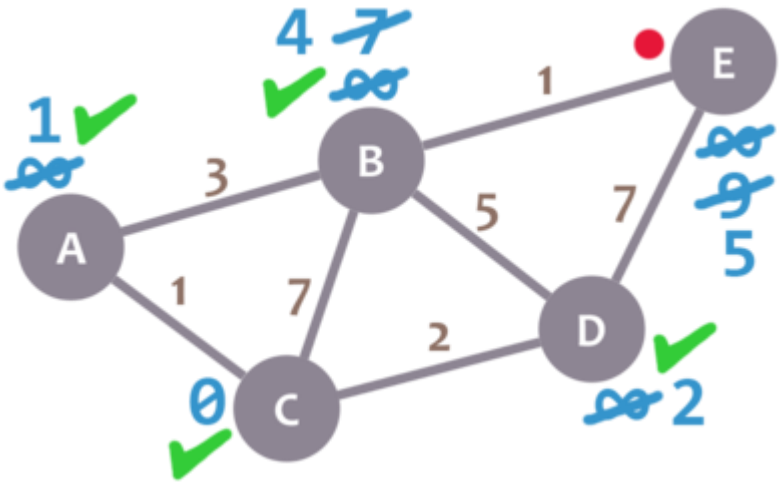
We repeat the algorithm again. This time, we check B and E.

For B, we obtain 2 + 5 = 7. We compare that value with B's minimum distance (4) and leave the smallest value (4). For E, we obtain 2 + 7 = 9, compare it with the minimum distance of E (infinity) and leave the smallest one (9).

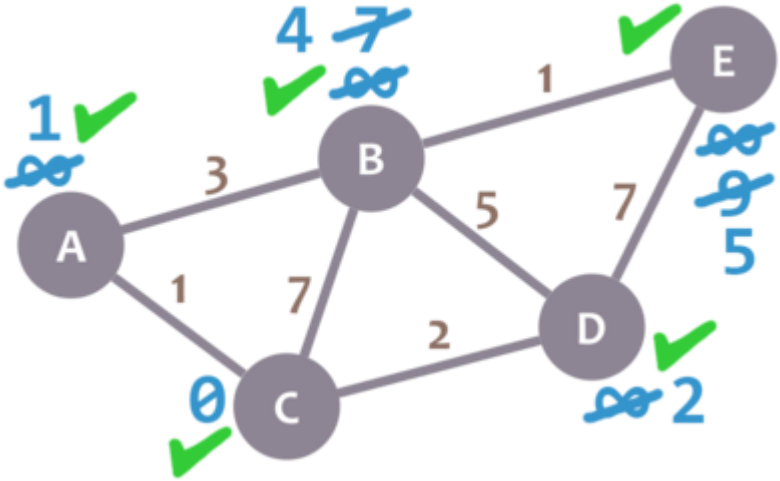
We mark D as visited and set our current node to B.



Almost there. We only need to check E. 4 + 1 = 5, which is less than E's minimum distance (9), so we leave the 5. Then, we mark B as visited and set E as the current node.



E doesn't have any non-visited neighbours, so we don't need to check anything. We mark it as visited.



As there are not univisited nodes, we're done! The minimum distance of each node now actually represents the minimum distance from that node to node C (the node we picked as our initial node)!

Here's a description of the algorithm:

1. Mark your selected initial node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the current node `C`.
3. For each neighbour `N` of your current node `C`: add the current distance of `C` with the weight of the edge connecting `C`-`N`. If it's smaller than the current distance of `N`, set it as the new current distance of `N`.
4. Mark the current node `C` as visited.
5. If there are non-visited nodes, go to step 2.
