# MIN-MAX algorithm:

- The **min max algorithm in AI,** popularly known as the minimax, is a backtracking algorithm.
- It is used in decision making, game theory and artificial intelligence (AI).
- It is used to find the optimal move for a player, assuming that the opponent is also playing optimally.
- Popular two-player computer or online games like Chess, Tic-Tac-Toe, Checkers, Go, etc. use this algorithm.

A backtracking algorithm is used to find a solution to computational problems in such a way that a candidate is incrementally built towards a solution, one step at a time. And the candidate that fails to complete a solution is immediately abandoned.

## How does it work?

In the **min max algorithm in AI,** there are two players, Maximiser and Minimiser. Both these players play the game as one tries to get the highest score possible or the maximum benefit while the opponent tries to get the lowest score or the minimum benefit.

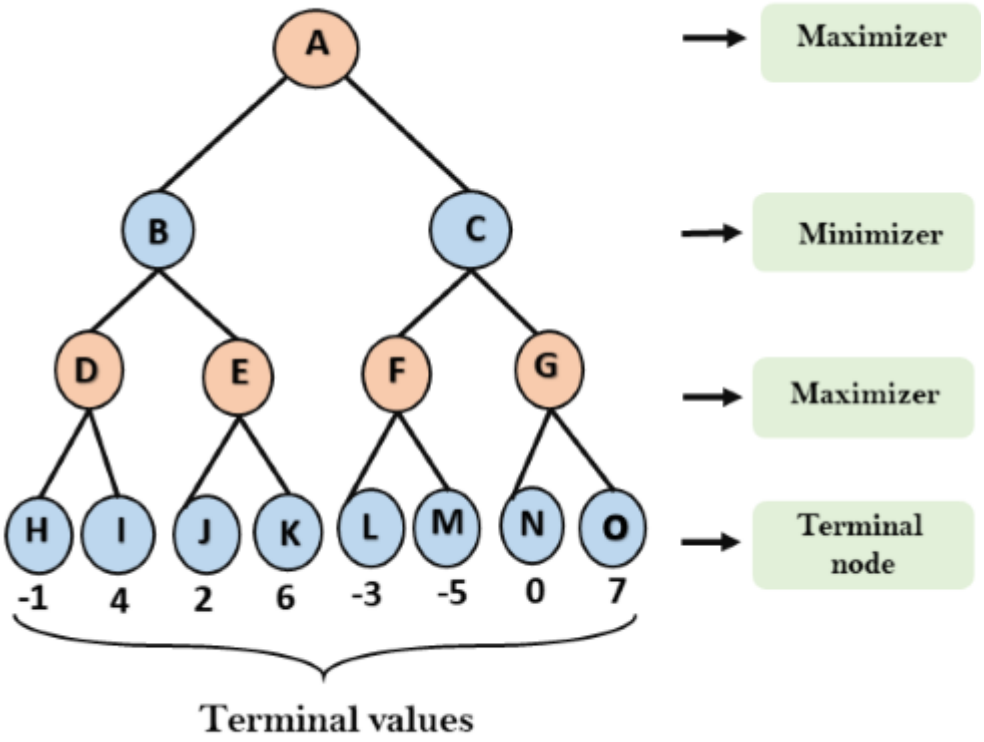
Every game board has an evaluation score assigned to it, so the Maximiser will select the maximised value, and the Minimiser will select the minimised value with counter moves. If the Maximiser has the upper hand, then the board score will be a positive value, and if the Minimiser has the upper hand, then the board score will be a negative value.

## Working of Min-Max Algorithm:

- The working of the minimax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game.
- In this example, there are two players one is called Maximizer and other is called Minimizer.

- Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score.
- This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes.
- At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs. Following are the main steps involved in solving the two-player game tree:

**Step-1:** In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. In the below tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value =- infinity, and minimizer will take next turn which has worst-case initial value = +infinity.

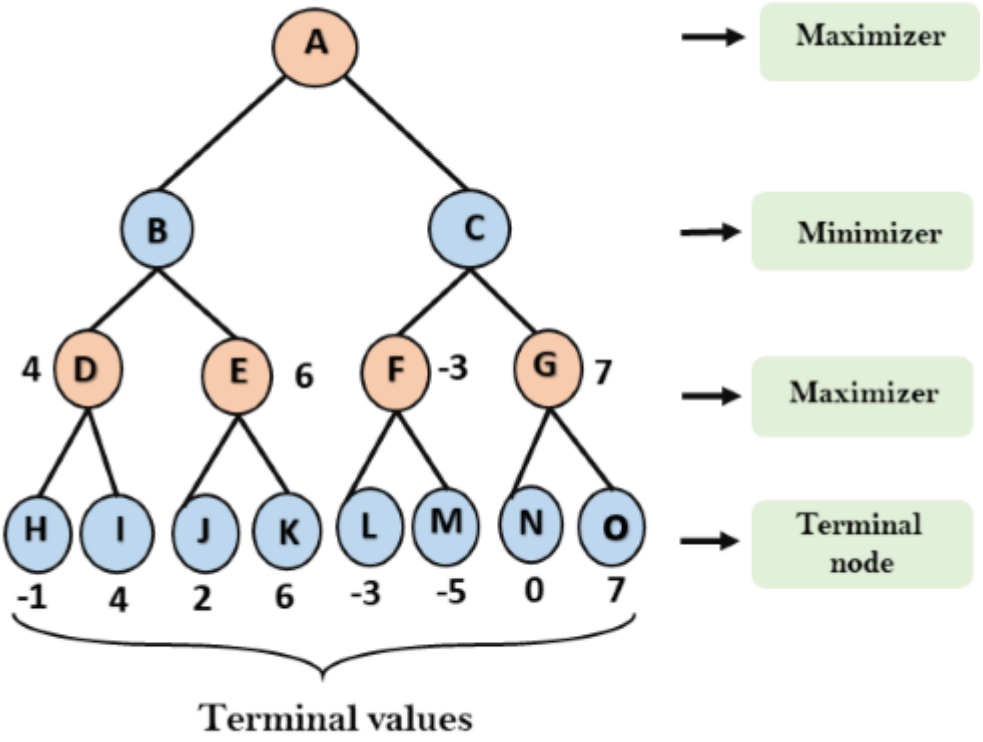



**Step 2:** Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.



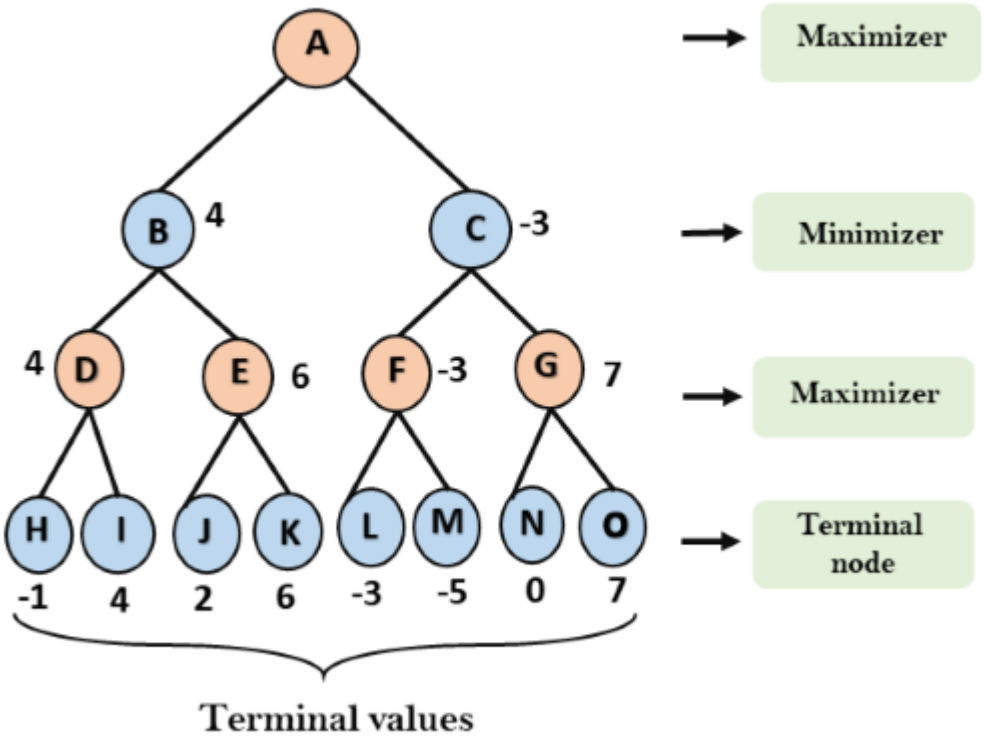
- For node D max(-1,- -∞) => max(-1,4)= 4

- For Node E max(2, -∞) => max(2, 6)= 6
- For Node F max(-3, -∞) => max(-3,-5) = -3
- For node G max(0, -∞) = max(0, 7) = 7



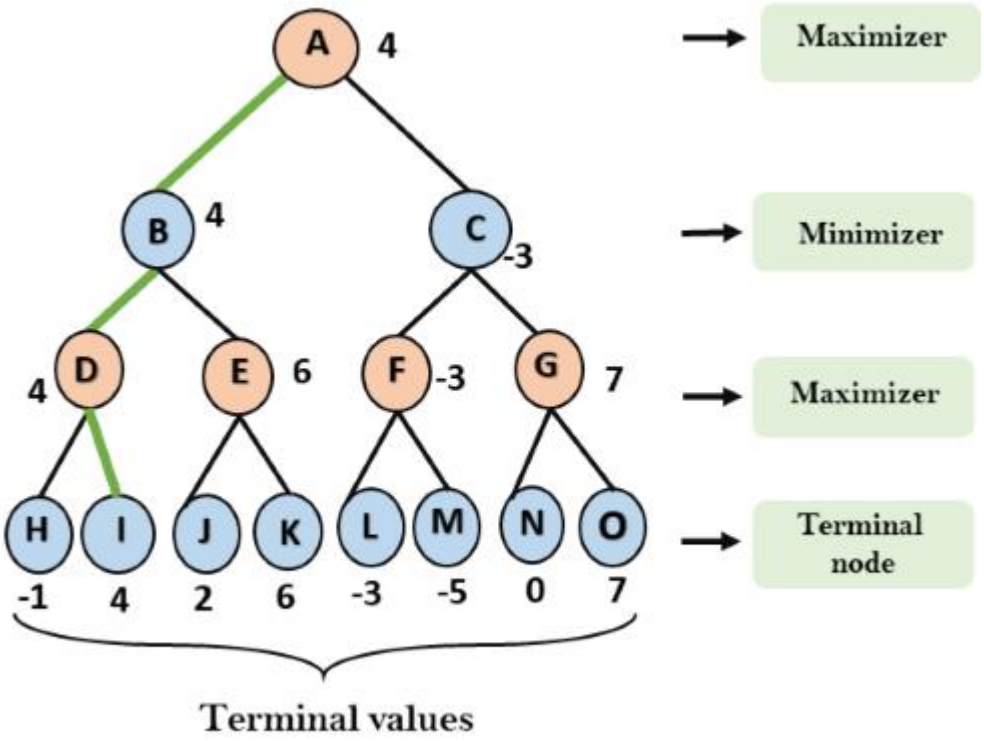

**Step 3:** In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

- For node B= min(4,6) = 4
- For node C= min (-3, 7) = -3




**Step 4:** Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.

- For node A max(4, -3)= 4




That was the complete workflow of the minimax two player game.

## Properties of Mini-Max algorithm:

- **Complete-** Min-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
- **Optimal-** Min-Max algorithm is optimal if both opponents are playing optimally.
- **Time complexity-** As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is **$O(b^m)$**, where b is branching factor of the game-tree, and m is the maximum depth of the tree.
- **Space Complexity-** Space complexity of Mini-max algorithm is also similar to DFS which is **O(bm)**.

## Limitation of the minimax Algorithm:

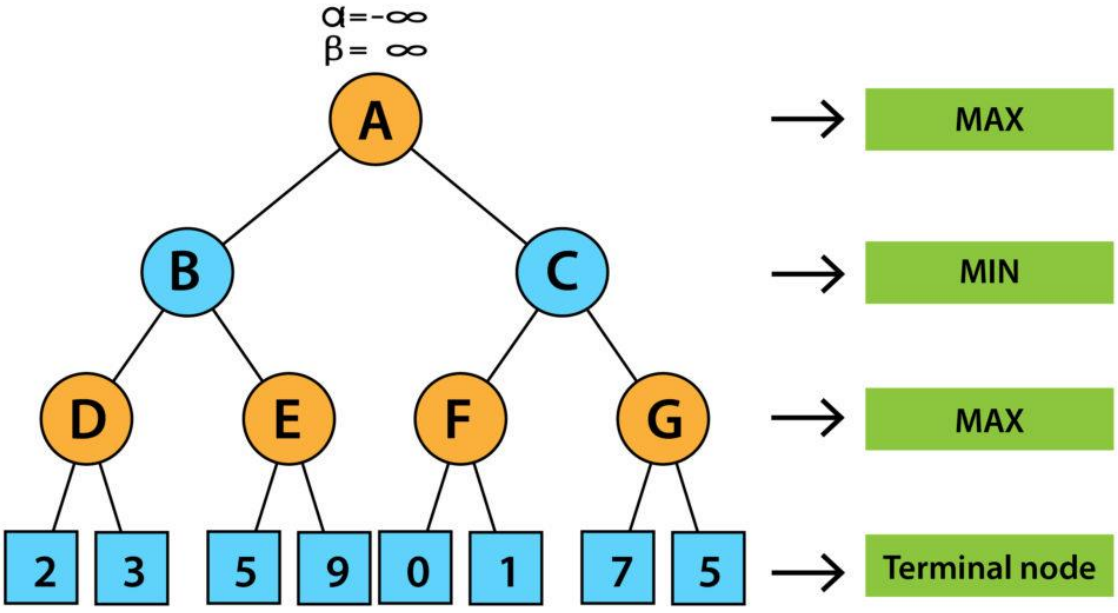
The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc. This type of games has a huge branching factor, and the player has lots of choices to decide. This limitation of the minimax algorithm can be improved from **alpha-beta pruning** which we have discussed in the next topic.

# Key points in Alpha-beta Pruning

- Alpha: Alpha is the best choice or the highest value that we have found at any instance along the path of Maximizer. The initial value for alpha is – ∞.
- Beta: Beta is the best choice or the lowest value that we have found at any instance along the path of Minimizer. The initial value for alpha is + ∞.
- The condition for Alpha-beta Pruning is that $\alpha >= \beta$.
- Each node has to keep track of its alpha and beta values. Alpha can be updated only when it's MAX's turn and, similarly, beta can be updated only when it's MIN's chance.
- MAX will update only alpha values and MIN player will update only beta values.
- The node values will be passed to upper nodes instead of values of alpha and beta during go into reverse of tree.
- Alpha and Beta values only be passed to child nodes.

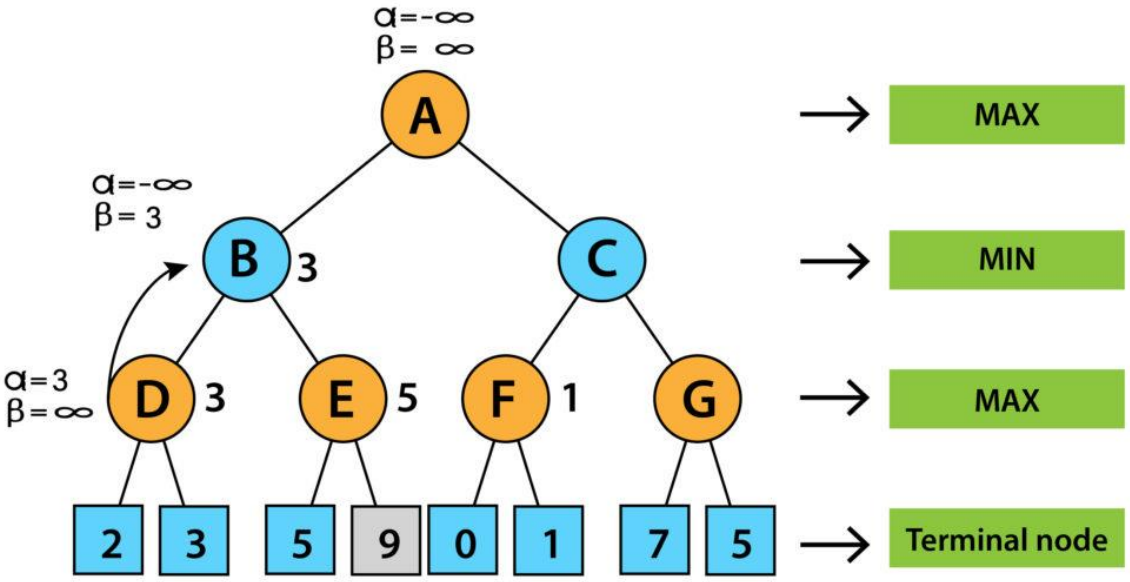
# Working of Alpha-beta Pruning

1. We will first start with the initial move. We will initially define the alpha and beta values as the worst case i.e. $\alpha = -\infty$ and $\beta = +\infty$. We will prune the node only when alpha becomes greater than or equal to beta.



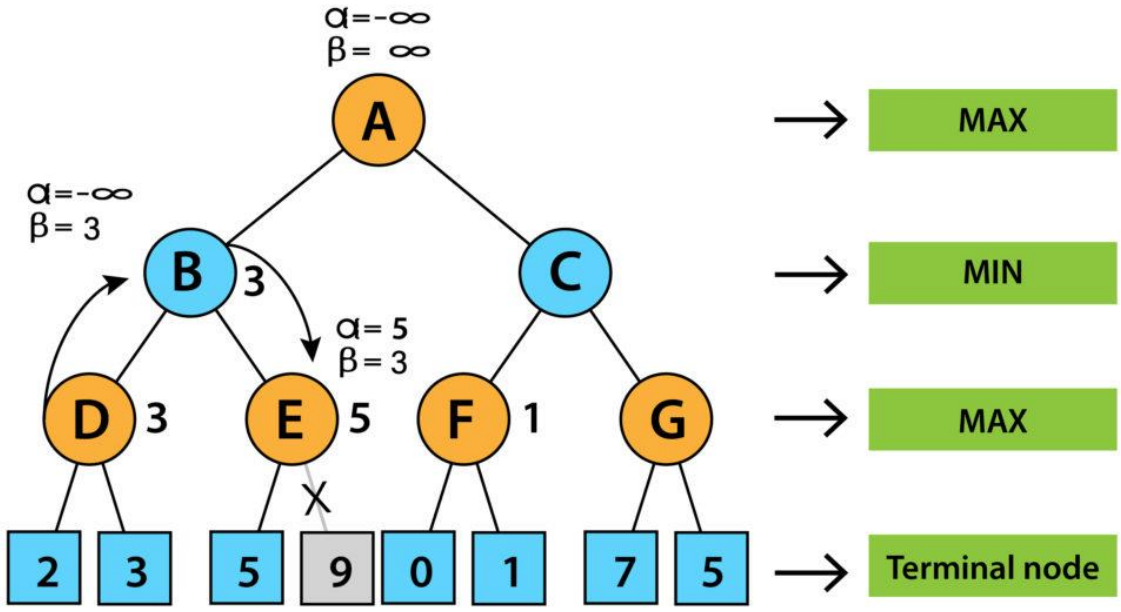
2. Since the initial value of alpha is less than beta so we didn't prune it. Now it's turn for MAX. So, at node D, value of alpha will be calculated. The value of alpha at node D will be max (2, 3). So, value of alpha at node D will be 3.

3. Now the next move will be on node B and its turn for MIN now. So, at node B, the value of alpha beta will be min (3, ∞). So, at node B values will be alpha= – ∞ and beta will be 3.



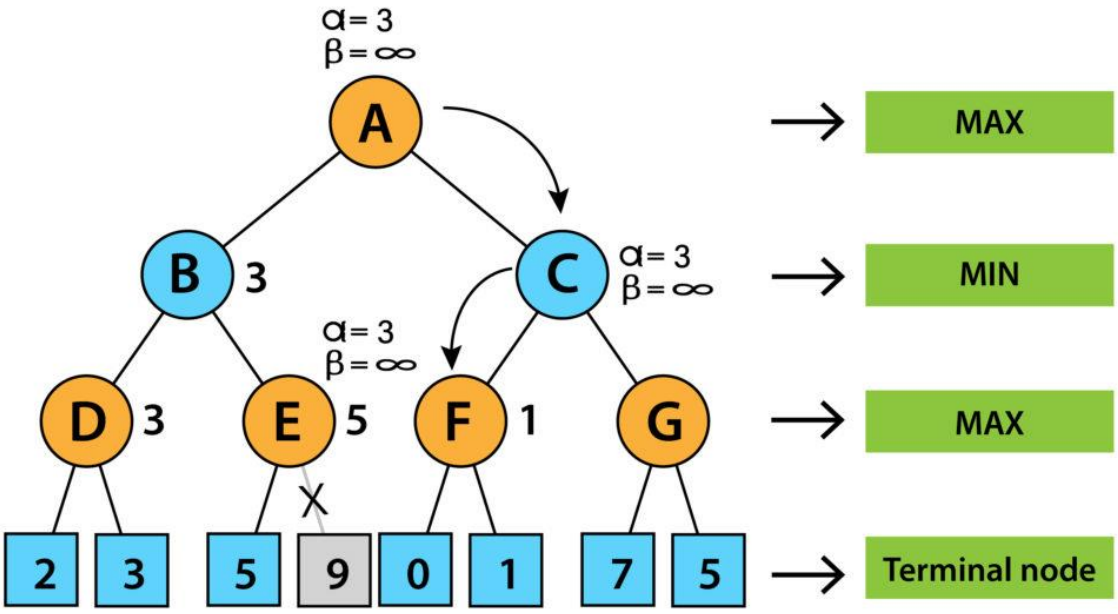

In the next step, algorithms traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

4. Now it's turn for MAX. So, at node E we will look for MAX. The current value of alpha at E is – ∞ and it will be compared with 5. So, MAX (- ∞, 5) will be 5. So, at node E, alpha = 5, Beta = 5. Now as we can see that alpha is greater than beta which is satisfying the pruning condition so we can prune the right successor of node E and algorithm will not be traversed and the value at node E will be 5.

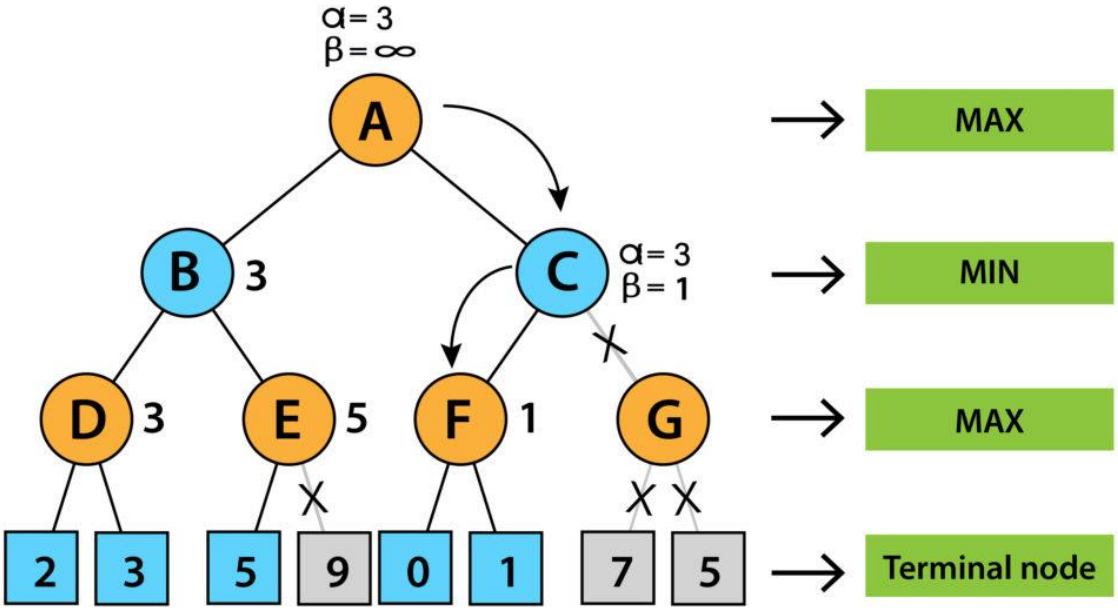



6. In the next step the algorithm again comes to node A from node B. At node A alpha will be changed to maximum value as MAX (- ∞, 3). So now the value of alpha and beta at node A will be (3, + ∞) respectively and will be transferred to node C. These same values will be transferred to node F.

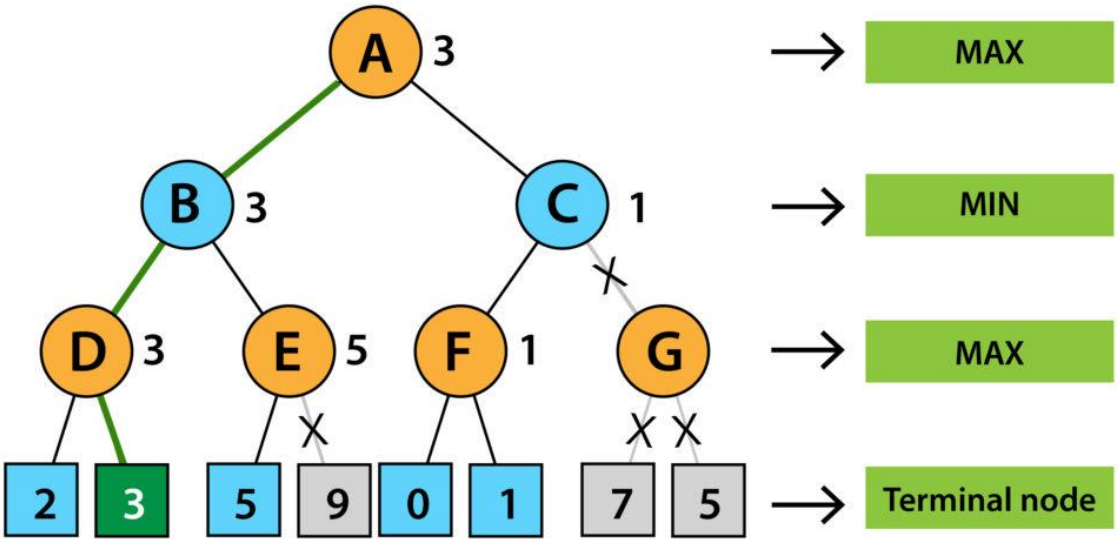
7. At node F the value of alpha will be compared to the left branch which is 0. So, MAX (0, 3) will be 3 and then compared with the right child which is 1, and MAX (3,1) = 3 still α remains 3, but the node value of F will become 1.




8. Now node F will return the node value 1 to C and will compare to beta value at C. Now its turn for MIN. So, MIN (+ ∞, 1) will be 1. Now at node C, α= 3, and β= 1 and alpha is greater than beta which again satisfies the pruning condition. So, the next successor of node C i.e. G will be pruned and the algorithm didn't compute the entire subtree G.




Now, C will return the node value to A and the best value of A will be MAX (1, 3) will be 3.




The above represented tree is the final tree which is showing the nodes which are computed and the nodes which are not computed. So, for this example the optimal value of the maximizer will be 3.