In the MAXMIN Alpha-Beta pruning process we employ two indicators:

- **Alpha:** The best (or highest-value) choice we have found so far at any point along the path of Maximizer.
- The Max player will only update the value of alpha.

Initially, we set it to **-**∞. Winner pessimism, until we otherwise by updating the game.

- **Beta:** The best (lowest-value) choice we have found so far at any point along the path of Minimizer.
- The Min player will only update the value of beta.

Initially, we set to is +∞.

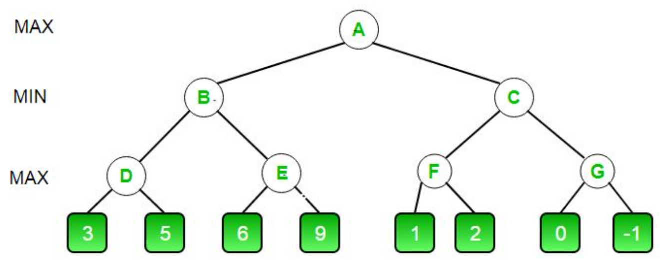
Condition for Alpha-beta pruning:

If we encounter a situation whereby $\alpha>=\beta$ there is no point in pursuing this branch of moves.

(that is if the best option for the Maximizer along the path so far is greater than or equal to the best choice the opponent (the minimizer) has found, then why bother, the minimizer will always play what is best for it, i.e. this lowest value on the maximizer- and hence what we have at this point is the best possible scenario anyway, so no need to go deeper in the search.)

- the node values will be passed to upper nodes in the backtracking process.
- the alpha and beta values are passed downward to the children nodes.

Alpha Beta Pruning- Example



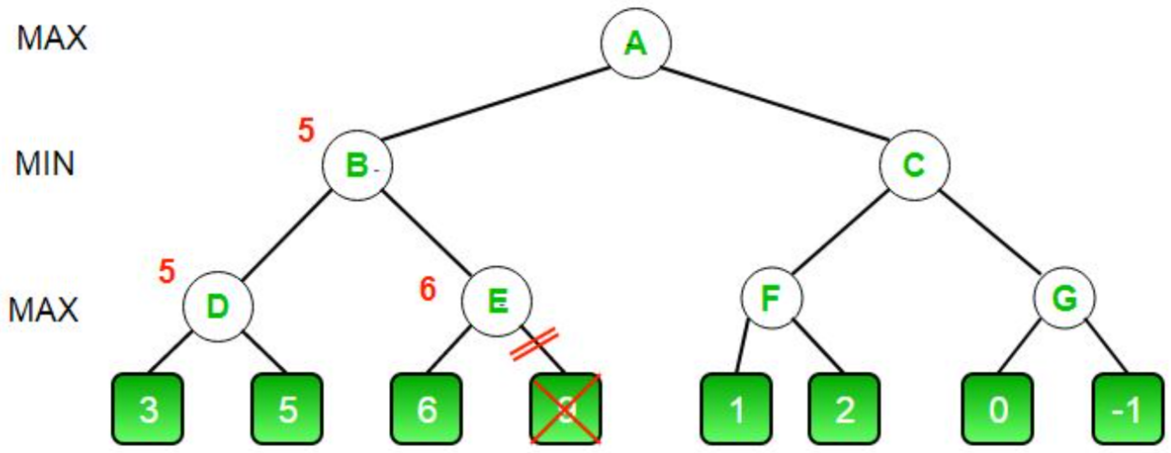
- At A (maximizer): we set alpha to **-INFINITY** and beta to **+INFINITY**.
- the maximizer must choose max of **B** and **C**, so **A** calls **B** first, DF search.
- At **B** (minimizer) must choose min of **D** and **E** and hence calls **D** first (DF strategy).
- At **D (maximizer)**, left child which is a leaf node with value 3. Now the value of alpha at **D** is max(-INF, 3) which is 3 (remember initially we set alpha to -infinity, as best at the beginning).

- Still at D, is it worth it looking at the right node of D?

  We check the condition beta<=alpha (3<=infinity??). This is false since beta = +INF and alpha = 3. We proceed. That is at **D** we look at its right child which returns a value of 5.

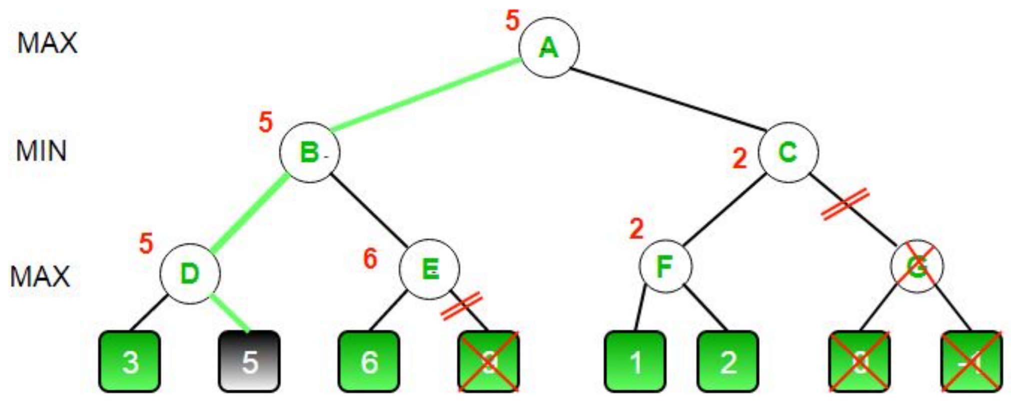
  At **D still**, alpha = max(3, 5) which is 5. Now the value of node **D** is 5

- **D** returns a value of 5 to **B**.
- At **B (Minimizer)**, beta = min(+INF, 5) which is 5. The minimizer is now guaranteed a value of 5 or lesser (depending on E).
- **B** now checks **E** to see if it can get a lower value than 5.
- At **E (Maximizer)** the values of alpha and beta is not -INF and +INF but instead -INF and 5 respectively, because the value of beta was changed at **B.** That is what **B** passes down to **E.**
- Now **E** looks at its left child which is 6.
- At **E**, alpha = max(-INF, 6) which is 6. Here the condition becomes true since beta now is 5 and alpha is 6. So beta<=alpha is true. Hence, it breaks the branch and **E** returns 6 to **B**
- *Note how it did not matter what the value of* ***E****'s right child is. It could have been +INF or -INF, it still wouldn't matter, We never even had to look at it because the minimizer was guaranteed a value of 5 or lesser. So as soon as the maximizer saw the 6 it knew the minimizer would never come this way because it can get a 5 on the left side of* ***B****. This way we didn't have to look at that 9 and hence saved computation time.*
- **E** returns a value of 6 to **B**.
- At **B**, beta = min(5, 6) which is 5.The value of node **B** is also 5.



- **B** returns 5 to **A**. At **A**, alpha = max(-INF, 5) which is 5. Now the maximizer is guaranteed a value of 5 or greater.
- **A** calls **C** to see if it can get a higher value than 5.
- At **C**, alpha = 5 (known from the parent as accomplished outcome so far) and beta = +INF.
- **C** calls **F**
- At **F**, alpha = 5 and beta = +INF. **F** looks at its left child which is a 1. alpha = max(5, 1) which is still 5.
- **F** looks at its right child which is 2. Hence the best value of this node is 2. Alpha still remains 5.
- **F** returns a value of 2 to **C**.
- At **C**, beta = min(+INF, 2). The condition beta <= alpha becomes true as beta = 2 and alpha = 5. So it breaks and it does not even have to compute the entire sub-tree of **G**.
- *The intuition behind this break-off is that, at **C** the minimizer was guaranteed a value of 2 or lesser. But the maximizer was already guaranteed a value of 5 if he choose **B**. So why would the maximizer ever choose **C** and get a value less than 2 ? Again you can see that it did not matter what those last 2 values were. We also saved a lot of computation by skipping a whole sub-tree.*
- **C** now returns a value of 2 to **A**.
- Therefore the best value at **A** is max(5, 2) which is a 5.
- Hence the optimal value that the maximizer can get is 5

The final game tree looks like.


MAX
MIN
MAX
5
A
5
B
2
C
5
D
6
E
2
F
G
3
5
6
1
2