Alpha Beta Pruning

in Artificial Intelligence

Introduction

- *Alpha-beta pruning* is the way through which we can find the optimal minimax solution while avoiding searching subtrees of moves which won't be selected. In the search tree for a two-player game, there are two kinds of nodes, one node represents *your* move and another node represents *opponent's* move.
- Alpha-beta pruning comes from two parameters.
  - They describe bounds on the values that appear anywhere along the path under consideration:
    - α = best value(i.e., highest value) choice found so far along the path for MAX
    - β = best value(i.e., lowest value) choice found so far along the path for MIN

Alpha Beta Pruning

- Alpha-beta pruning gets its name from two bounds that are passed along during the calculation, which restrict the set of possible solutions based on the portion of the search tree that has already been seen. Specifically,
- Beta is the *minimum upper bound* of possible solutions
- Alpha is the *maximum lower bound* of possible solutions
- Thus, when any new node is being considered as a possible path to the solution, it can only work if:

$$\alpha \leq N \leq \beta$$

where N is the current estimate of the value of the node

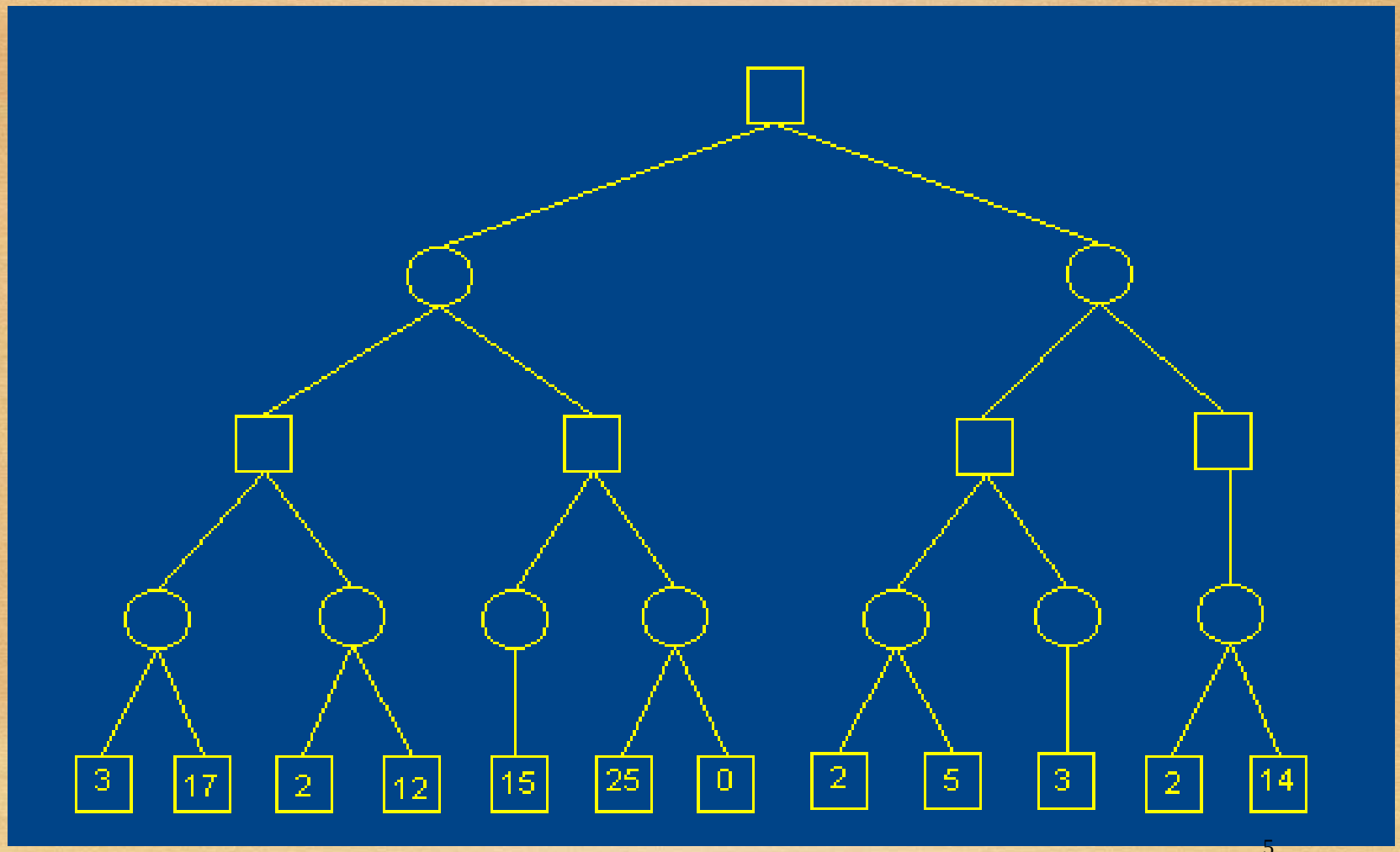
Algorithm: Alpha Beta Search

```
function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game
  v←MAX-VALUE(state, -∞ , +∞)
  return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state, α , β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← -∞
  for a,s in SUCCESSORS(state) do
    v ← MAX(v,MIN-VALUE(s, α , β))
    if v ≥ β then return v
    α ← MAX(α ,v)
  return v
```

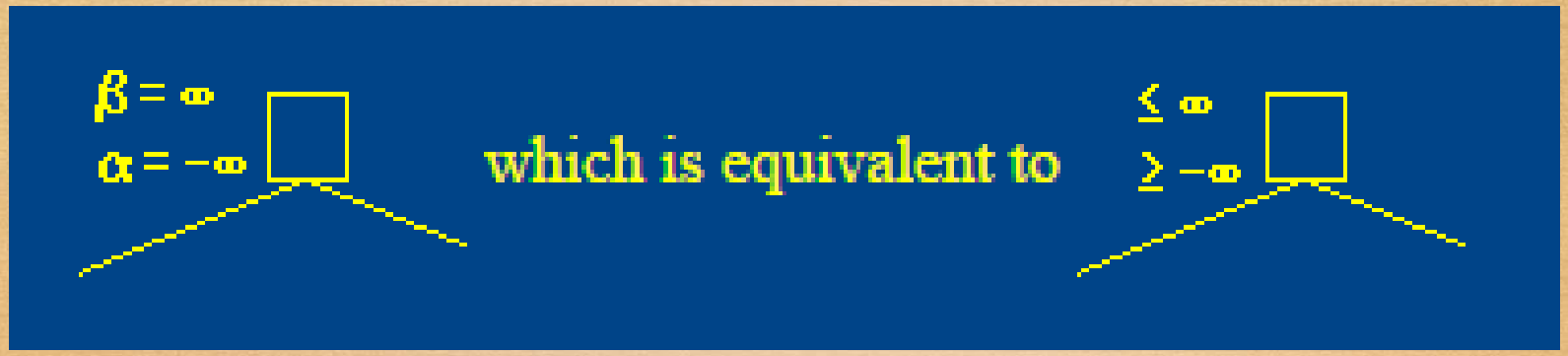
```
function MIN-VALUE(state, α , β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for a,s in SUCCESSORS(state) do
    v ← MIN(v,MAX-VALUE(s, α , β))
    if v ≤ α then return v
    β ← MIN(β ,v)
  return v
```

Example



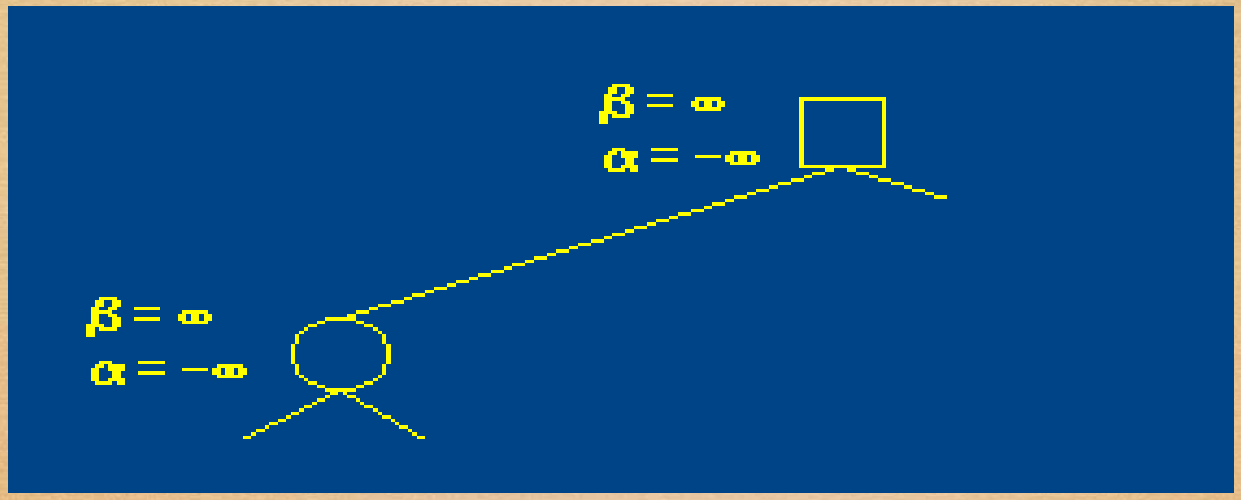
Initial Assumption for Alpha an d Beta

- At begining, we see only the current state (i.e. the current position of pieces on the game board). As for upper and lower bounds, all we know is that it's a number less than infinity and greater than negative infinity. Thus, here's what the initial situation looks like:



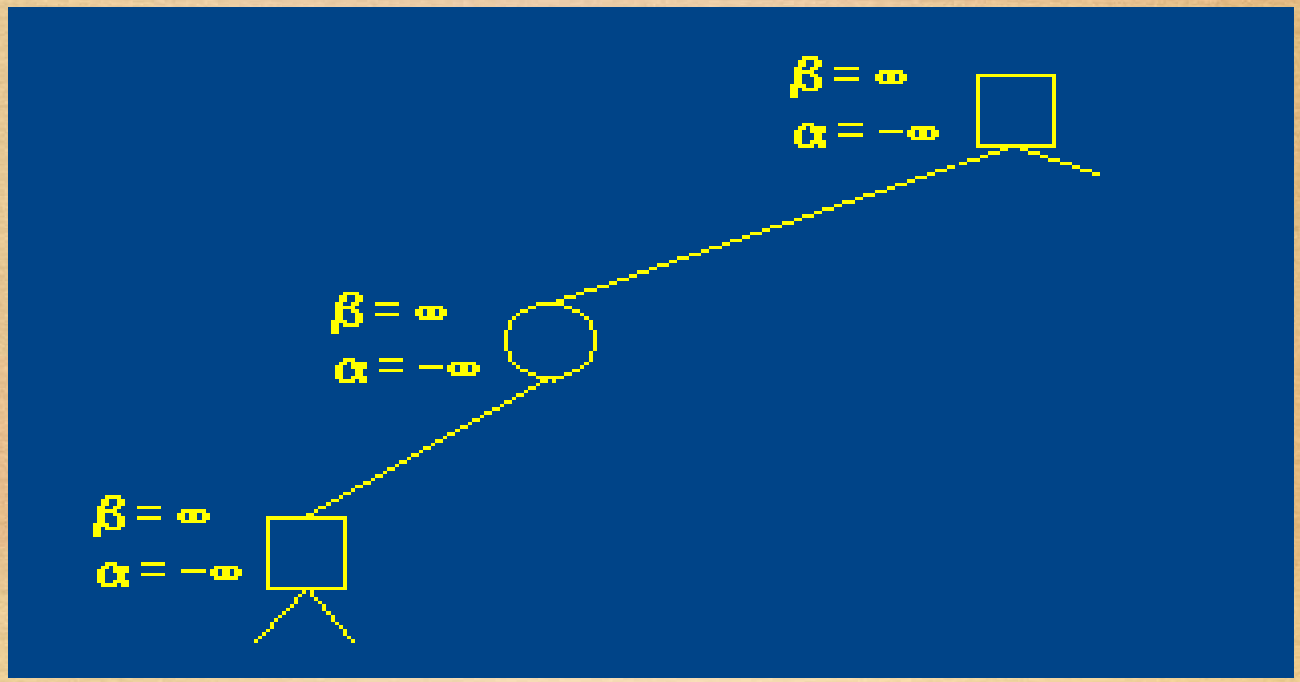
Example

- Since the bounds still contain a valid range, we start the problem by generating the first child state, and passing along the current set of bounds. At this point our search looks like this:



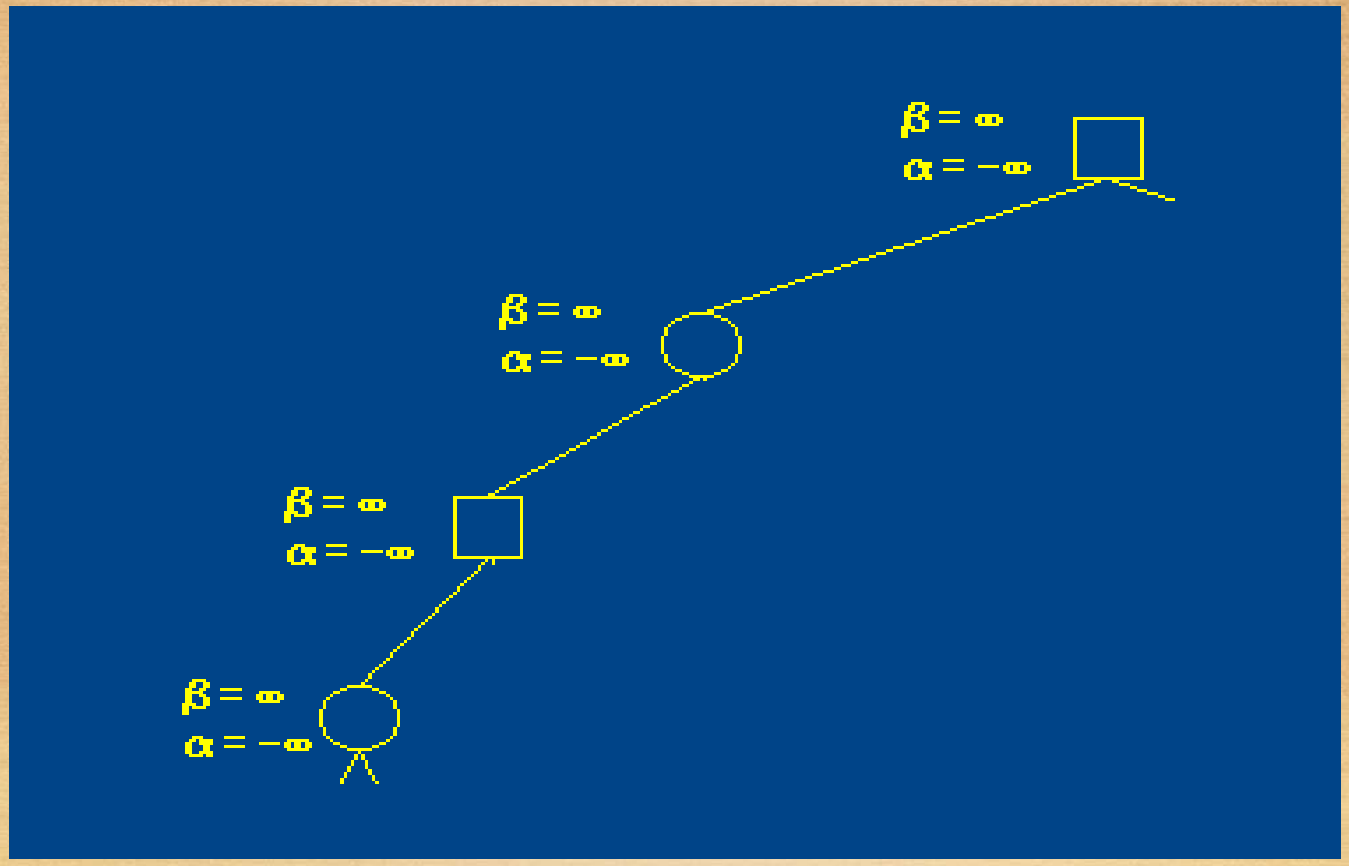
Example

- We are still not down to depth 4, so once again we will generate the first child node and pass along our current alpha and beta values:



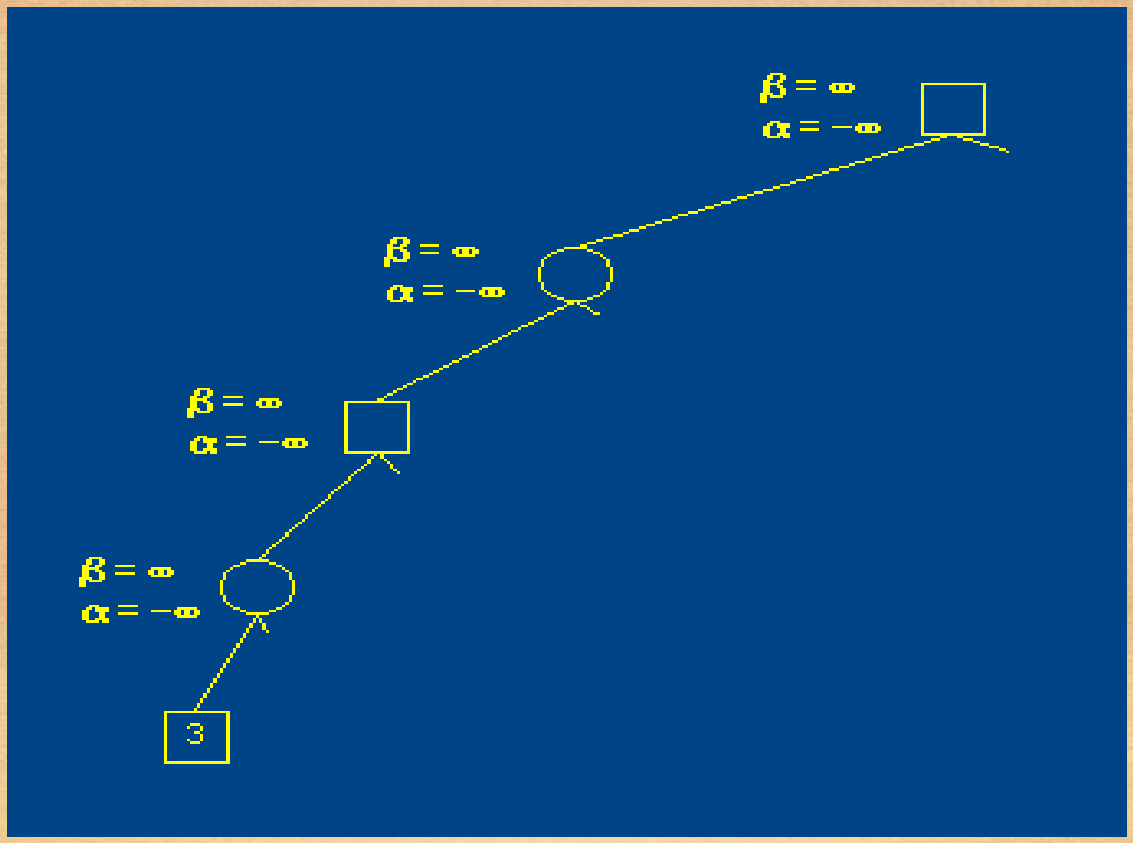
Example

And once more time



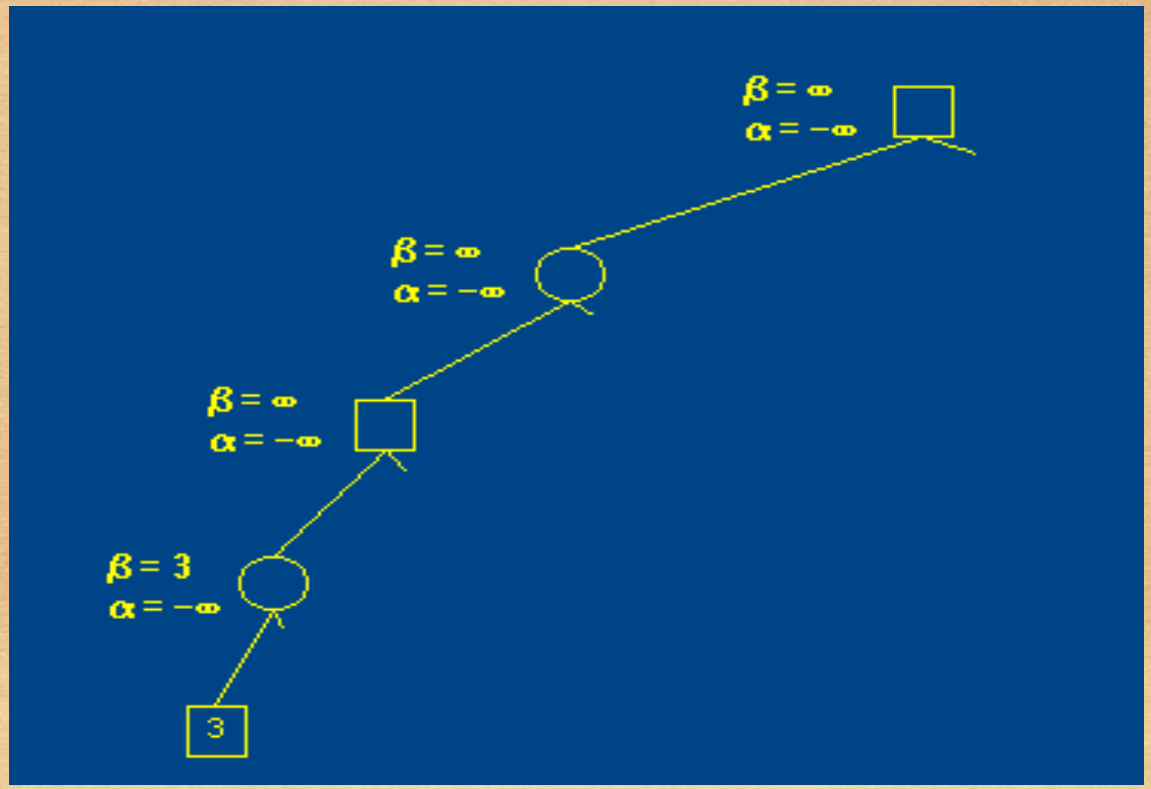
Example

- When we get to the first node at depth 4, we run our evaluation function on the state, and get the value 3. Thus we have this:



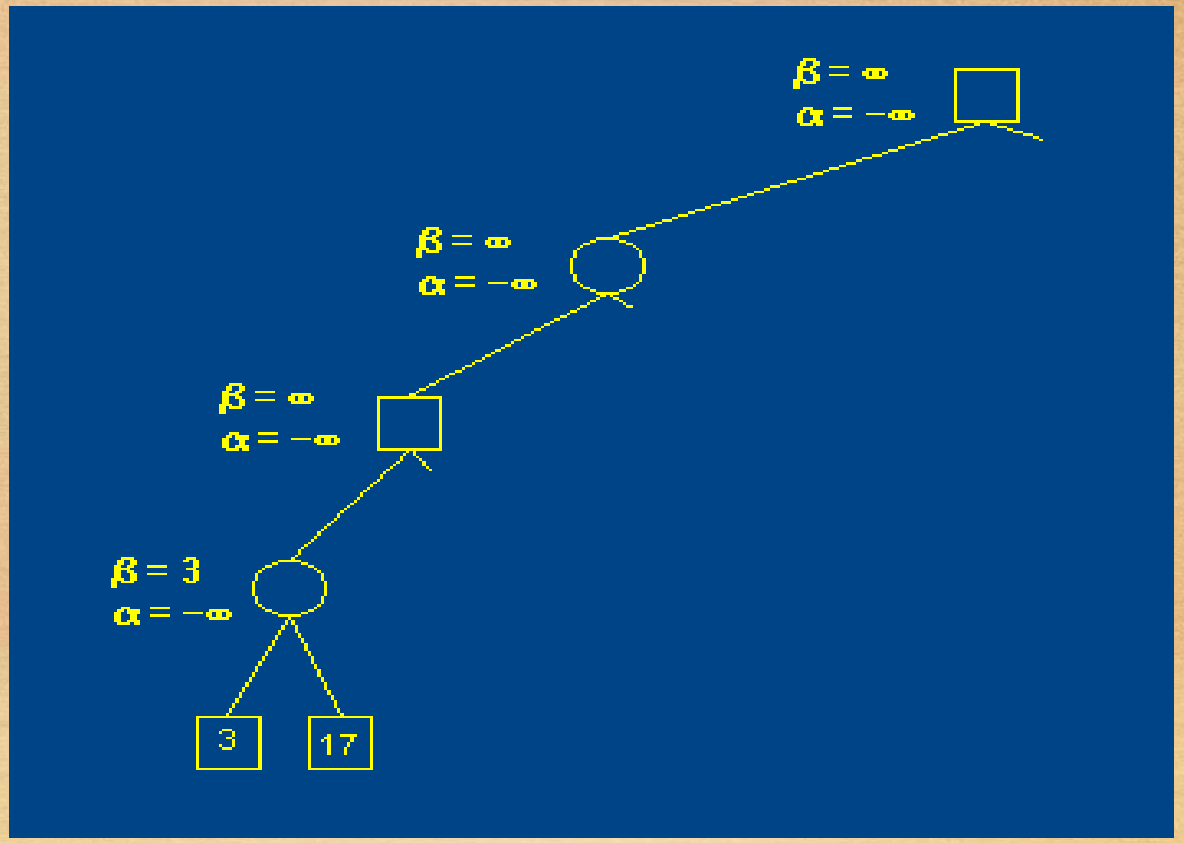
Example

- We pass this node back to the min node above. Since this is a min node, we now know that the minimax value of this node must be less than or equal to 3. In other words, we change beta to 3.



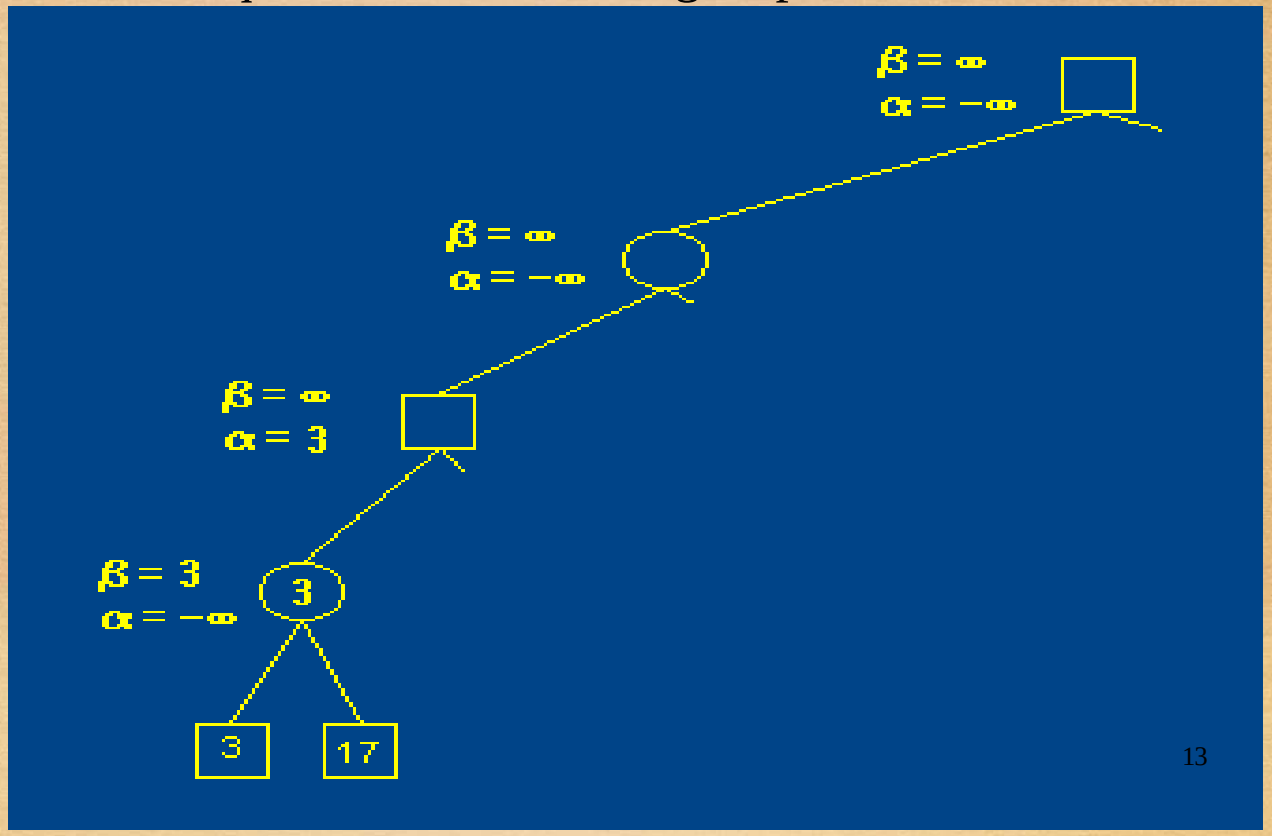
Example

- Next we generate the next child at depth 4, run our evaluation function, and return a value of 17 to the min node above:



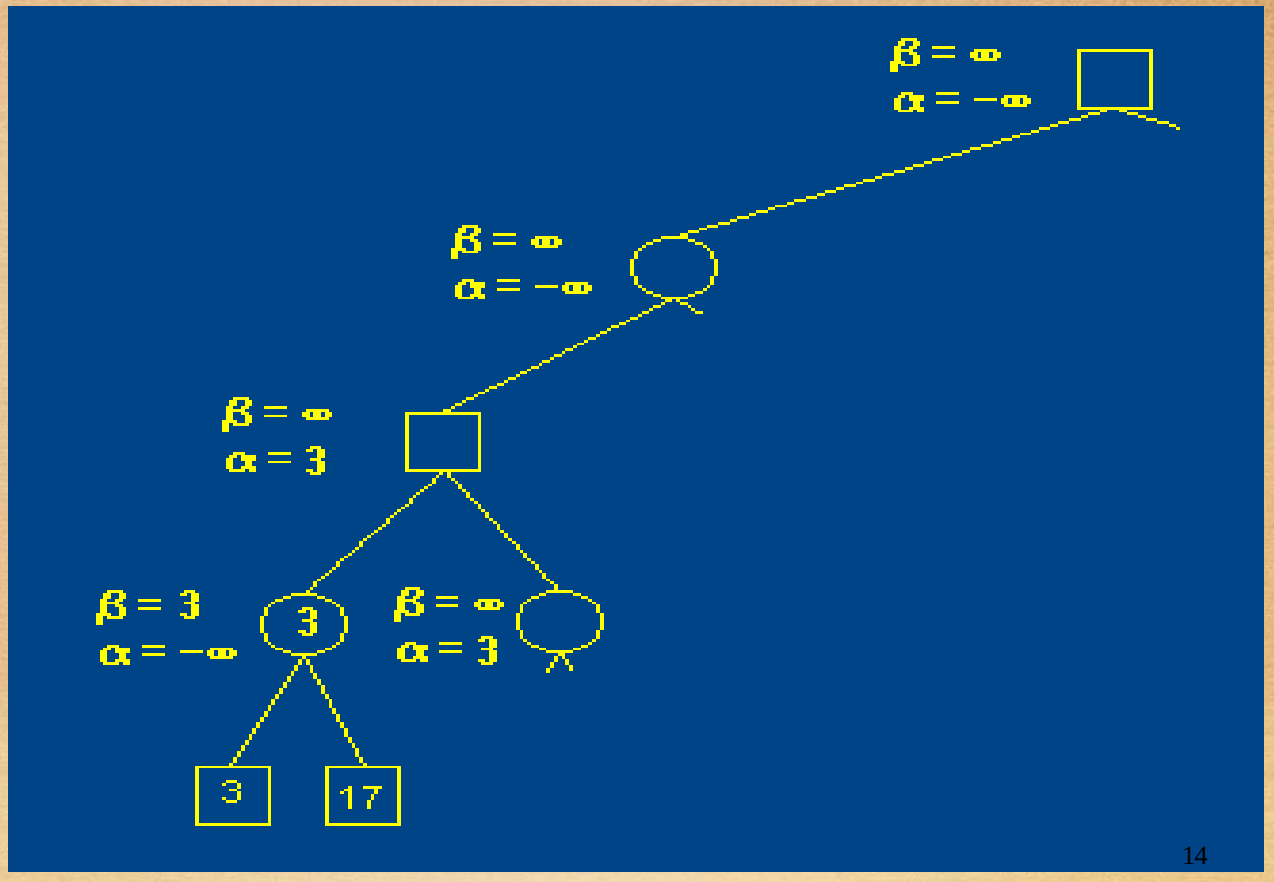
Example

- Since this is a min node and 17 is greater than 3, this child is ignored. Now we've seen all of the children of this min node, so we return the beta value to the max node above. Since it is a max node, we now know that it's value will be greater than or equal to 3, so we change alpha to 3:



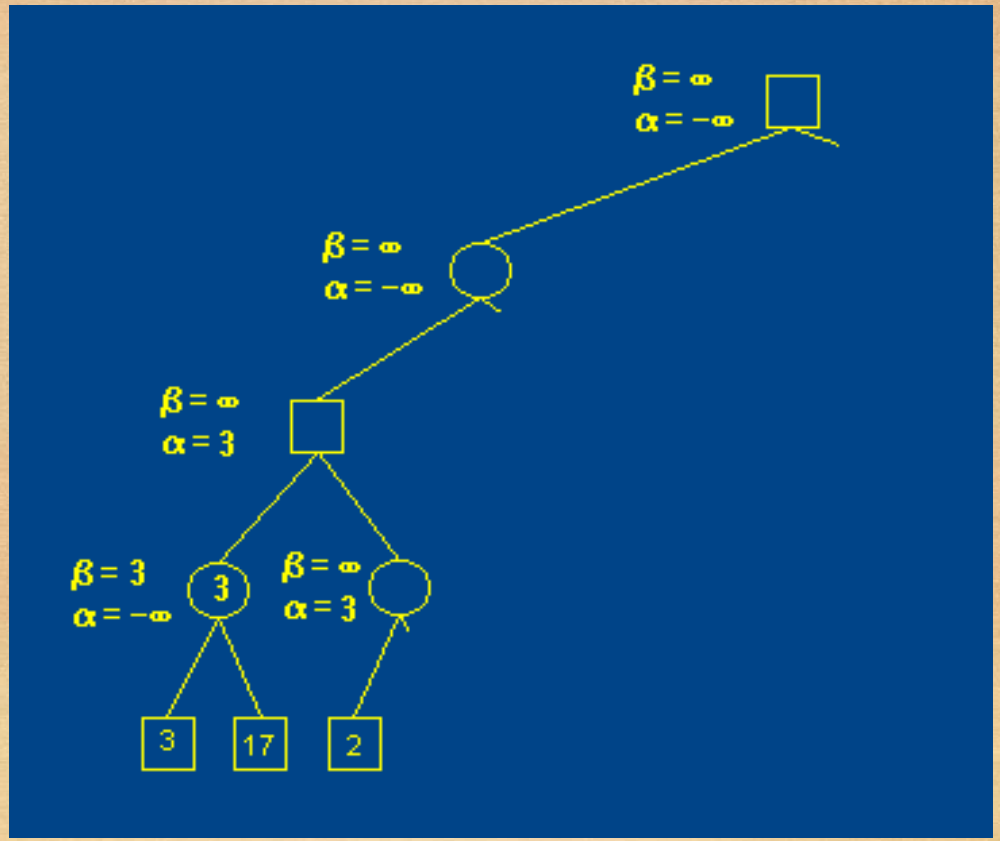
Example

- We generate the next child and pass the bounds along



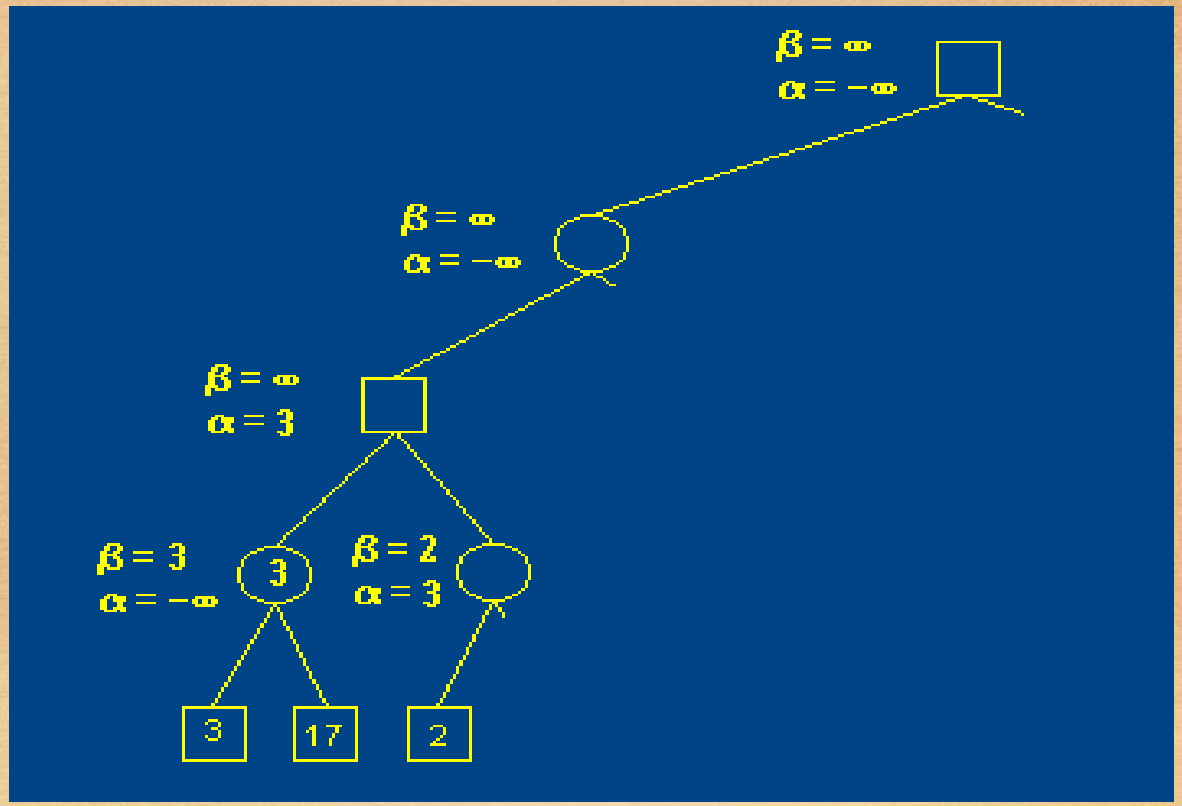
Example

- Since this node is not at the target depth, we generate its first child, run the evaluation function on that node, and return it's value



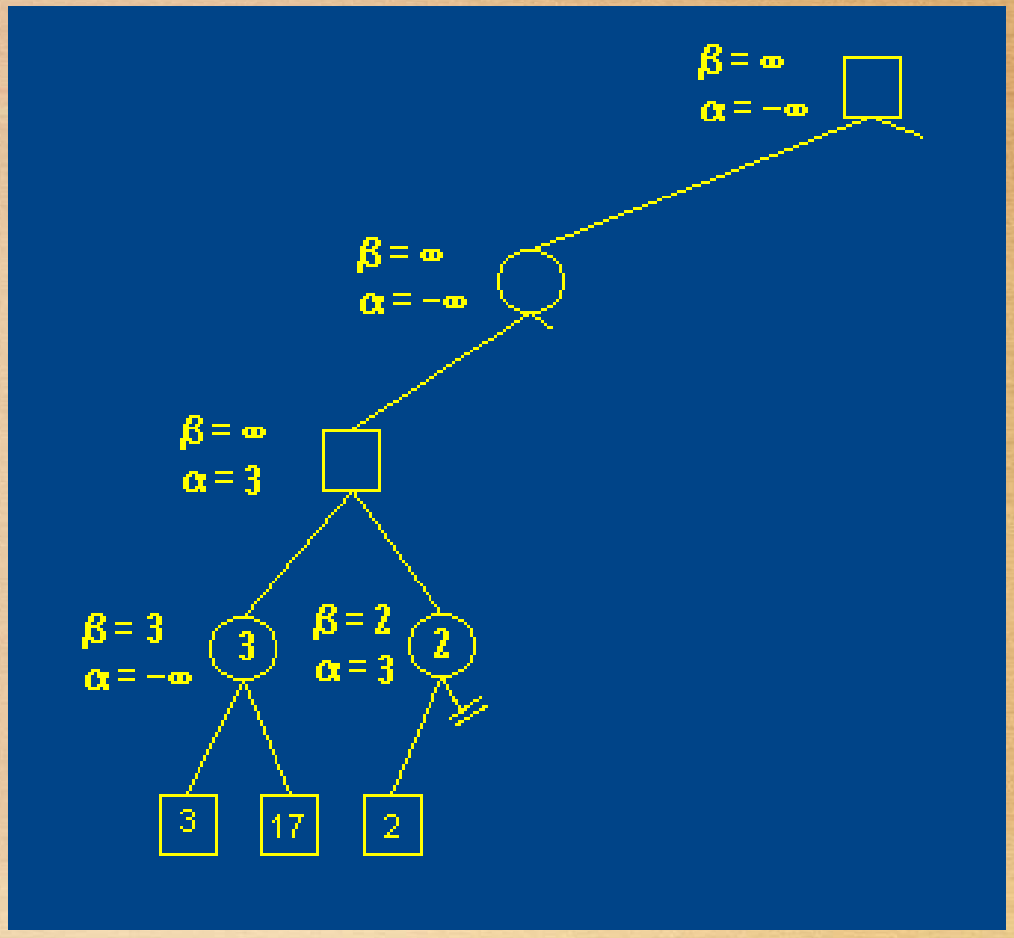
Example

- Since this is a min node, we now know that the value of this node will be less than or equal to 2, so we change beta to 2:



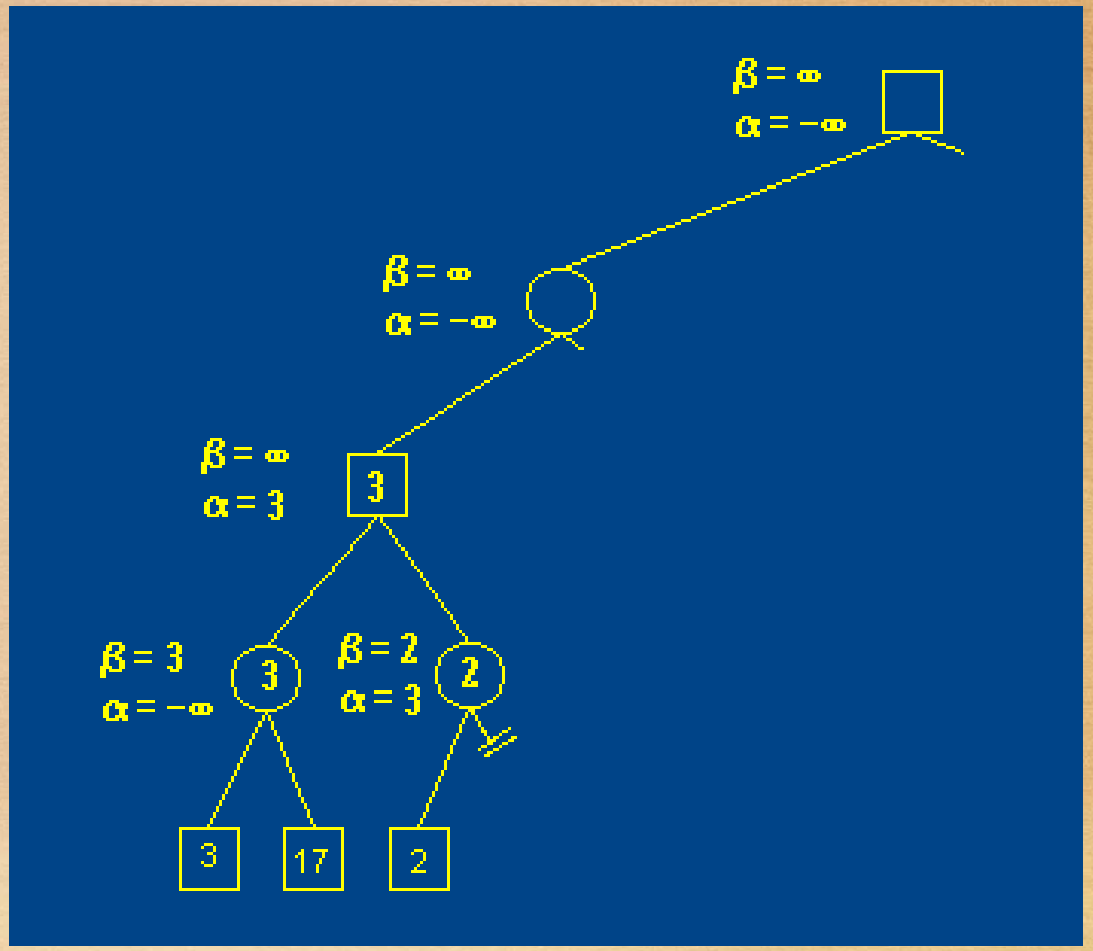
Example

- Certainly, we don't know the actual value of the node. There could be a 1 or 0 or -100 somewhere in the other children of this node. But even if there was such a value, searching for it won't help us find the optimal solution in the search tree. The 2 alone is enough to make this subtree fruitless, so we can prune any other children and return it.
- **That's all there is to beta Pruning…….**



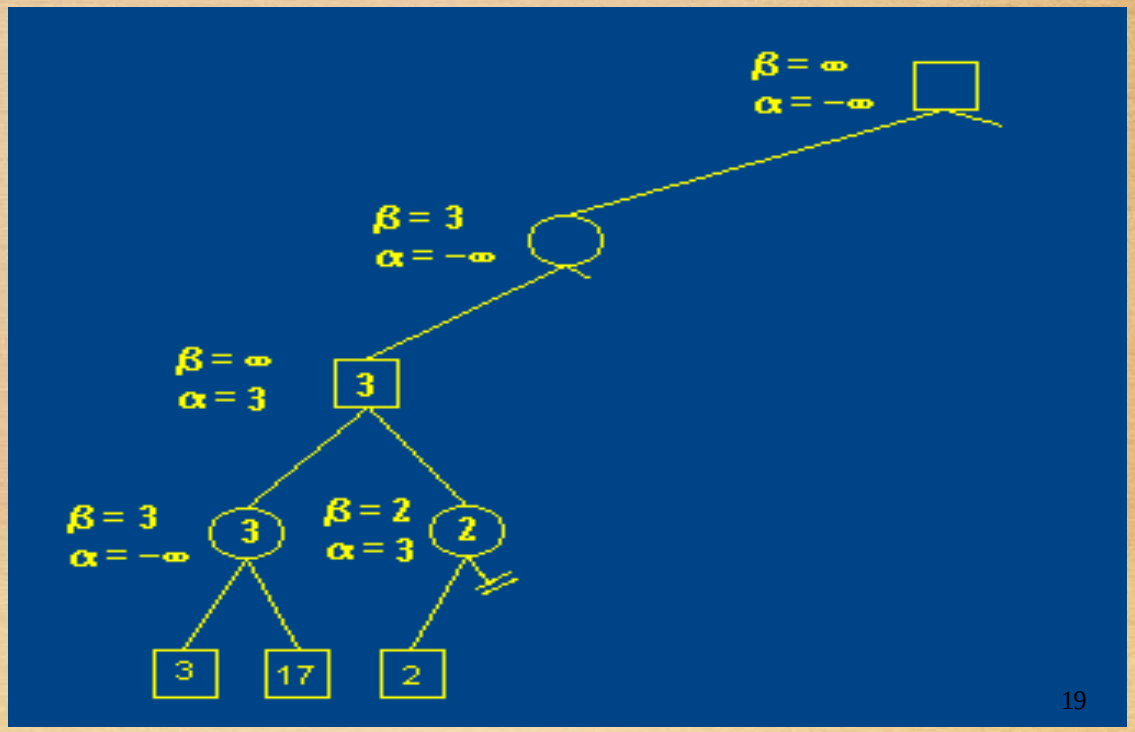
Example

- Back at the parent max node, our alpha value is already 3, which is more restrictive than 2, so we don't change it. At this point we've seen all the children of this max node, so we can set its value to the final alpha value:



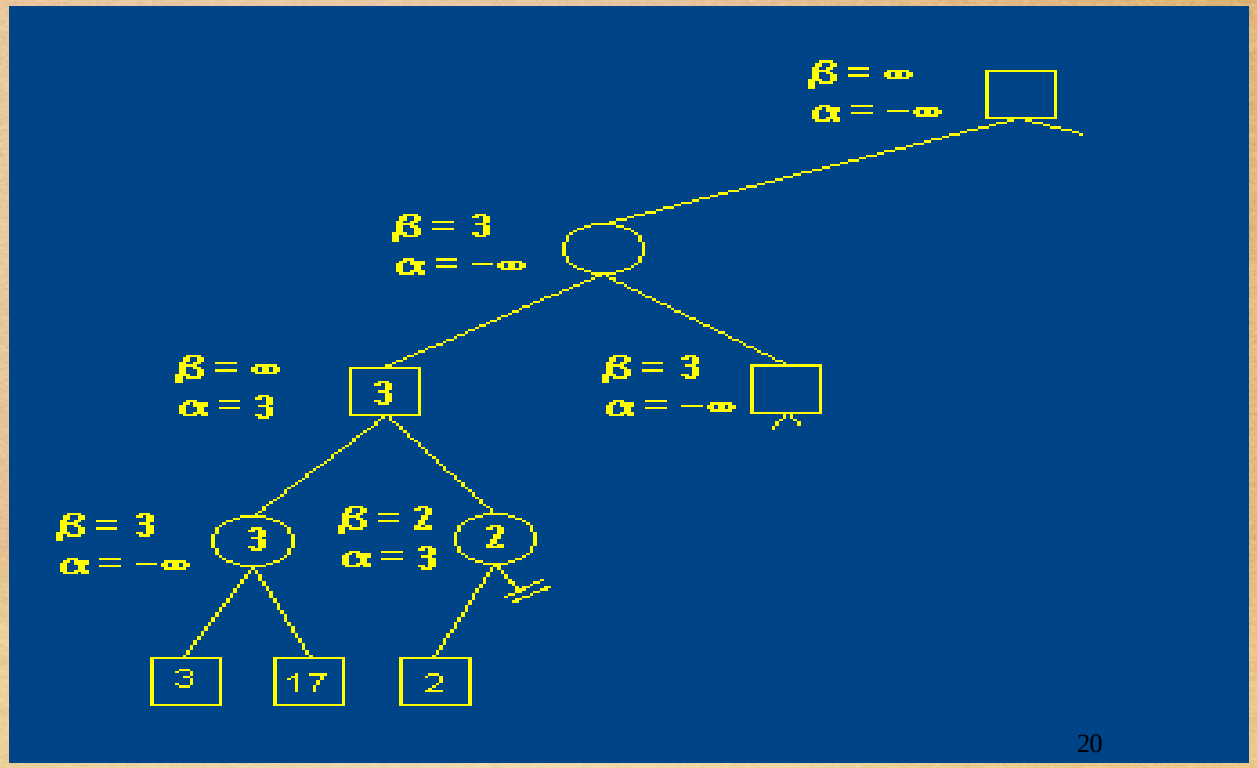
Example

- Now we move on to the parent min node. With the 3 for the first child value, we know that the value of the min node must be less than or equal to 3, thus we set beta to 3:



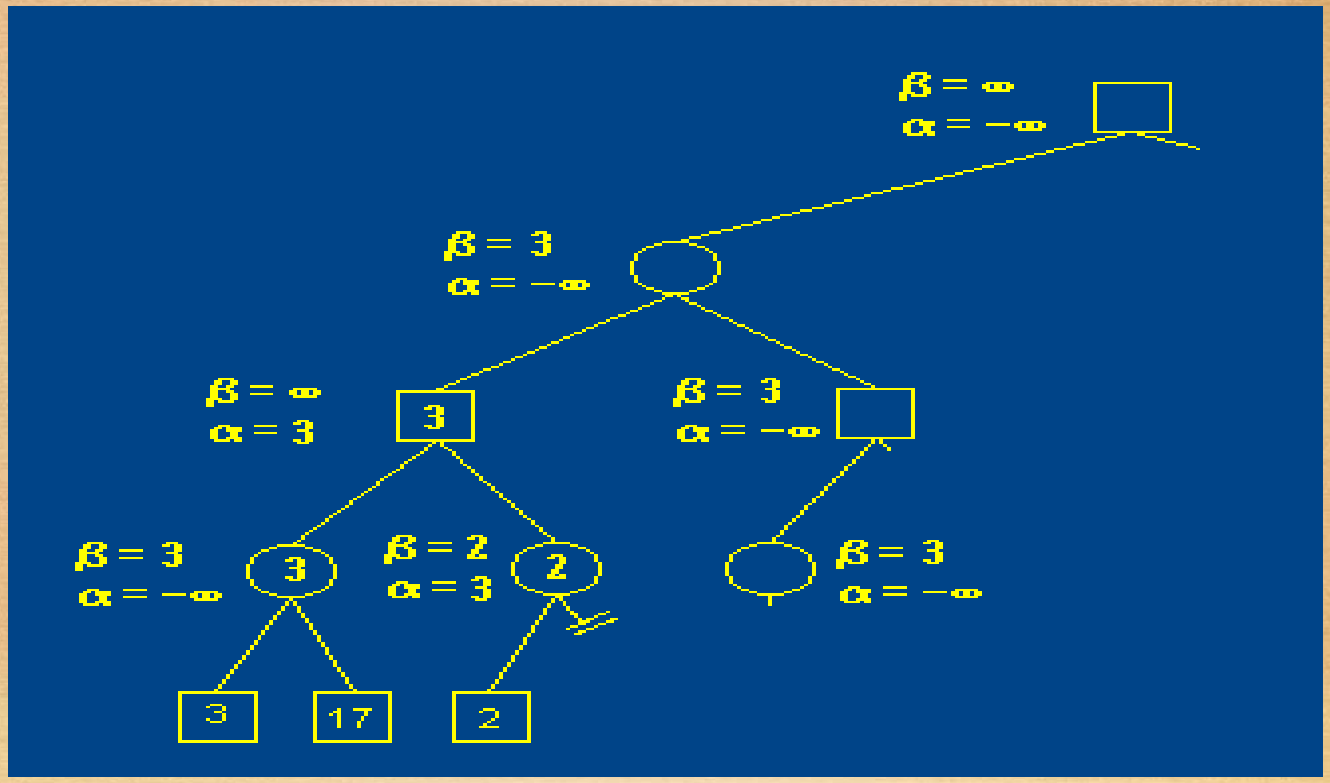
Example

- Since we still have a valid range, we go on to explore the next child. We generate the max node...



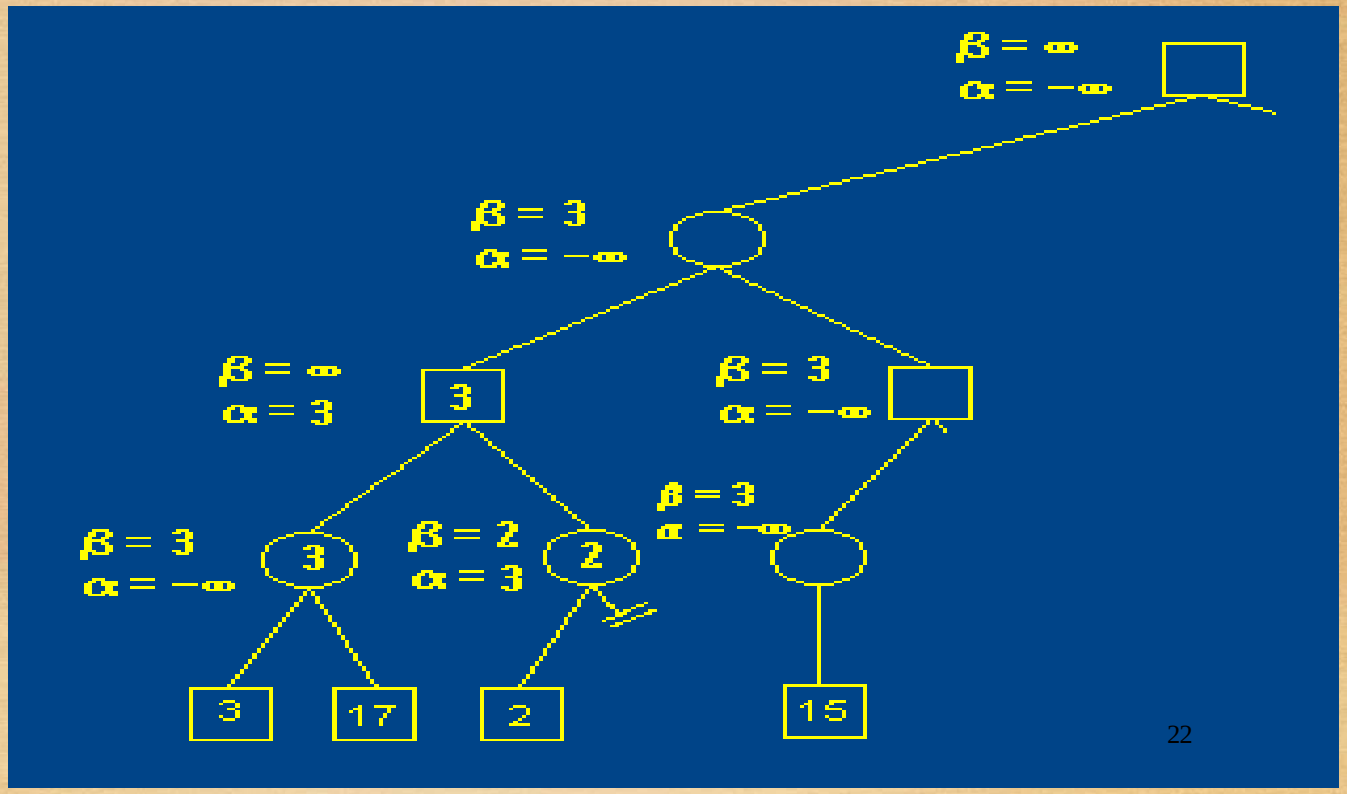
Example

- ... it's first child min node ...



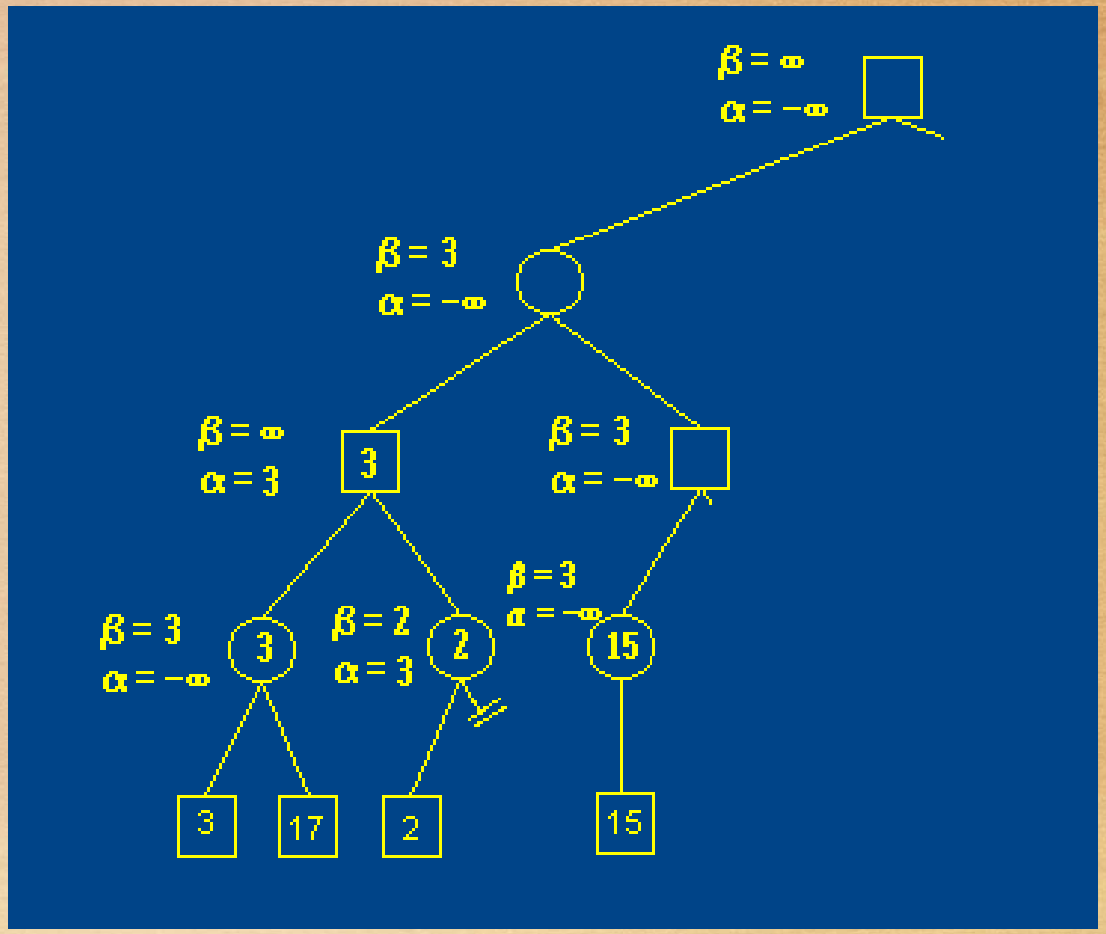
Example

- ... and finally the max node at the target depth. All along this path, we merely pass the alpha and beta bounds along.



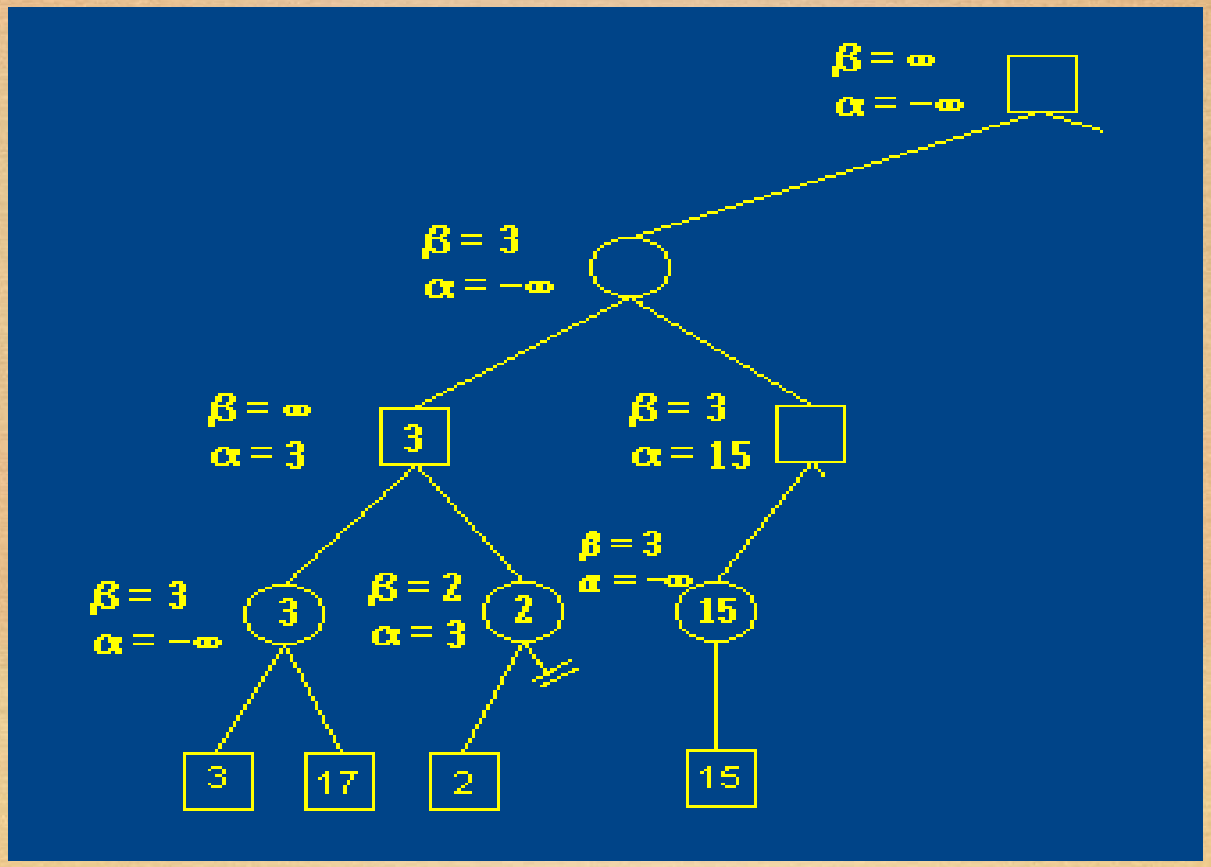
Example

- At this point, we've seen all of the children of the min node, and we haven't changed the beta bound. Since we haven't exceeded the bound, we should return the actual min value for the node. Notice that this is different than the case where we pruned, in which case you returned the beta value. The reason for this will become apparent shortly.



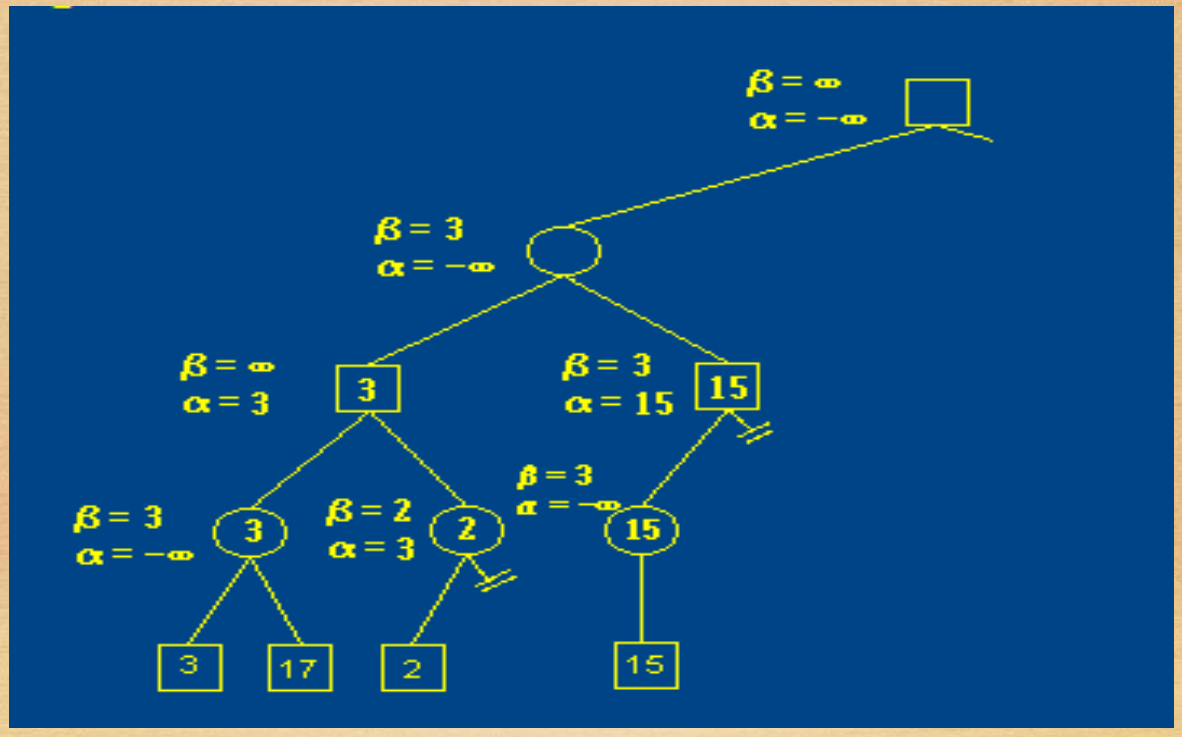
Example

- Now we return the value to the parent max node. Based on this value, we know that this max node will have a value of 15 or greater, so we set alpha to 15:



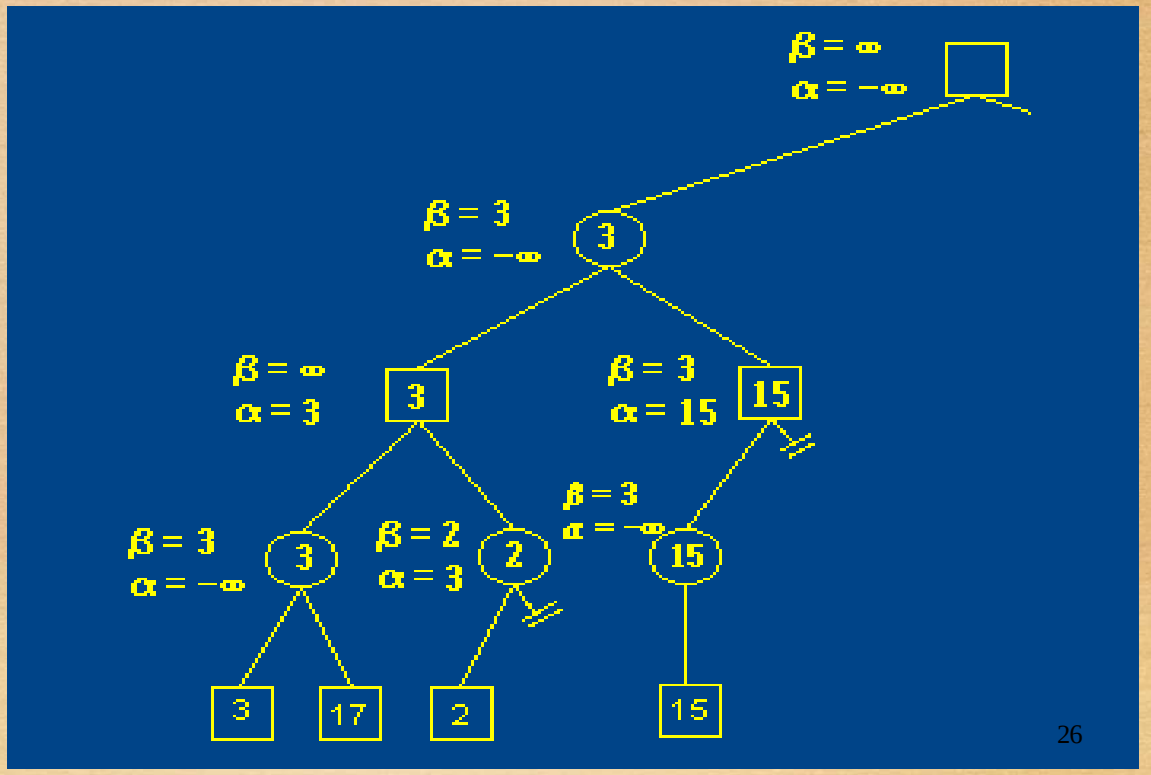
Example

- Once again the alpha and beta bounds have crossed, so we can prune the rest of this node's children and return the value that exceeded the bound (i.e. 15). Notice that if we had returned the beta value of the child min node (3) instead of the actual value (15), we wouldn't have been able to prune here.



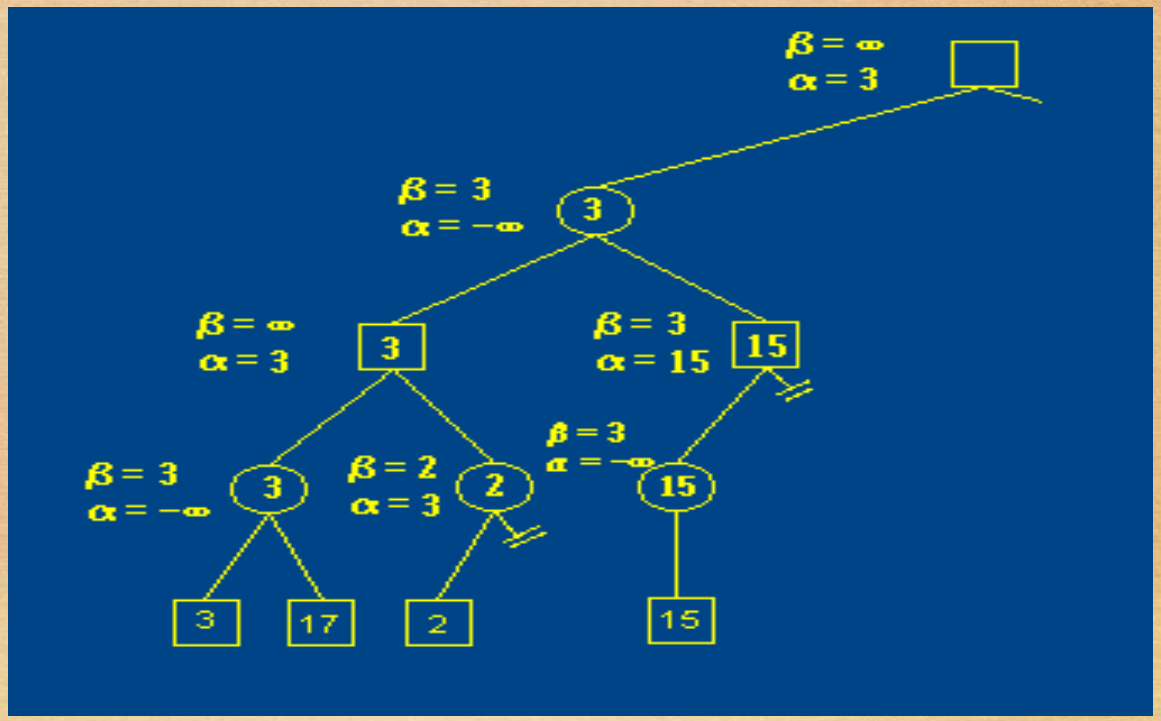
Example

- Now the parent min node has seen all of it's children, so it can select the minimum value of it's children (3) and return.



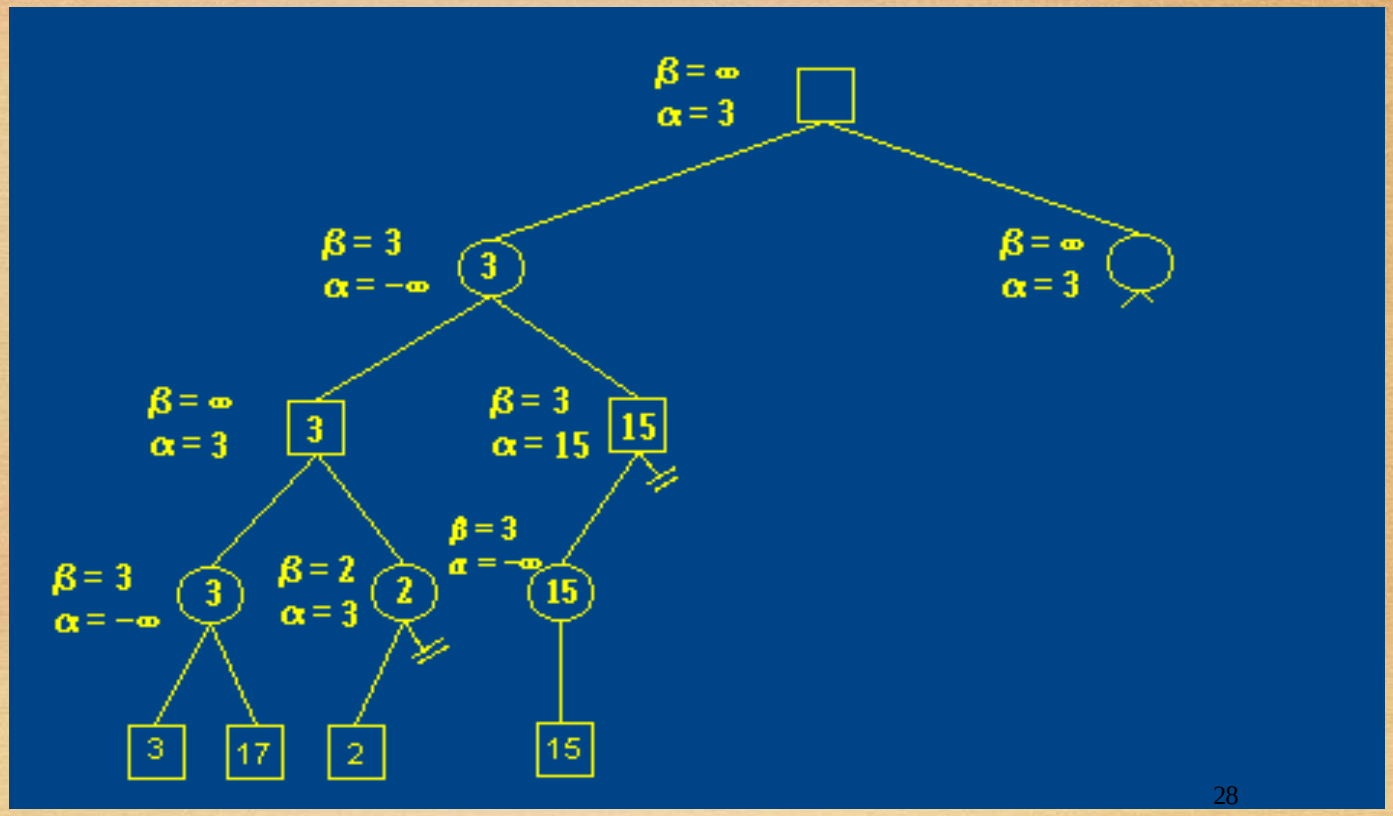
Example

- Finally we've finished with the first child of the root max node. We now know our solution will be at least 3, so we set the alpha value to 3 and go on to the second child.



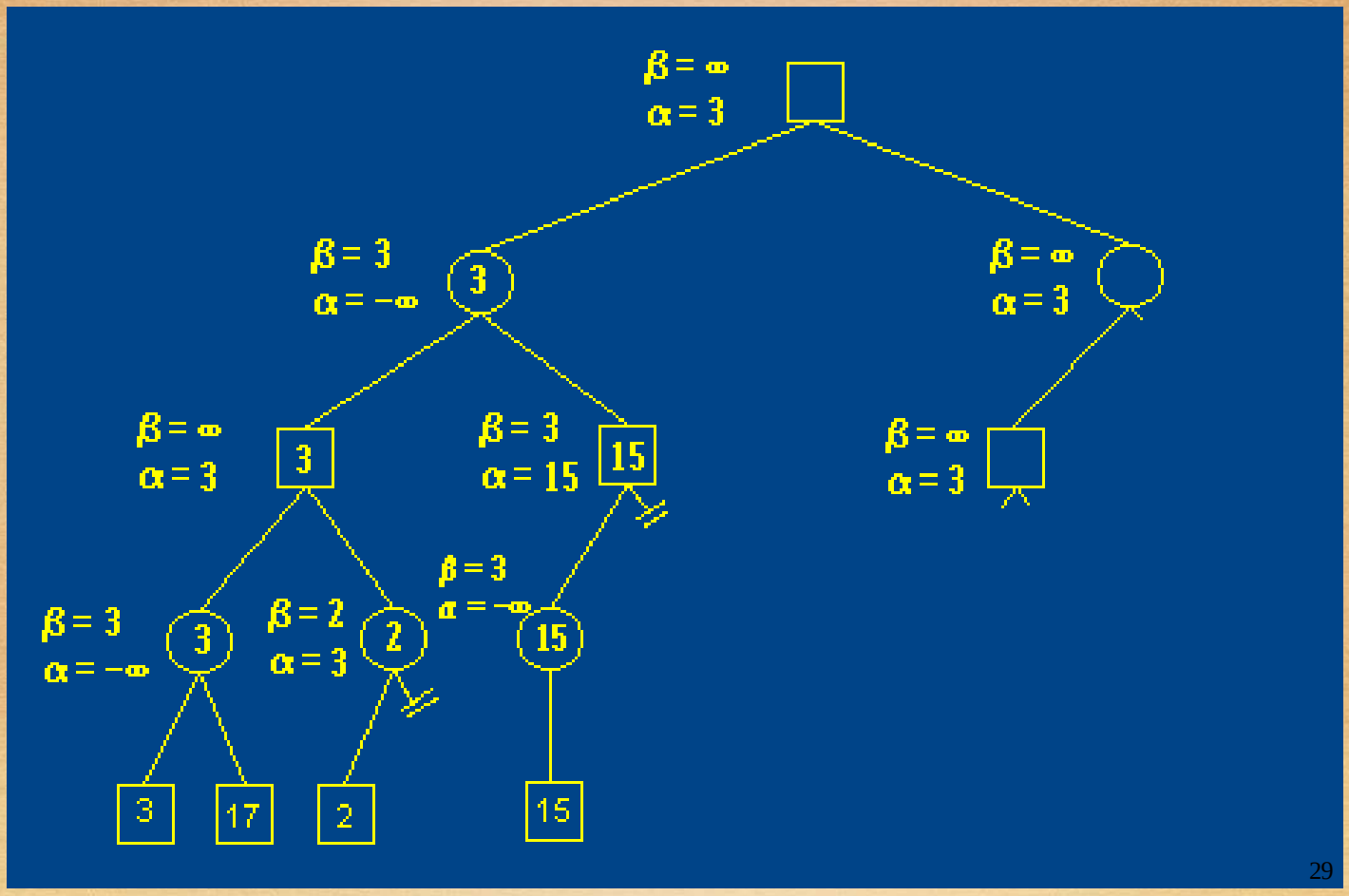
Example

- Passing the alpha and beta values along as we go, we generate the second child of the root node...

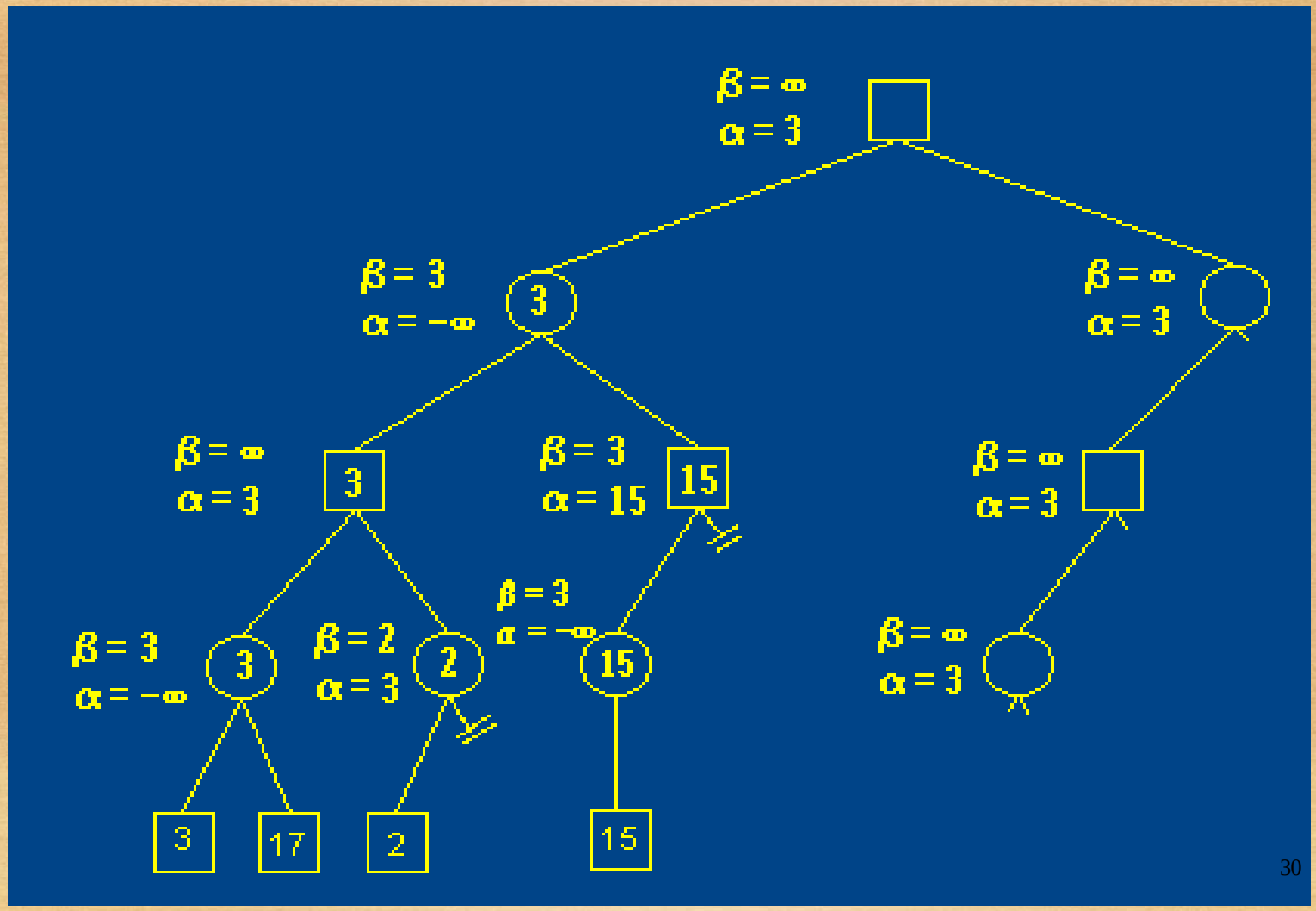


Example

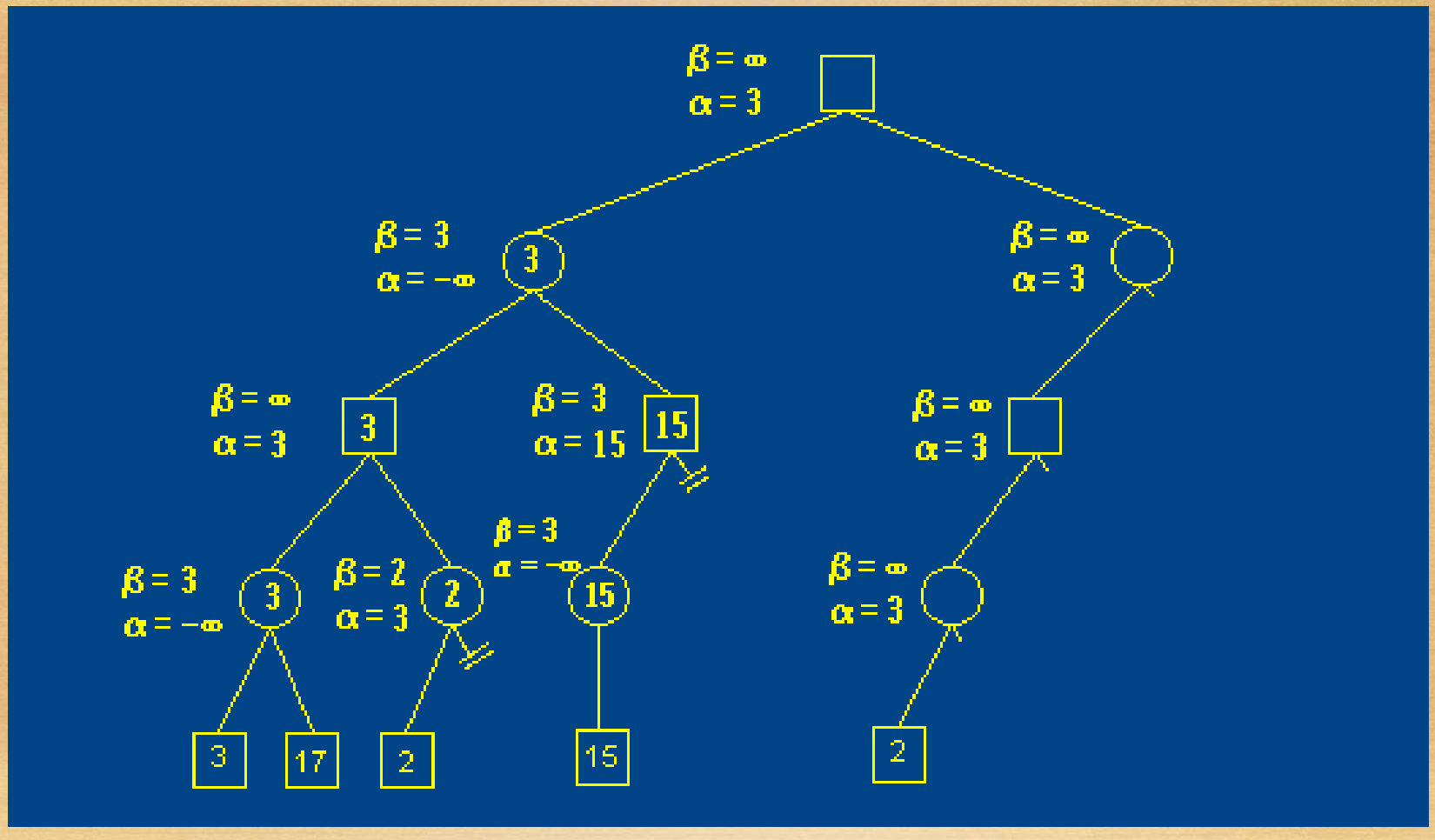
- ... and its first child ...



Example

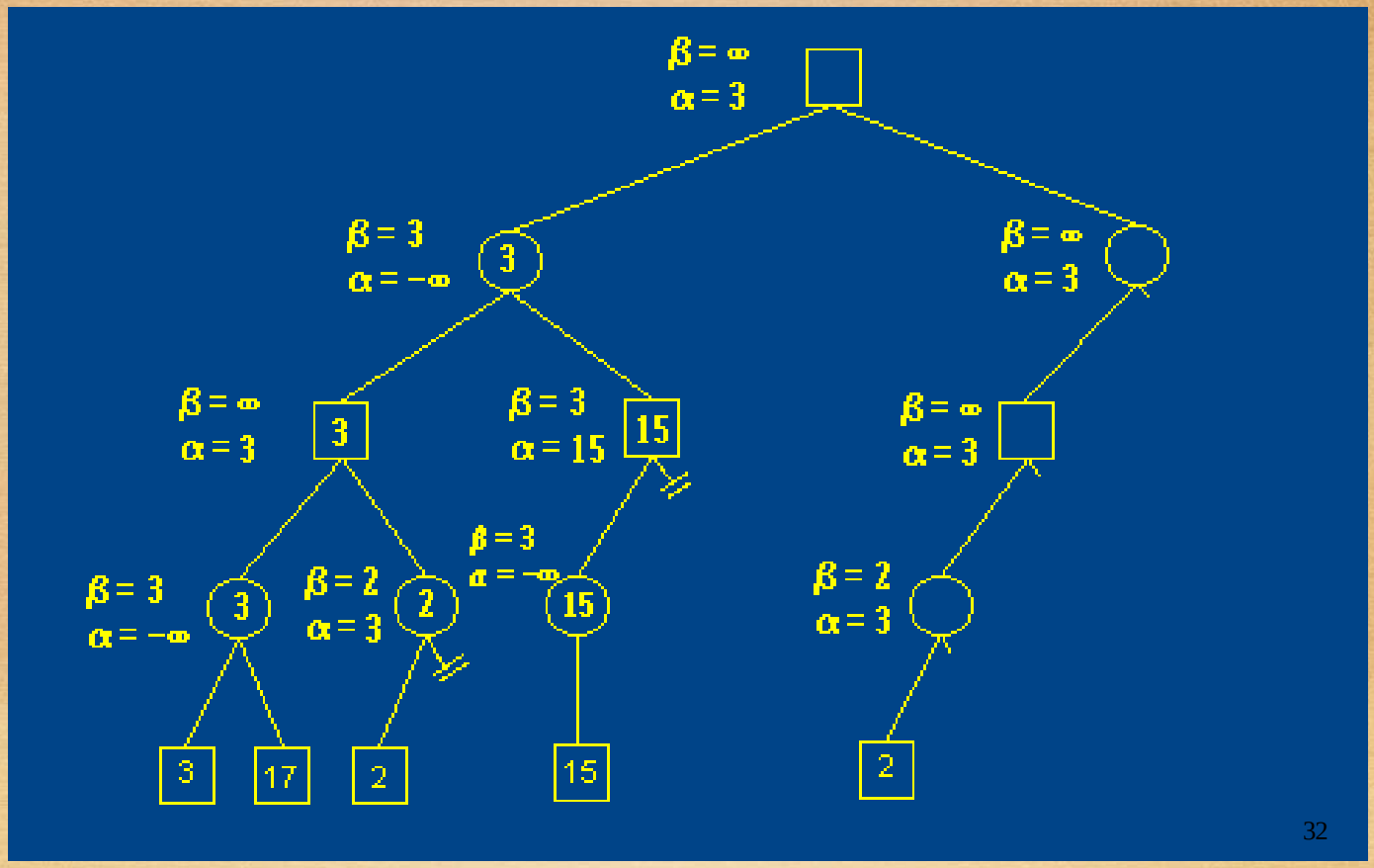


Example



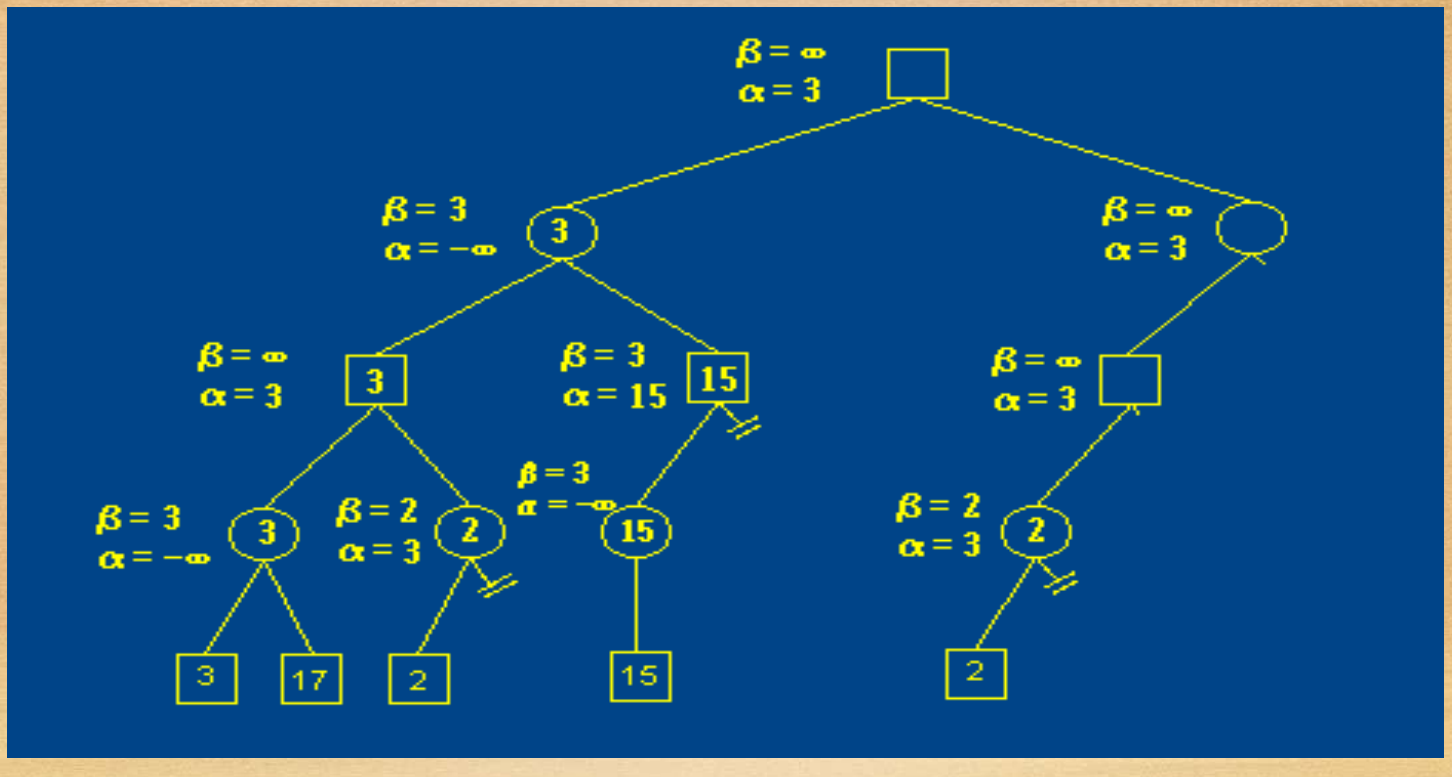
Example

- The min node parent uses this value to set it's beta value to 2:



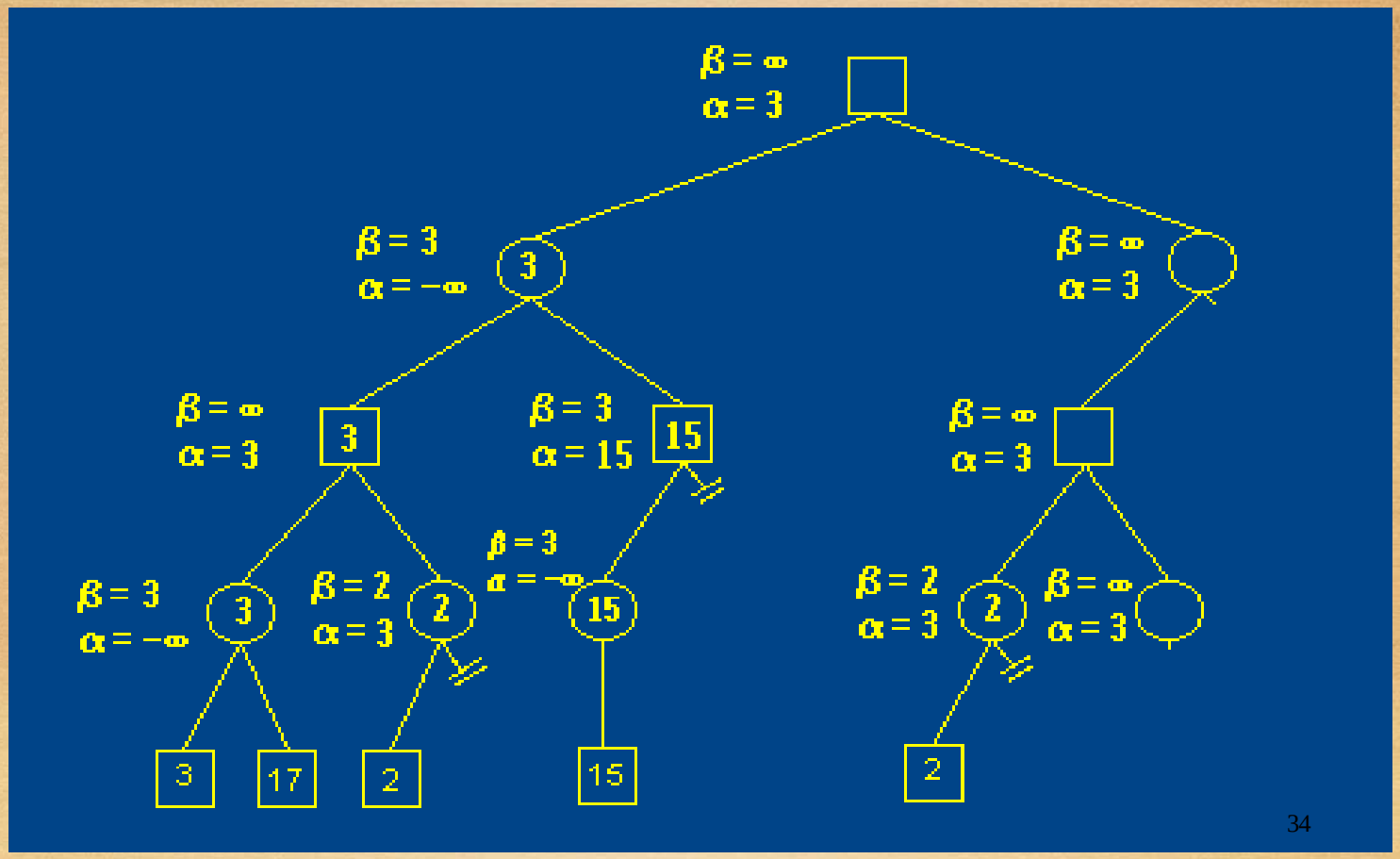
Example

- Once again we are able to prune the other children of this node and return the value that exceeded the bound. Since this value isn't greater than the alpha bound of the parent max node, we don't change the bounds.



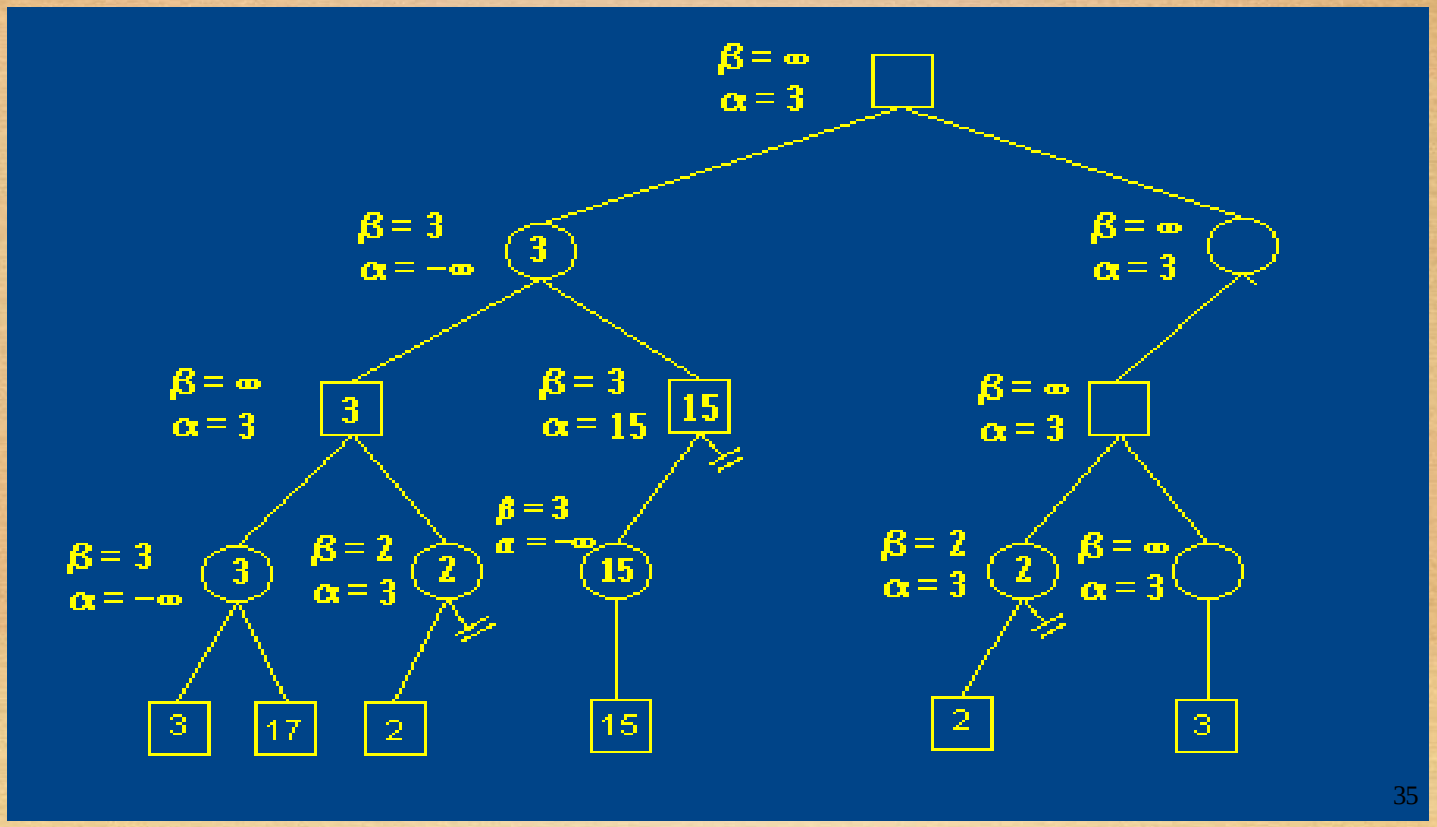
Example

- From here, we generate the next child of the max node:



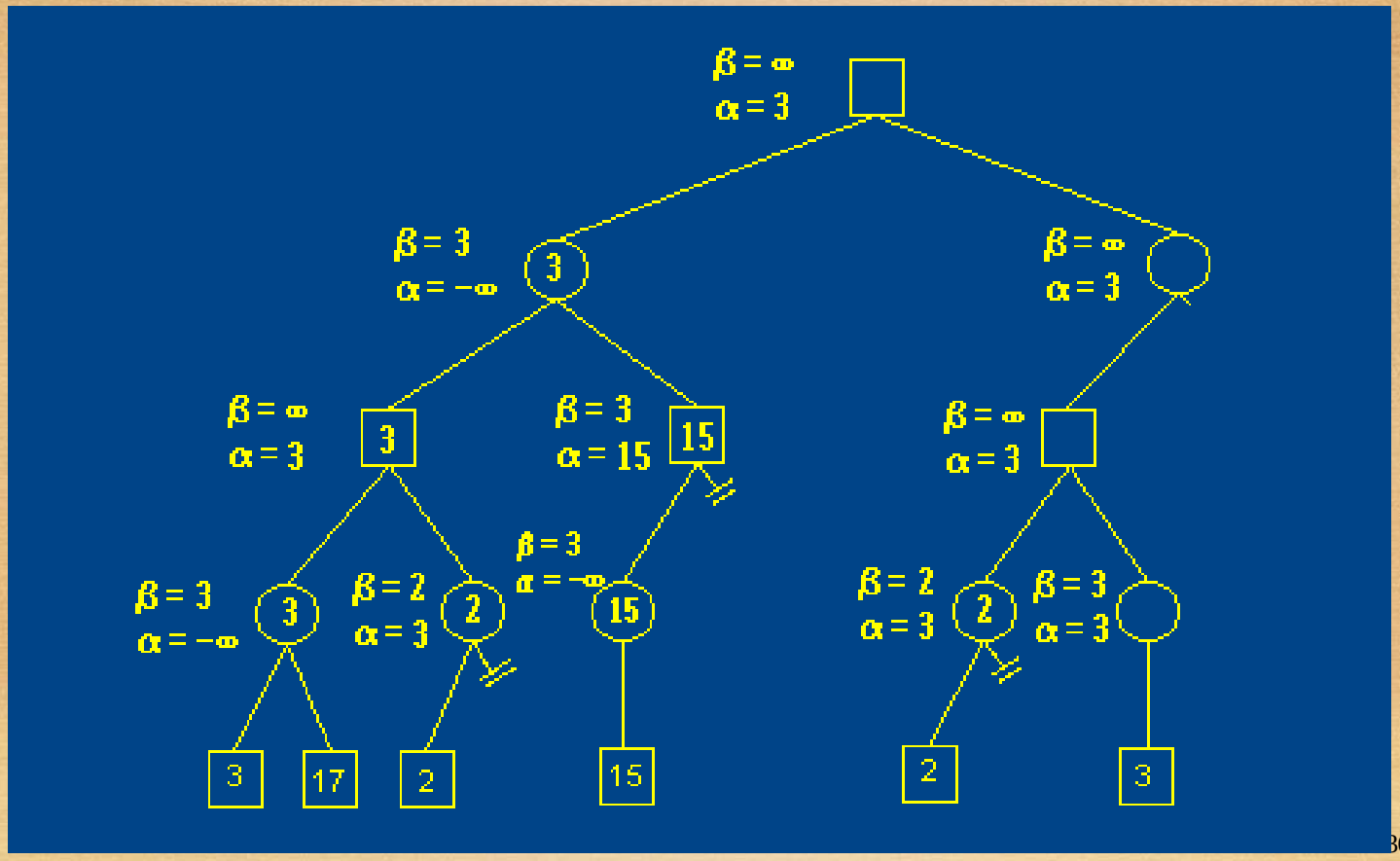
Example

- Then we generate its child, which is at the target depth. We call the evaluation function and get its value of 3.



Example

- The parent min node uses this value to set its upper bound (beta) to 3:

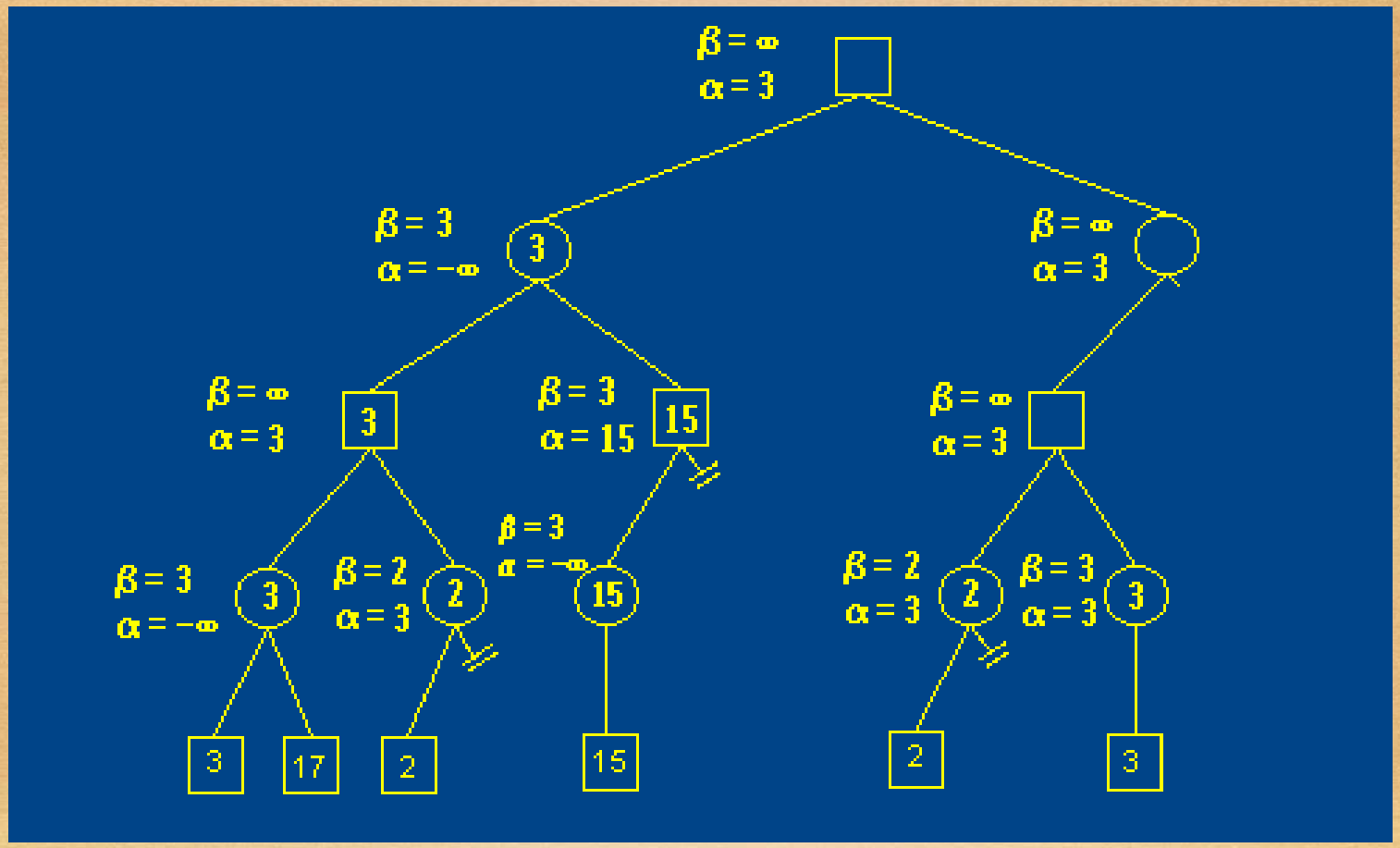


Example

- In other words, at this point alpha = beta. Should we prune here? We haven't actually*exceeded* the bounds, but since alpha and beta are equal, we know we can't really do *better* in this subtree.

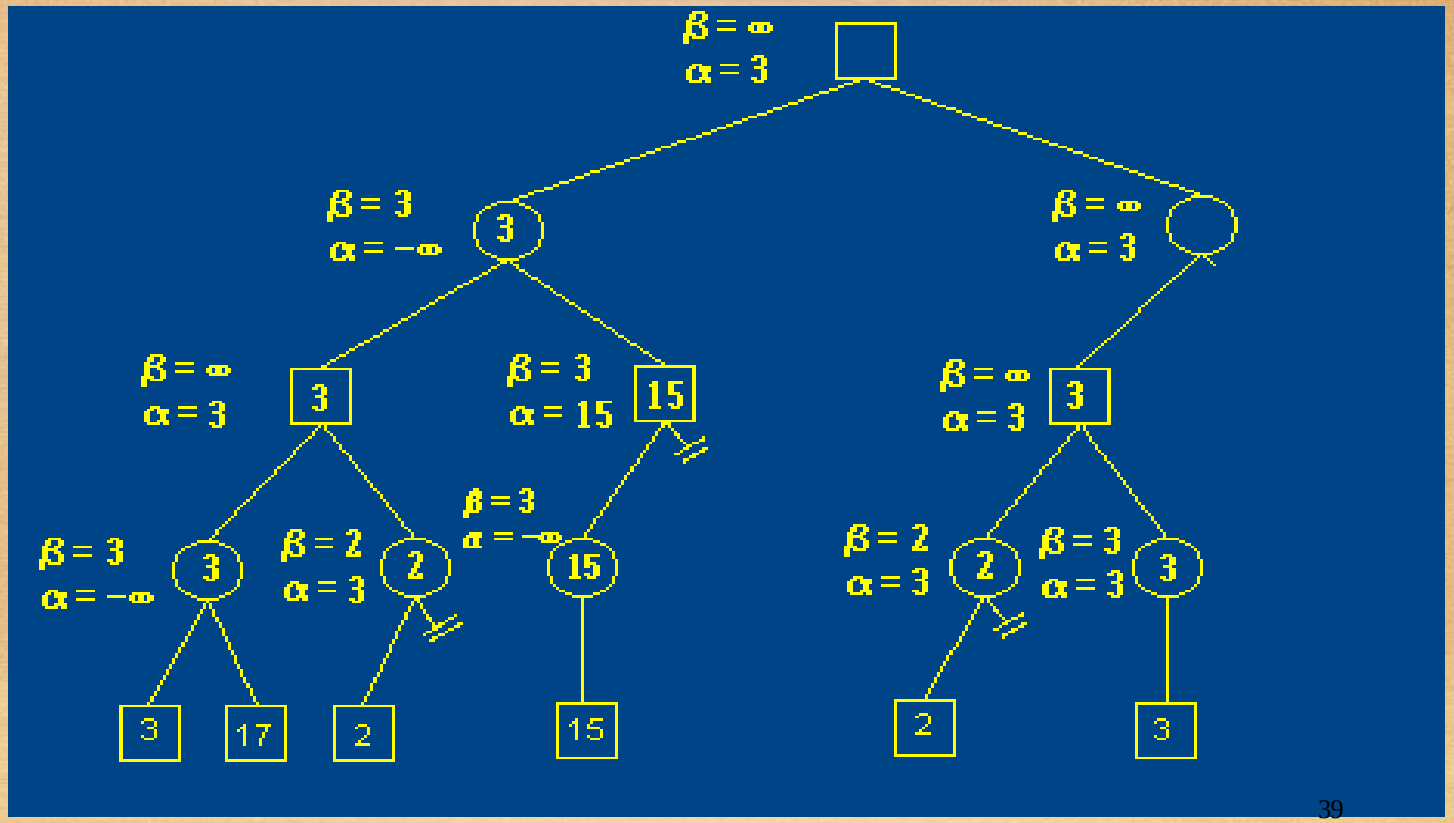
- The answer is yes, we should prune. The reason is that even though we can't do better, we might be able to do worse. Remember, the task of minimax is to find the best move to make at the state represented by the top level max node. As it happens we've finished with this node's children anyway, so we return the min value 3.

Example


β = ∞
α = 3
β = 3
α = −∞
3
β = ∞
α = 3
β = ∞
α = 3
3
β = 3
α = 15
15
β = ∞
α = 3
β = 3
α = −∞
3
β = 2
α = 3
2
β = 3
α = −∞
15
β = 2
α = 3
2
β = 3
α = 3
3
3
17
2
15
2
3

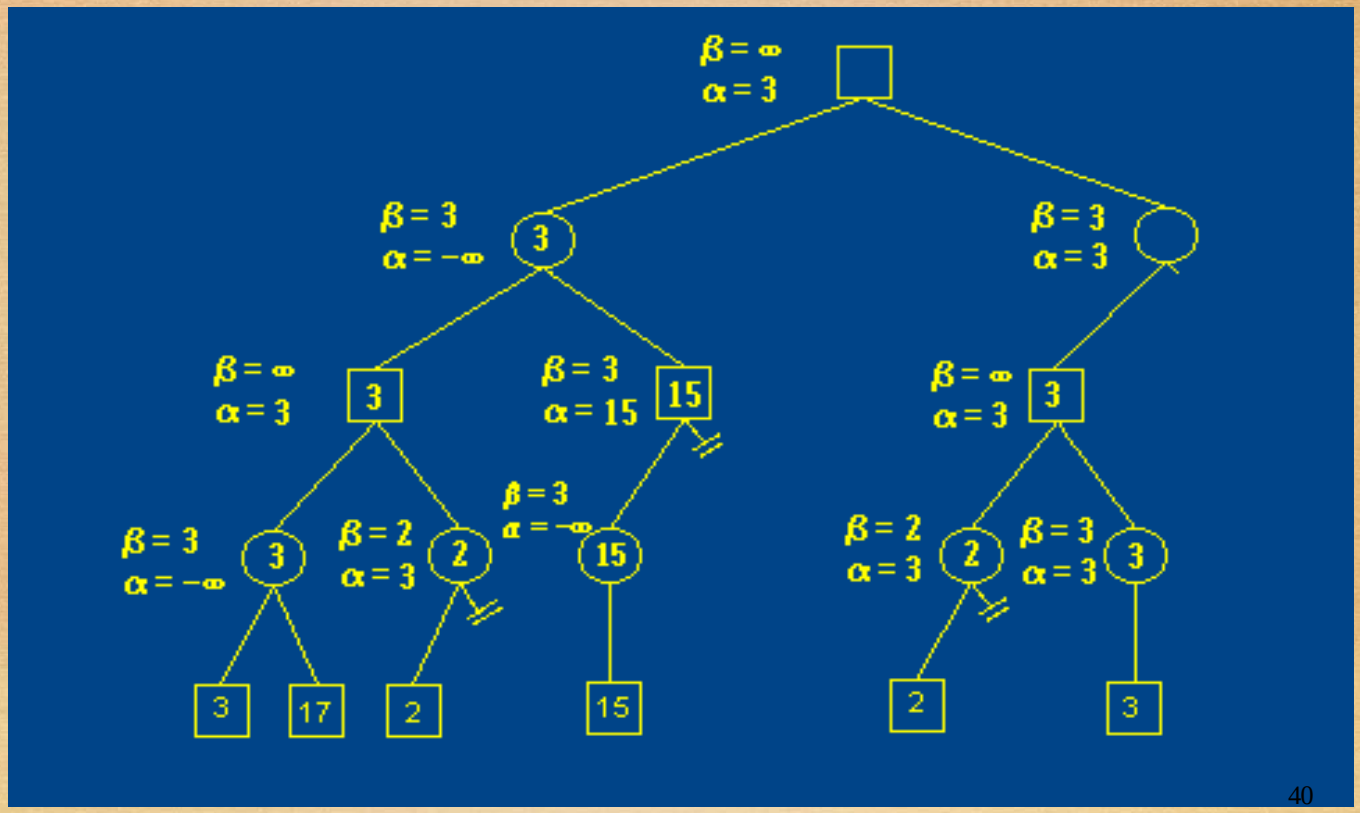
Example

- The max node above has now seen all of its children, so it returns the maximum value of those it has seen, which is 3.



Example

- This value is returned to its parent min node, which then has a new upper bound of 3, so it sets beta to 3:



- Once again, we are at a point where alpha and beta are tied, so we prune.
- If you were to run minimax on the list version presented at the start of the example, your minimax would return a value of 3 and 6 terminal nodes would have been examined.

Conclusion

- Pruning will not affect final results.
- Whole subtrees can be pruned, not just leaves.
- Better move *ordering* improves effectiveness of pruning.
- With *perfect ordering*, time complexity is $O(b^{m/2})$.
  - Effective branching factor of sqrt(b)
  - Consequence: alpha-beta pruning can look twice as deep as minimax in the same amount of time.