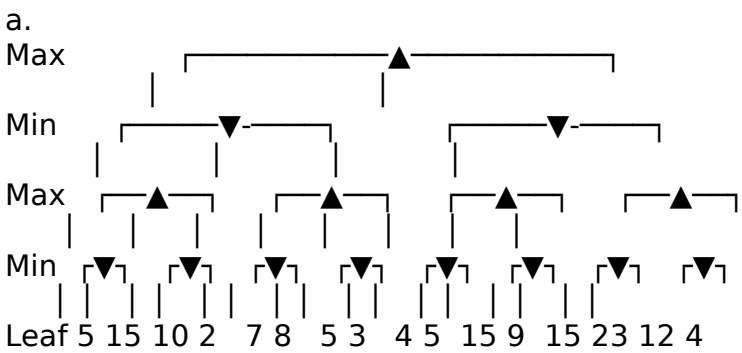
Adversarial Search

Chapter 5

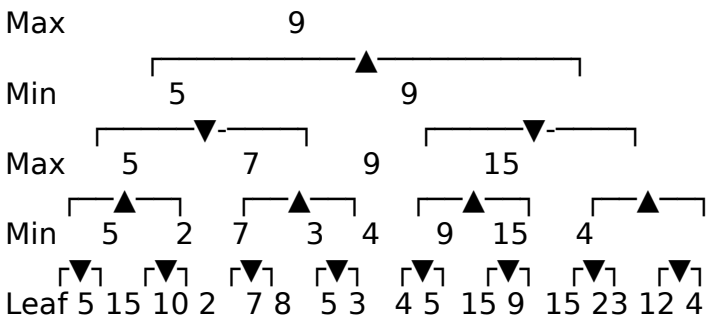
A tree with branching factor 2 and depth 4, has the following values for its leaves: 5, 15, 10, 2, 7, 8, 5, 3, 4, 5, 15, 9, 15, 23, 12, 4

a. Draw the described tree.
b. Find the MinMax value of the tree.
c. Apply Alpha-Beta pruning showing the nodes that will not be evaluated.

a.



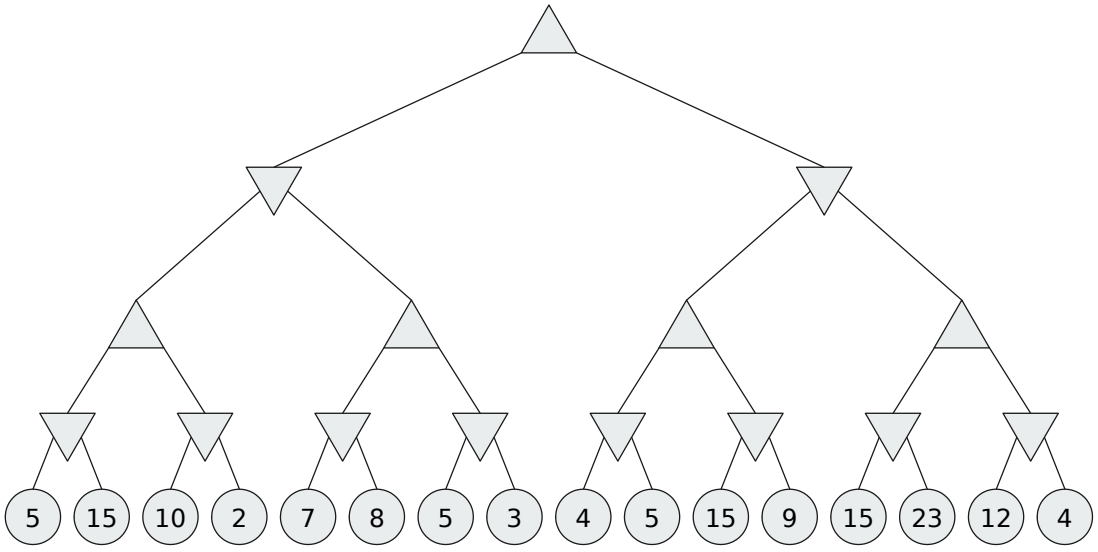
b.



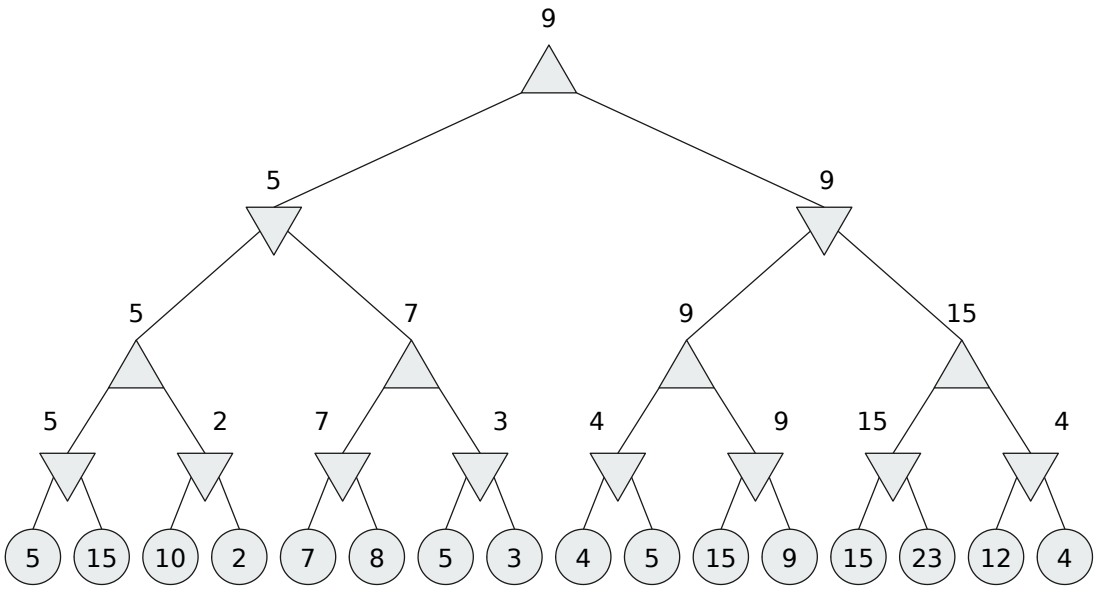
Minimax Algorithm:

```
Function max_search(state):
        If is_terminal(state): return value_of(state)
        value = -∞
        For action in get_actions(state, max_player):
                child = get_successor(state, max_player, action)
```

a.



b.



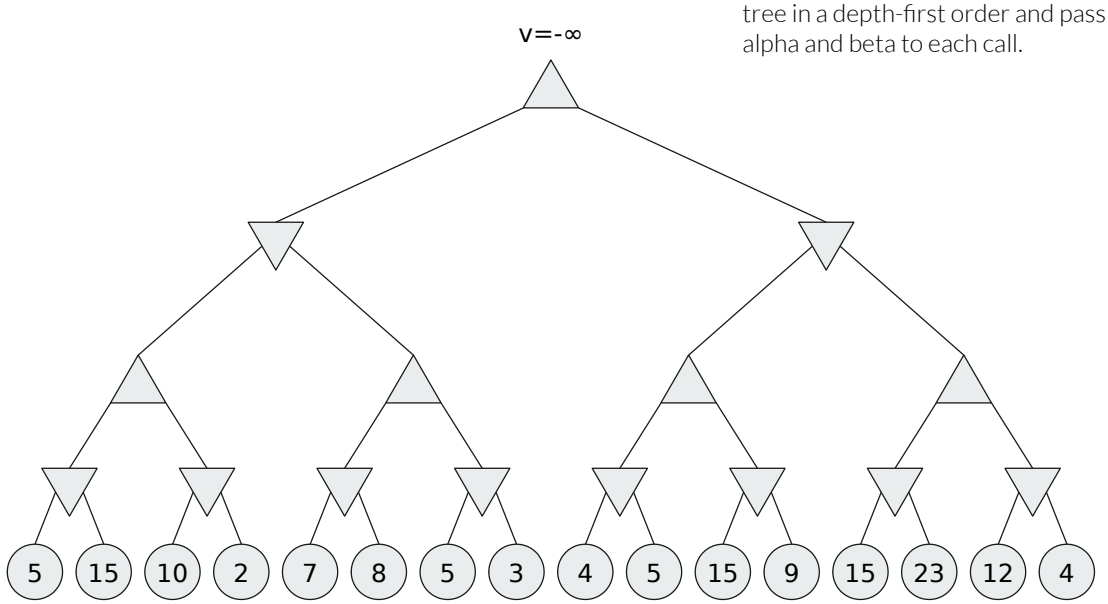
```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

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

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

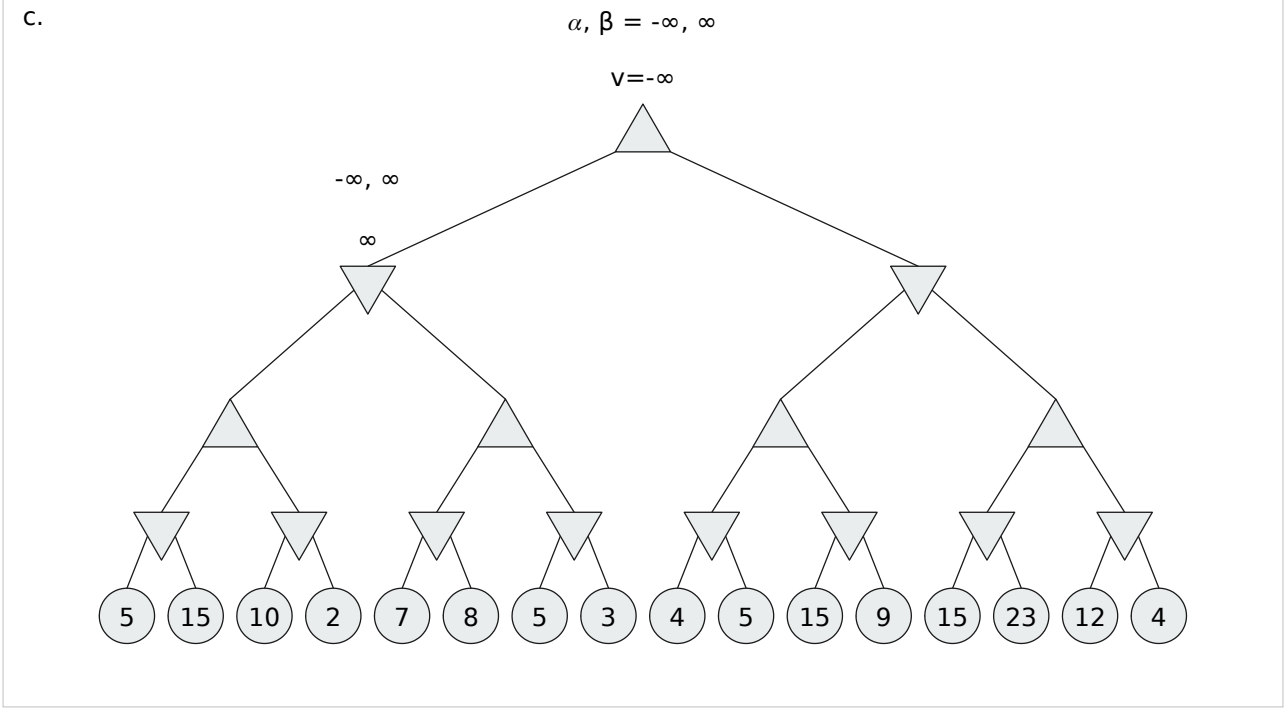
c.

$\alpha, \beta = -\infty, \infty$

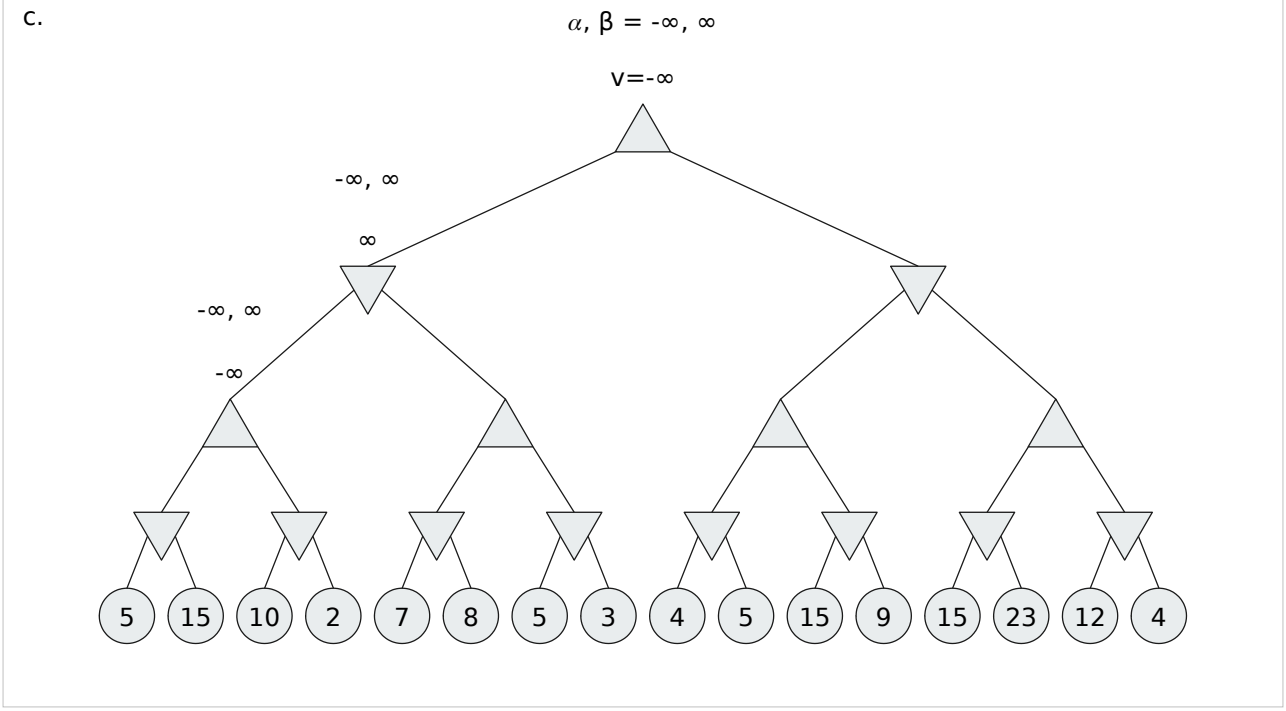
Starting from the root, we traverse the tree in a depth-first order and pass the alpha and beta to each call.



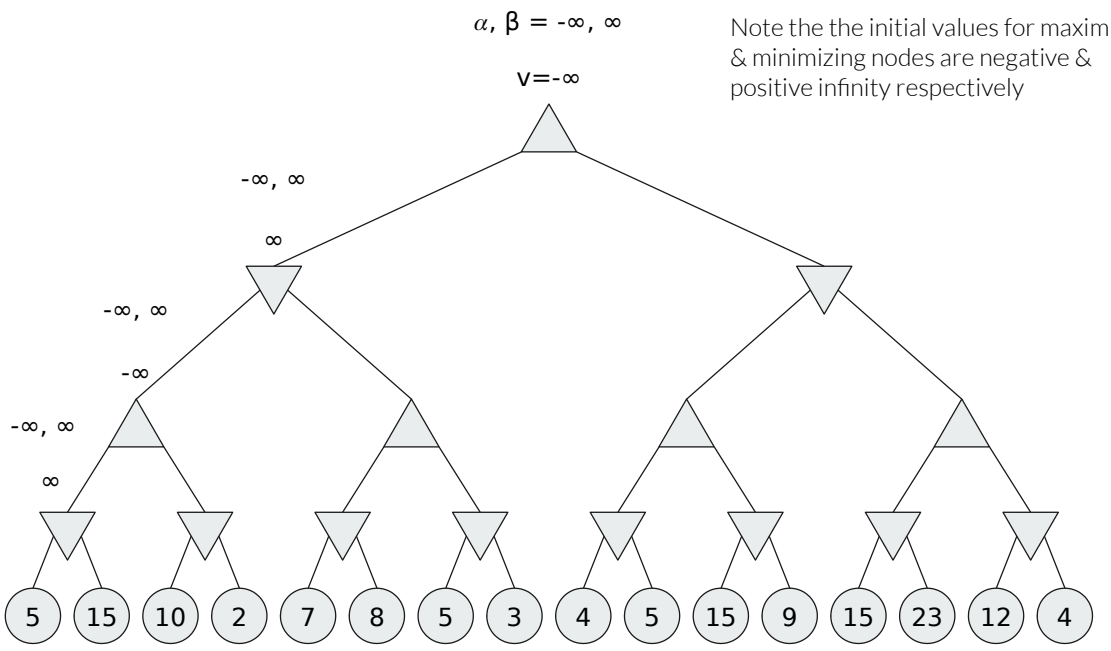
c.



c.

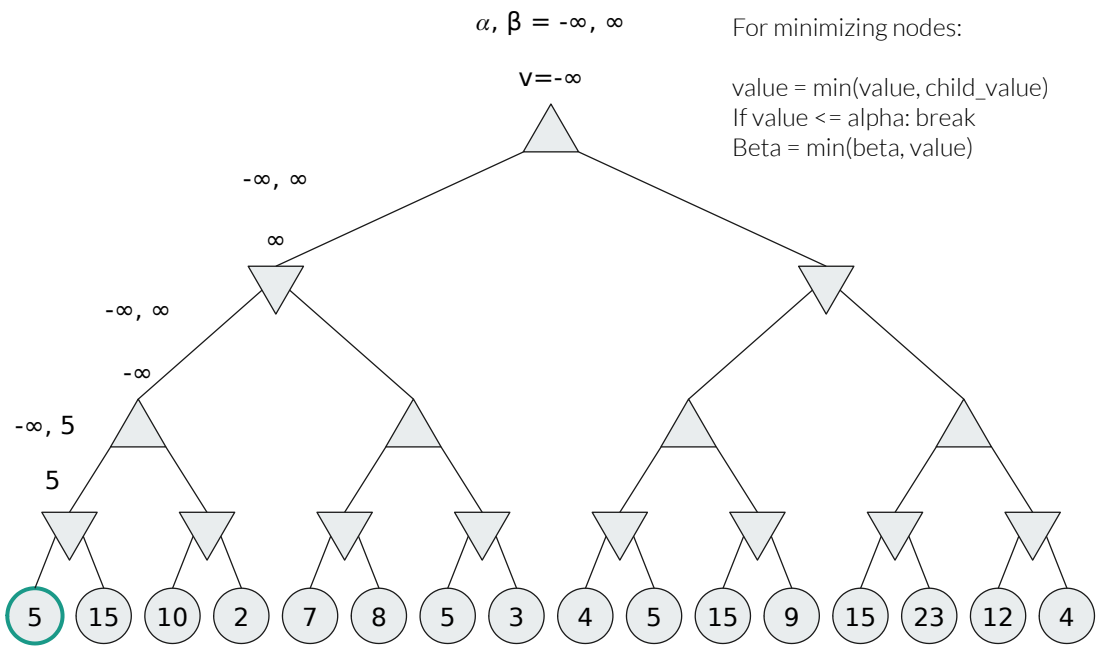


c.



Note the the initial values for maximizing & minimizing nodes are negative & positive infinity respectively

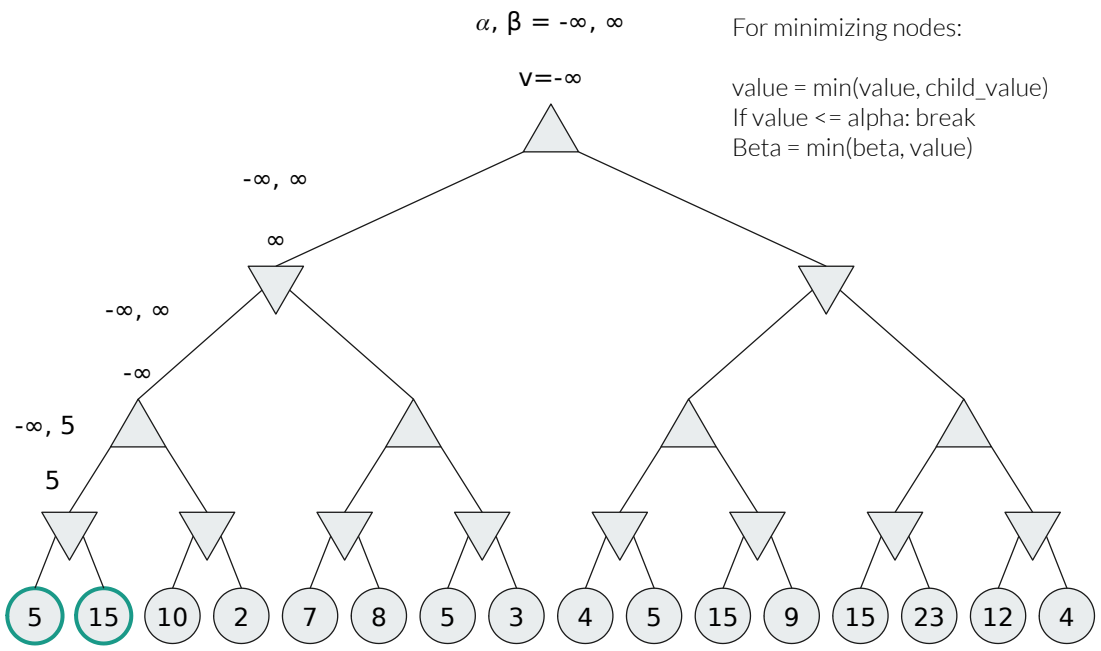
c.



For minimizing nodes:

value = min(value, child_value)
If value <= alpha: break
Beta = min(beta, value)

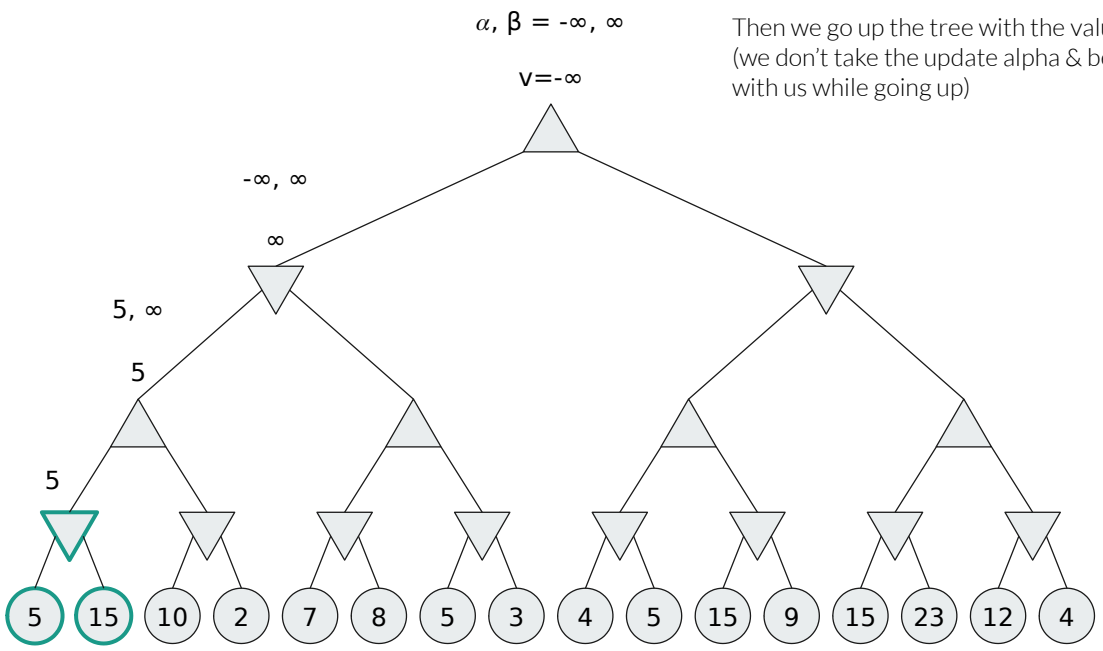
c.



For minimizing nodes:

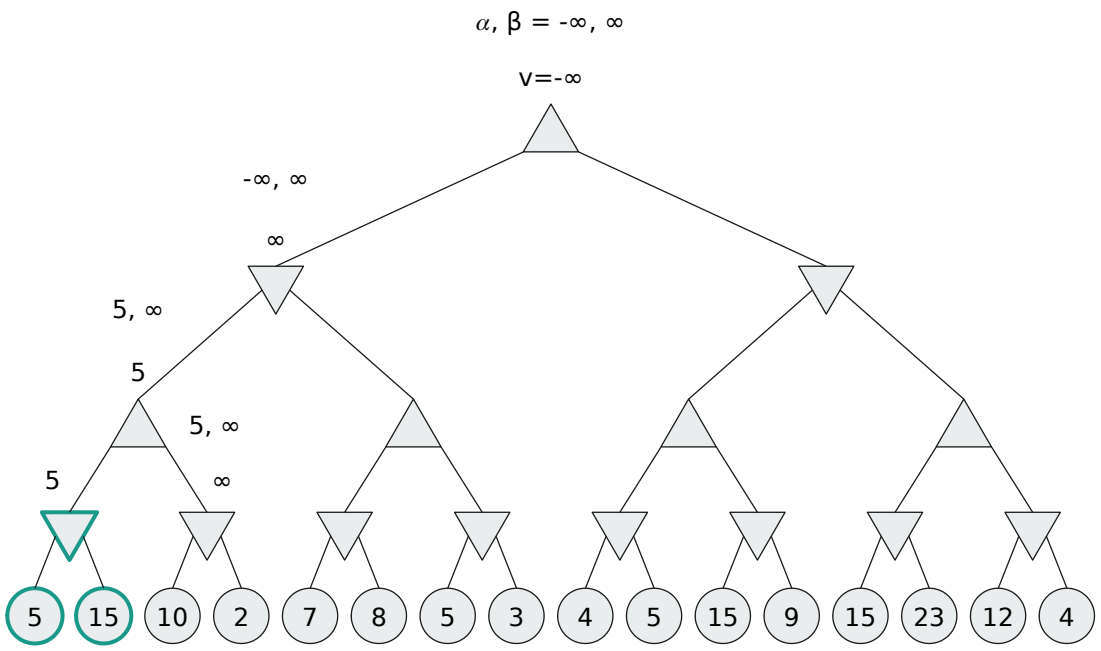
value = min(value, child_value)
If value <= alpha: break
Beta = min(beta, value)

c.

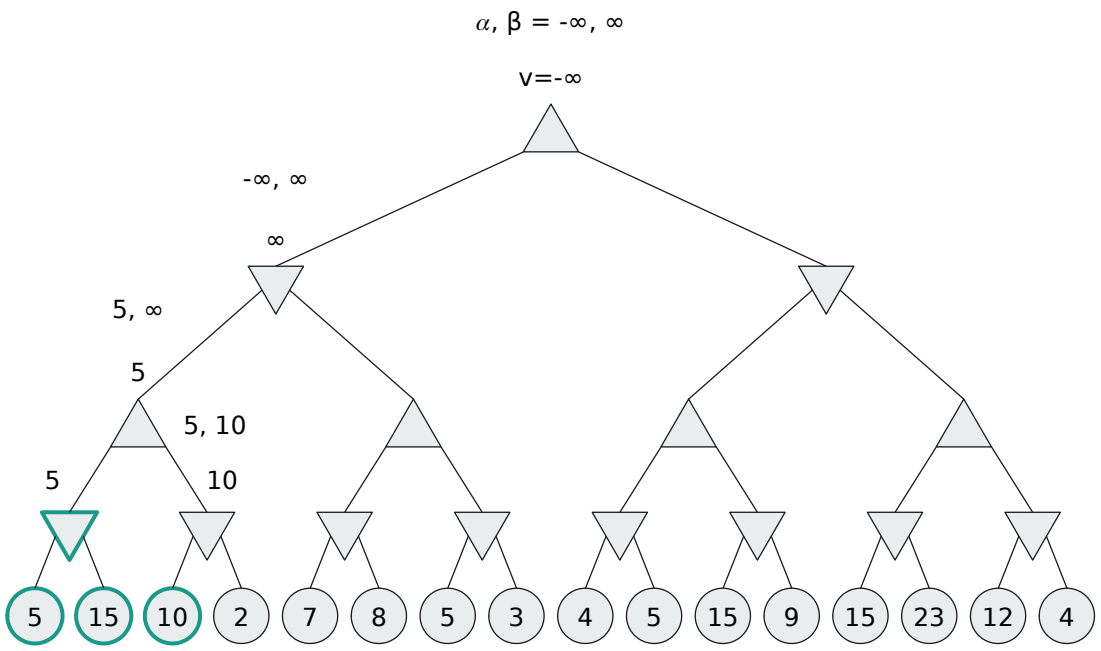


Then we go up the tree with the value (we don't take the update alpha & beta with us while going up)

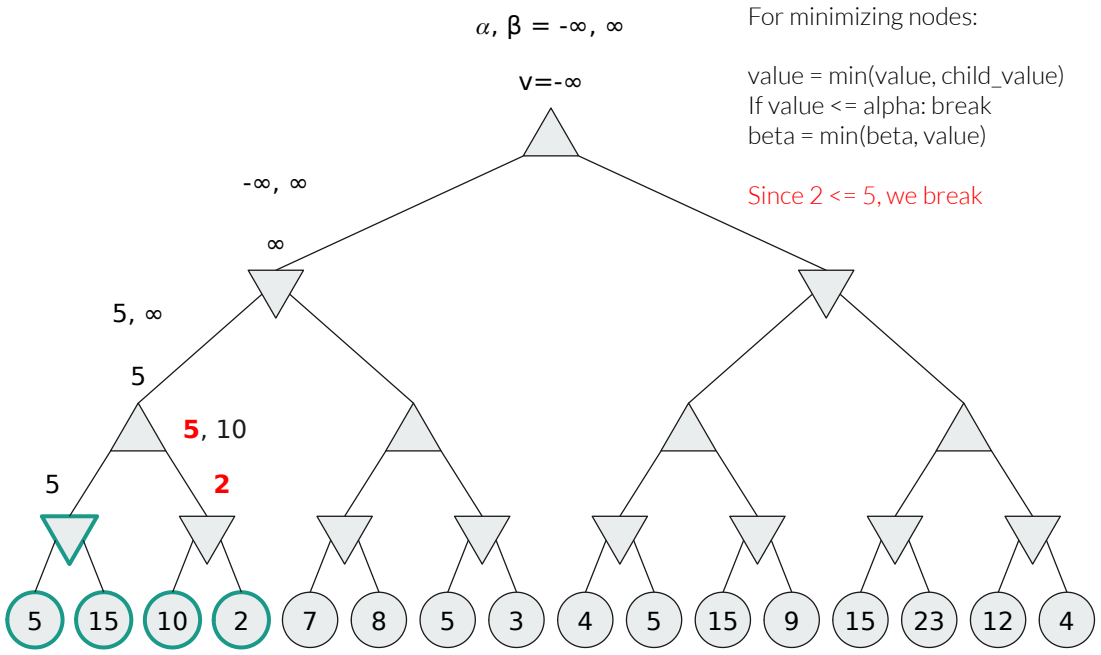
c.



c.



c.

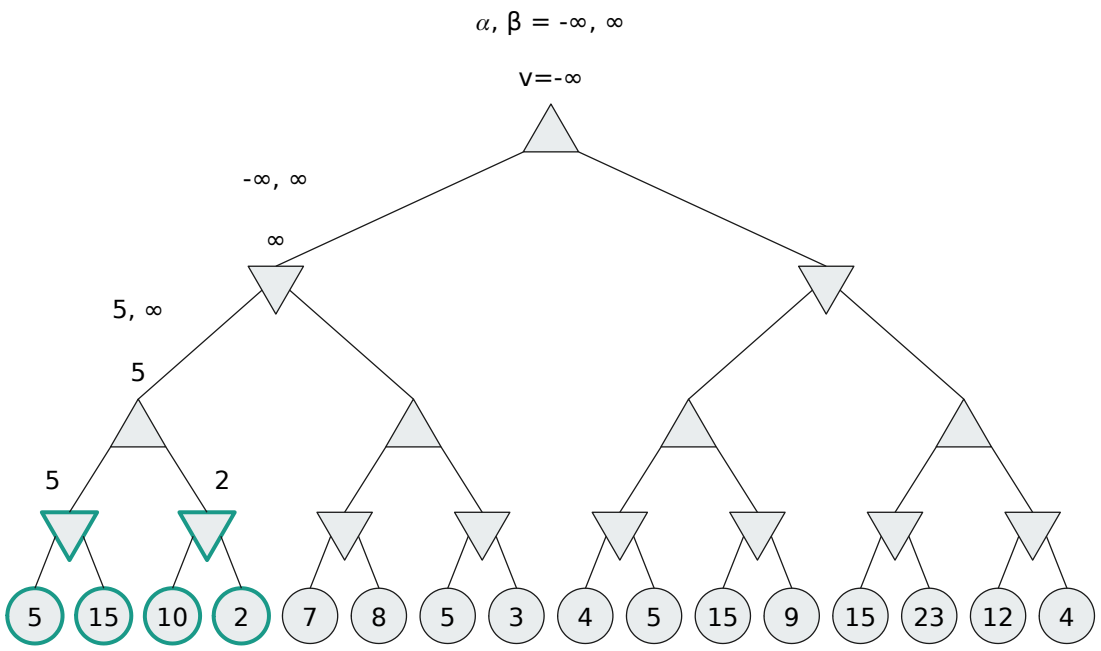


For minimizing nodes:

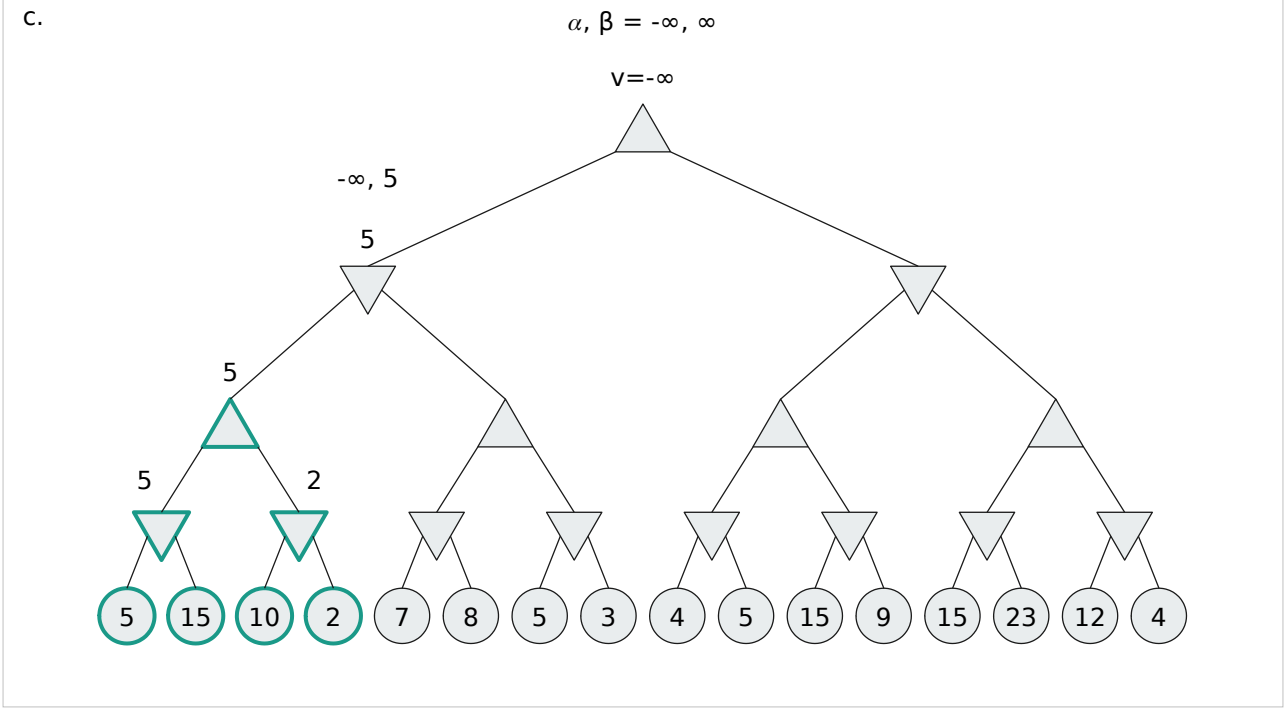
value = min(value, child_value)
If value <= alpha: break
beta = min(beta, value)

Since 2 <= 5, we break

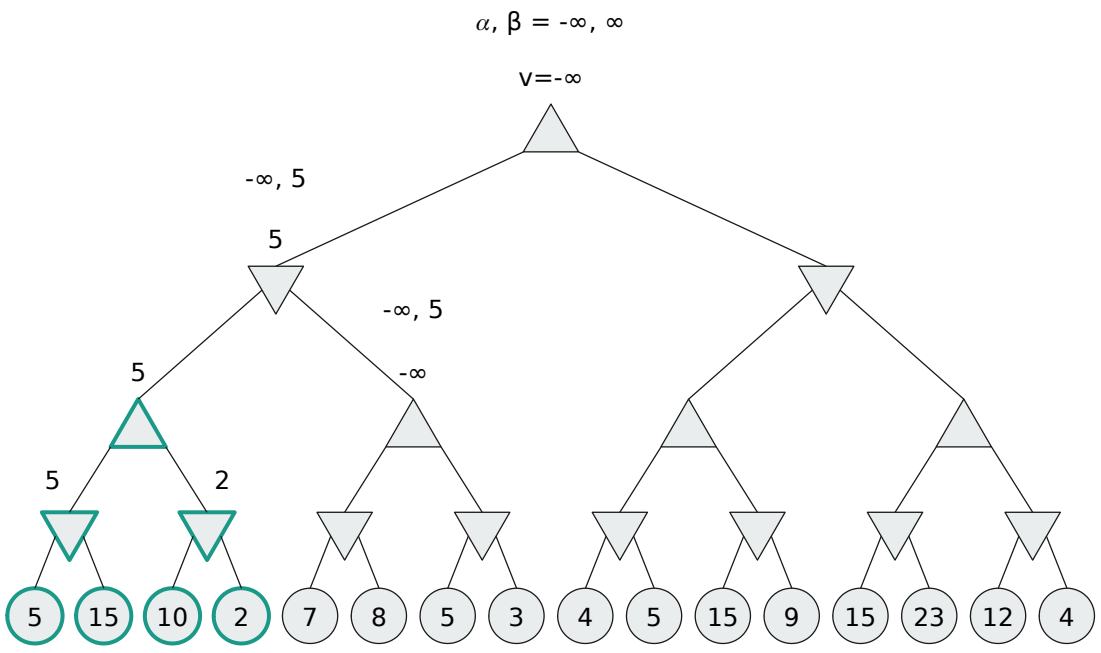
c.



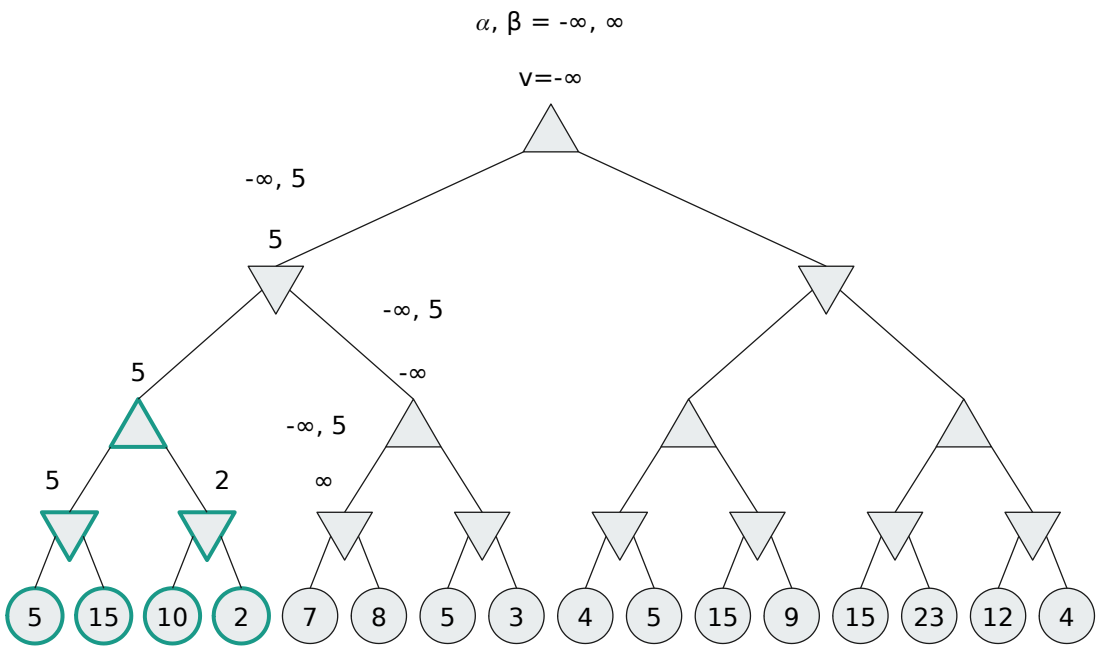
c.



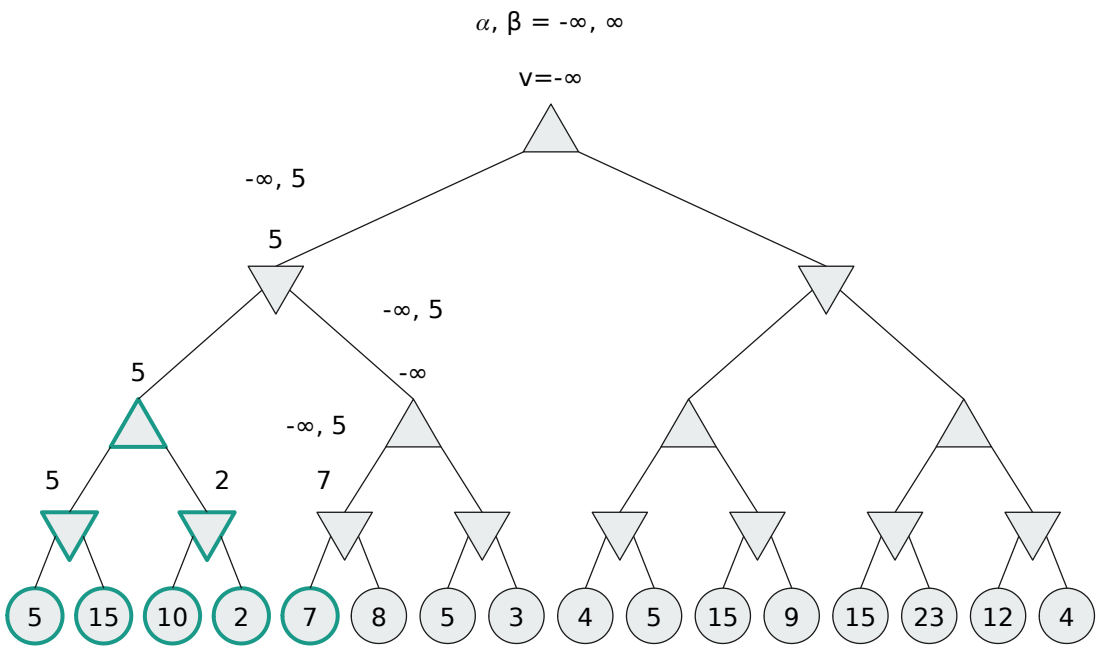
c.



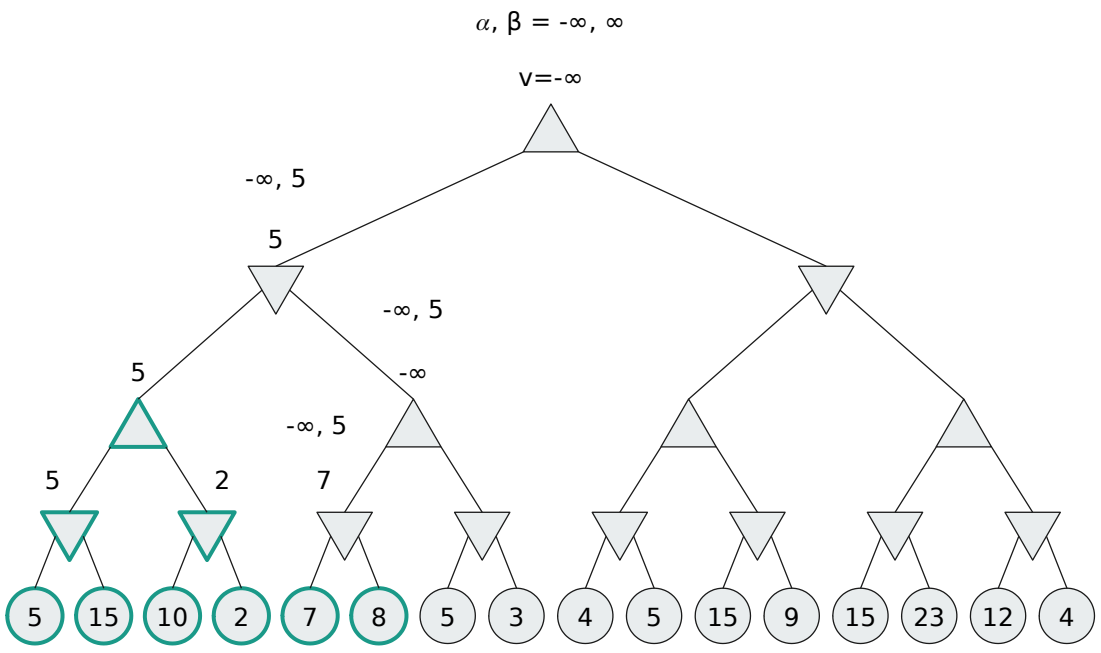
c.



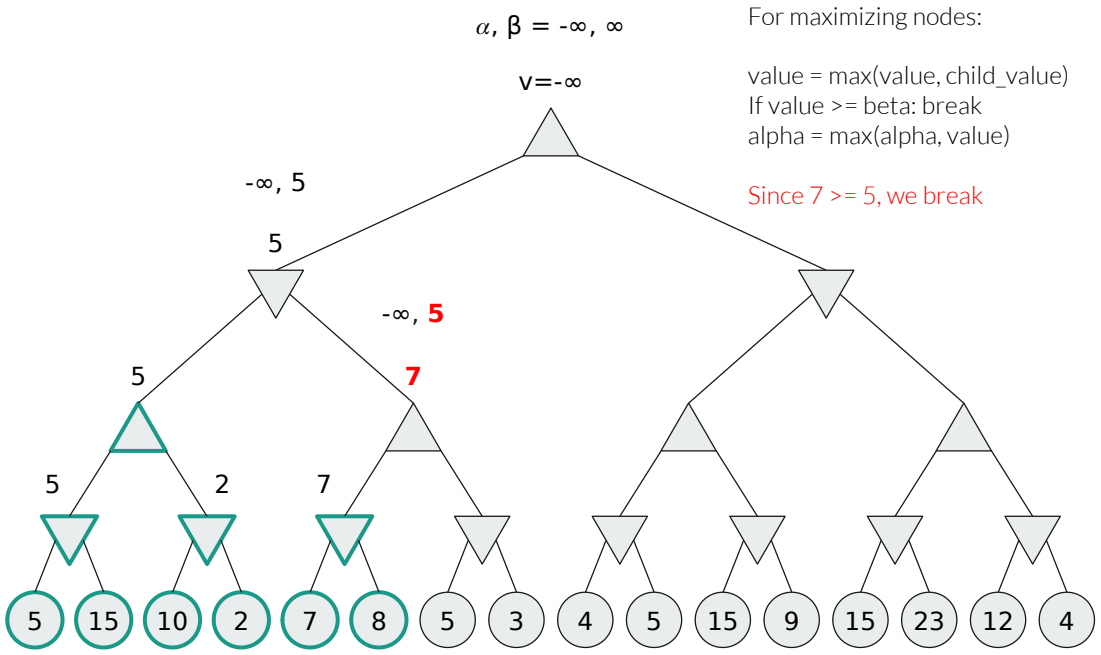
c.



c.



c.

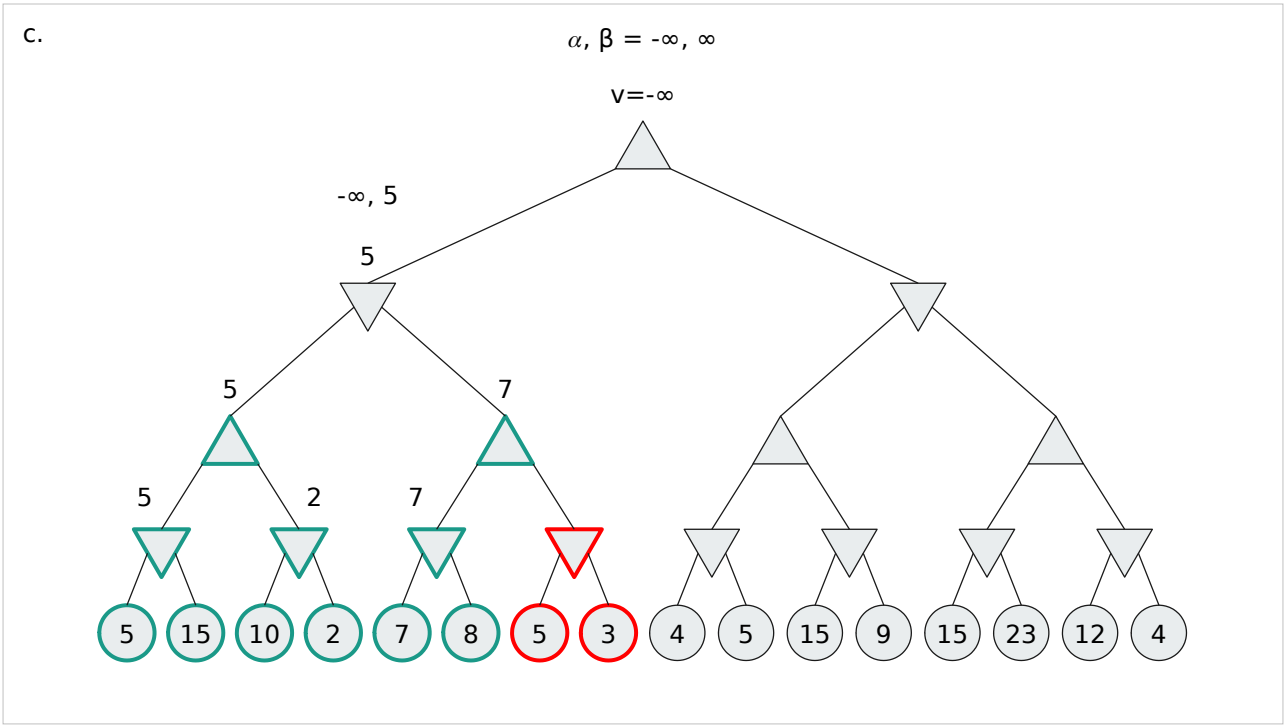


For maximizing nodes:

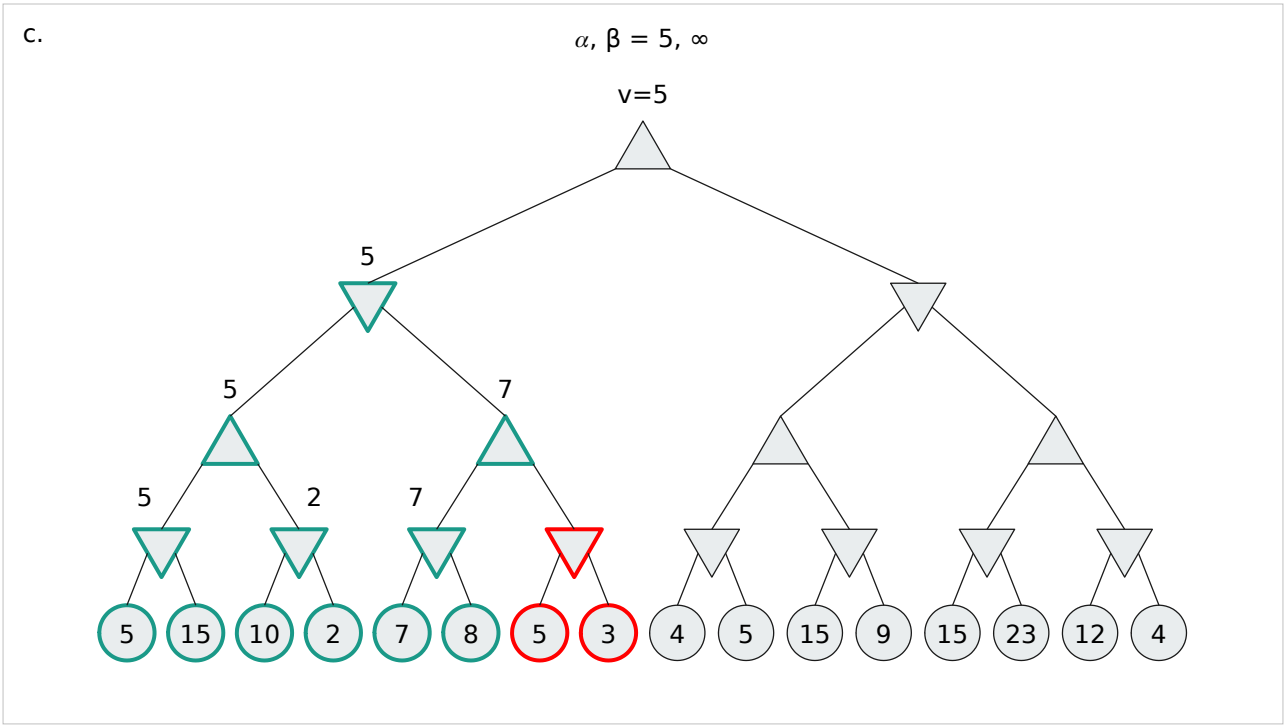
value = max(value, child_value)
If value >= beta: break
alpha = max(alpha, value)

Since 7 >= 5, we break

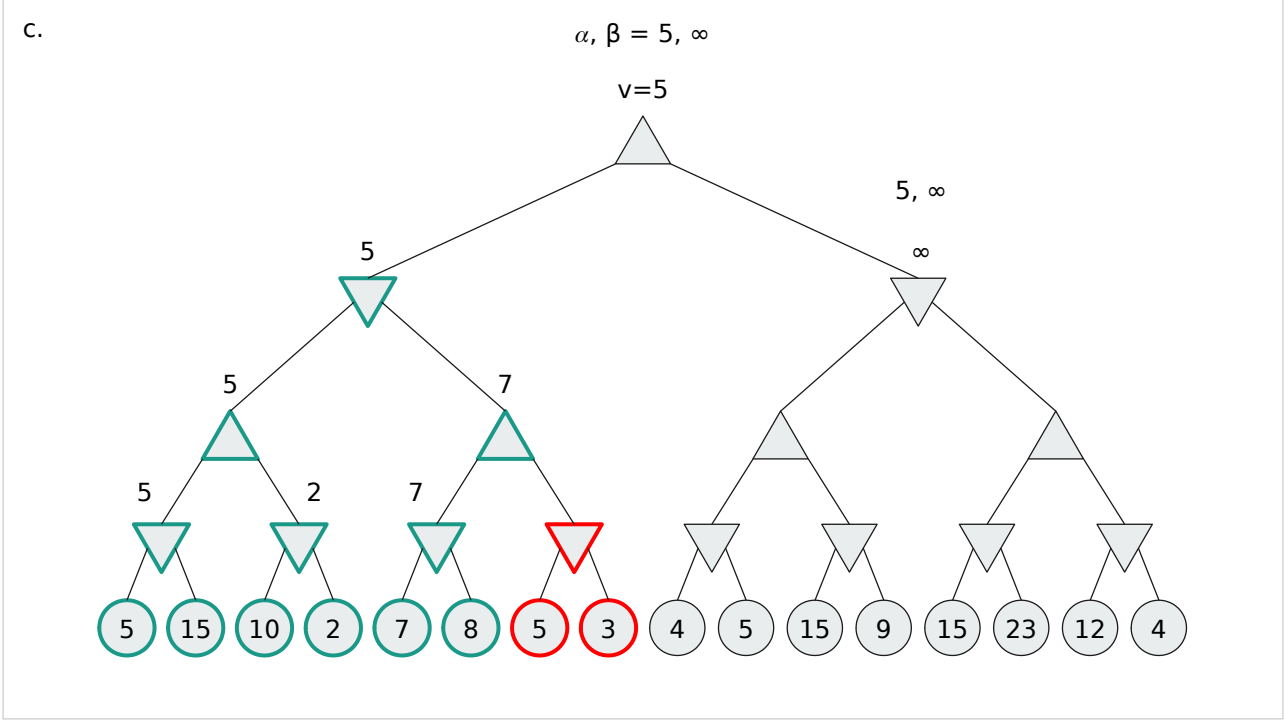
c.



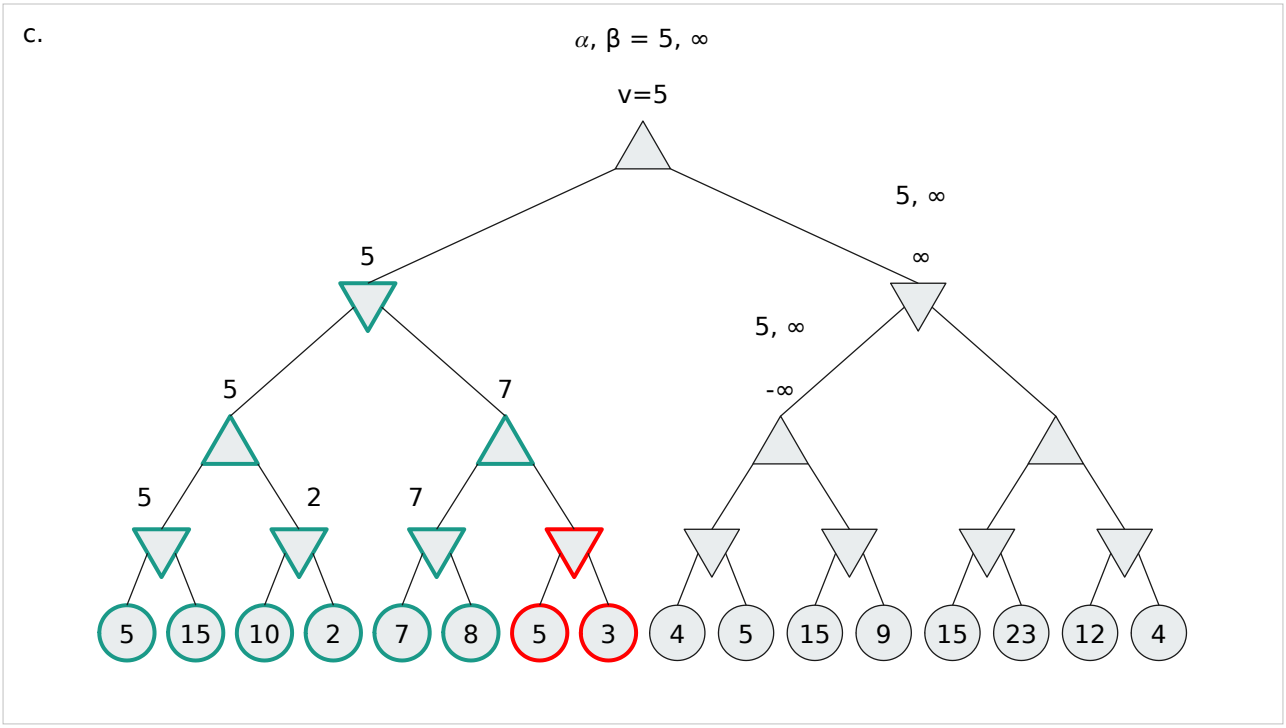
c.



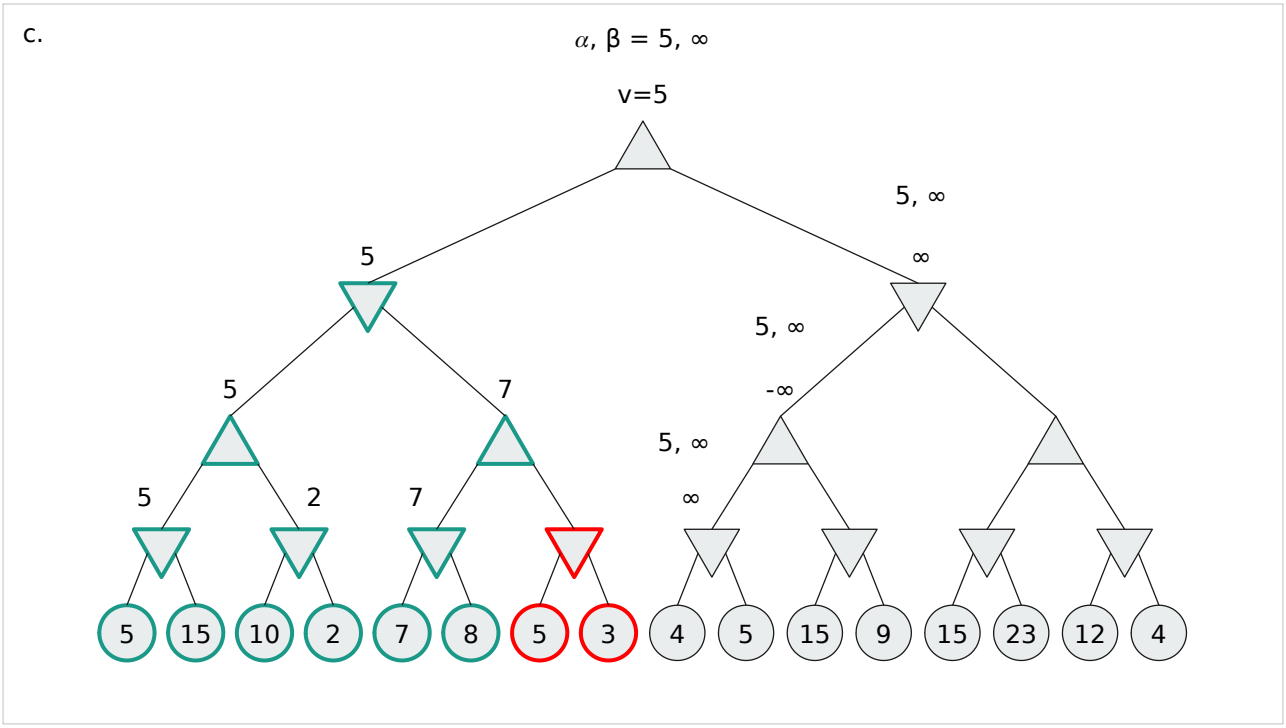
c.



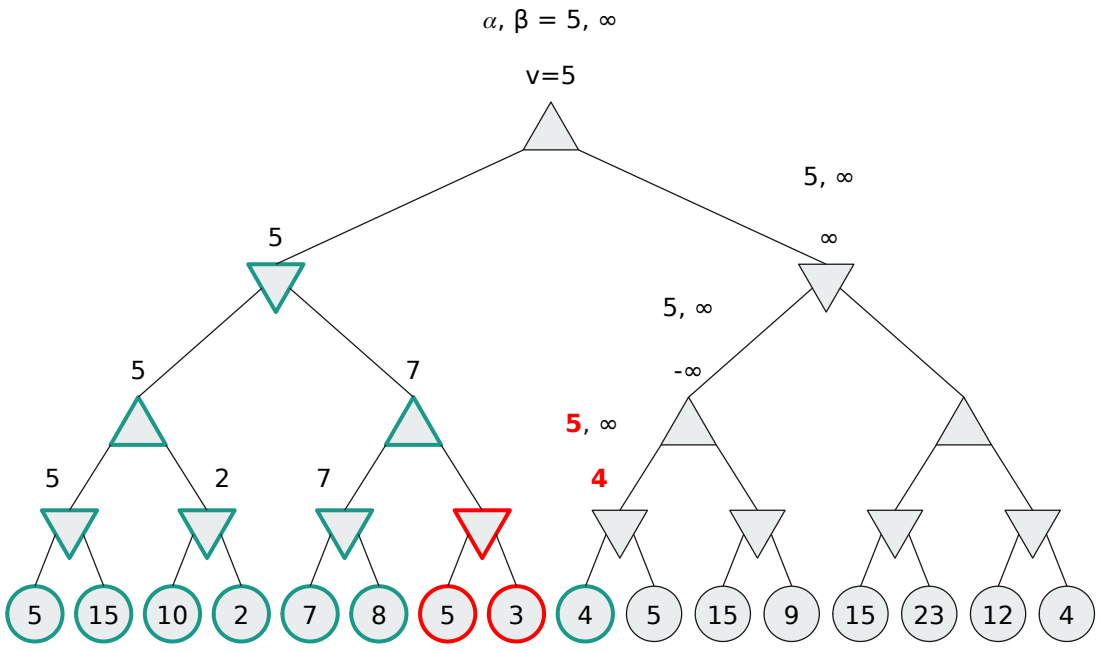
c.



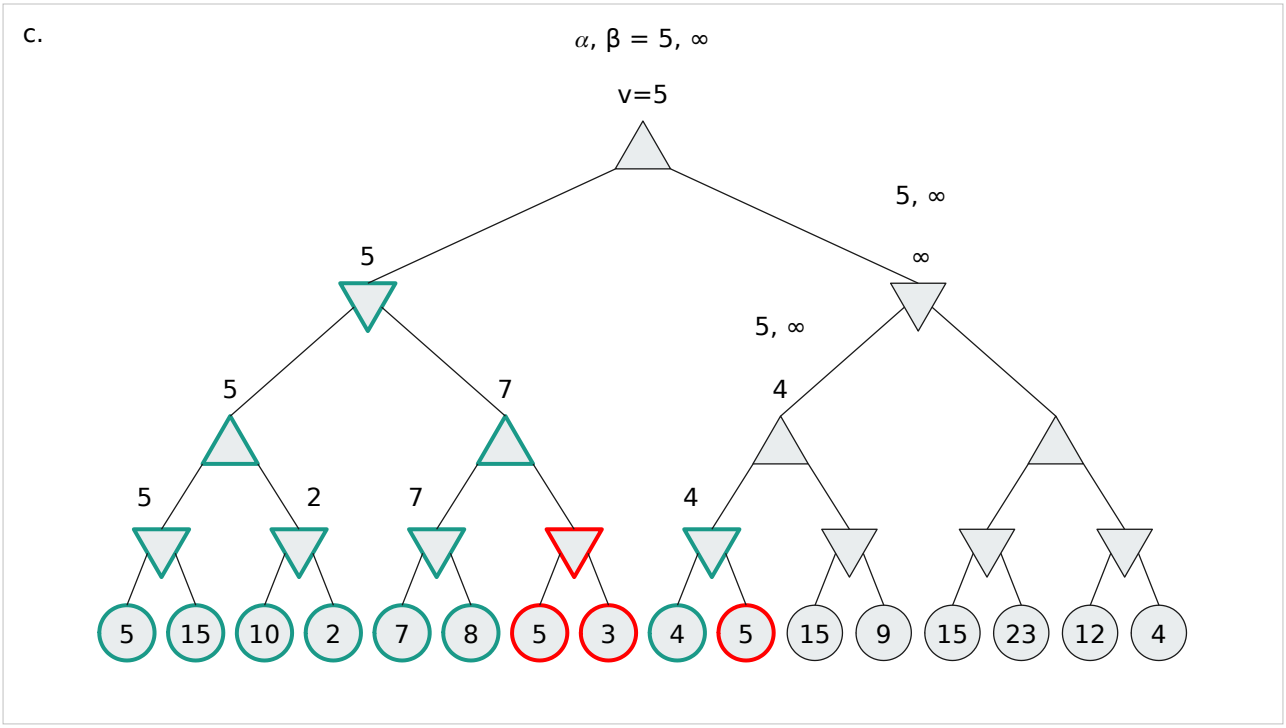
c.



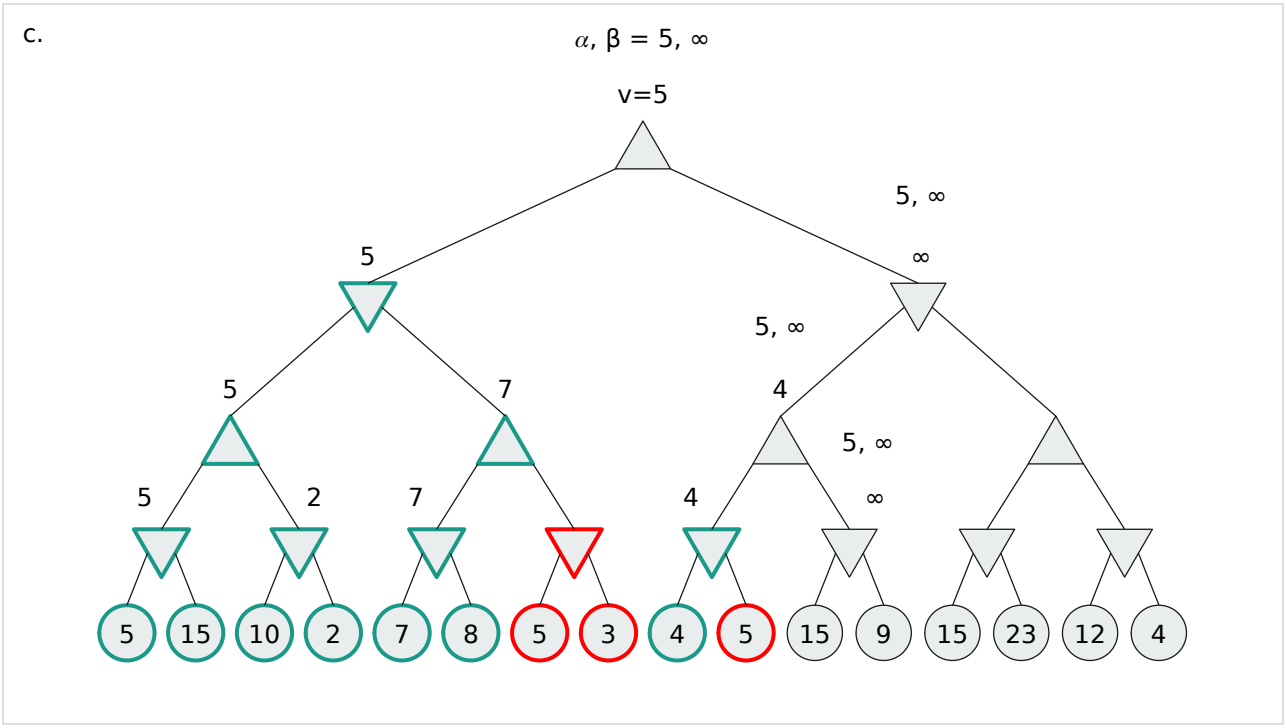
c.



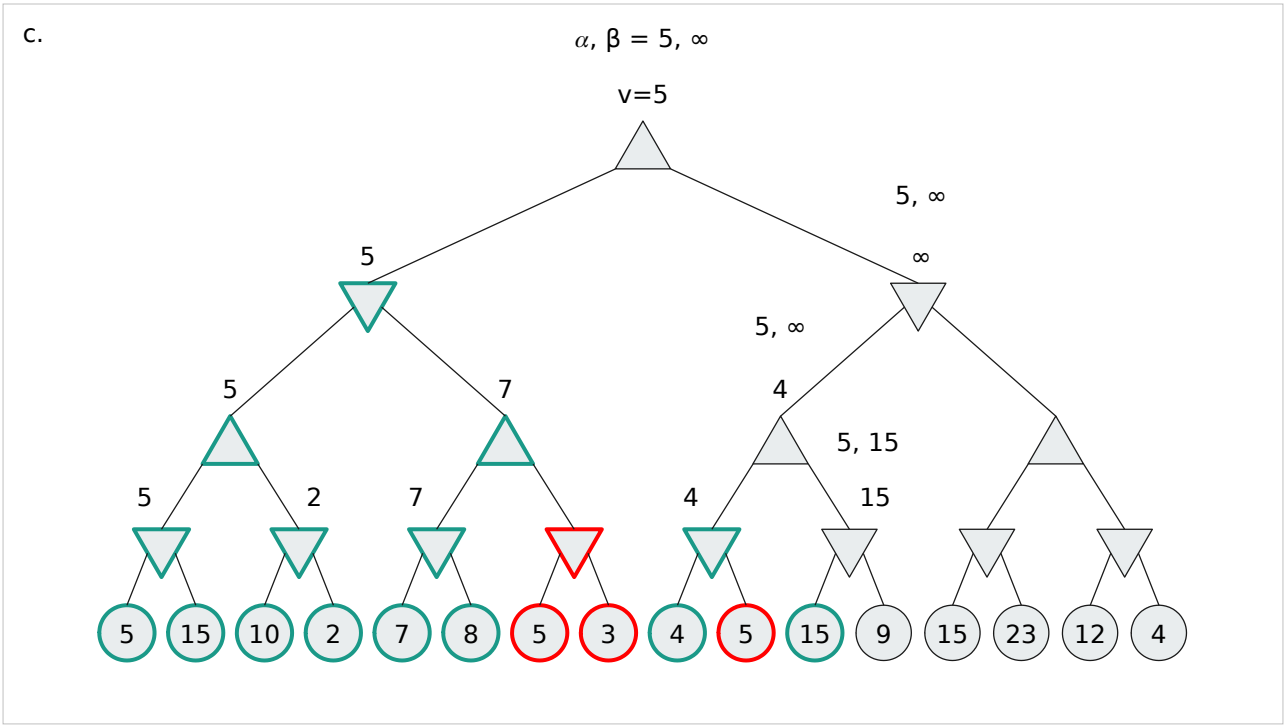
c.



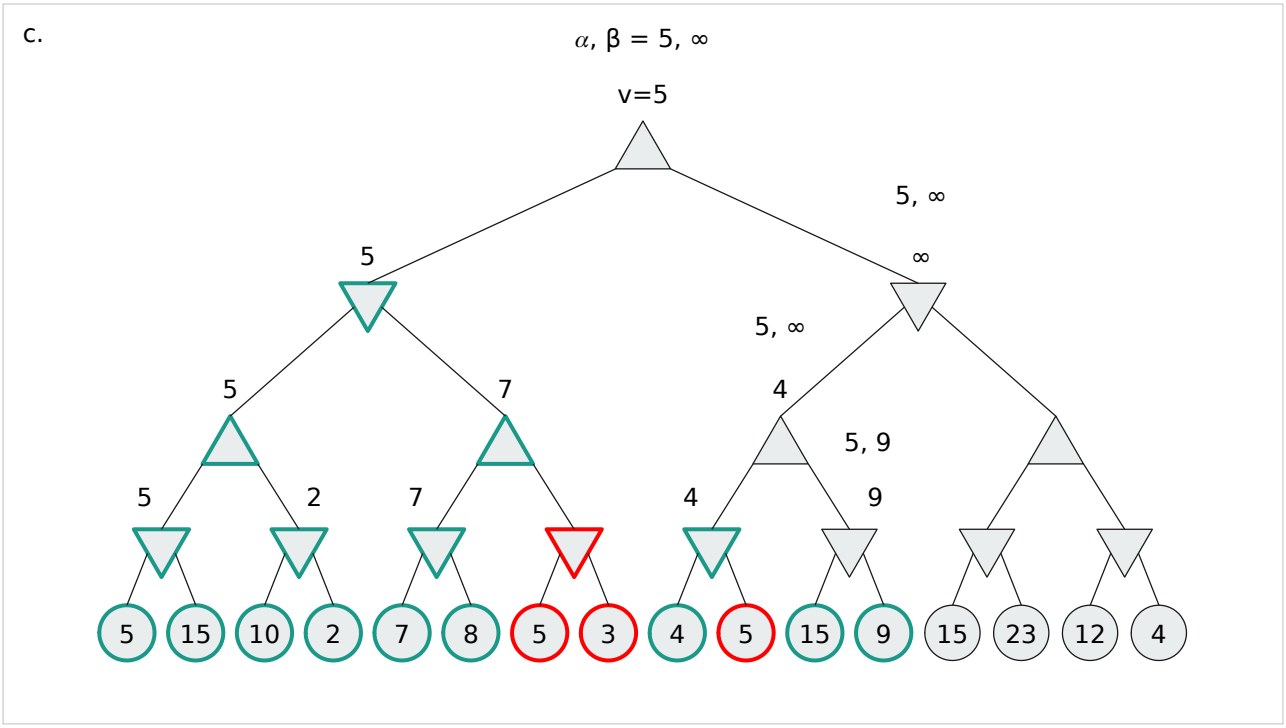
c.



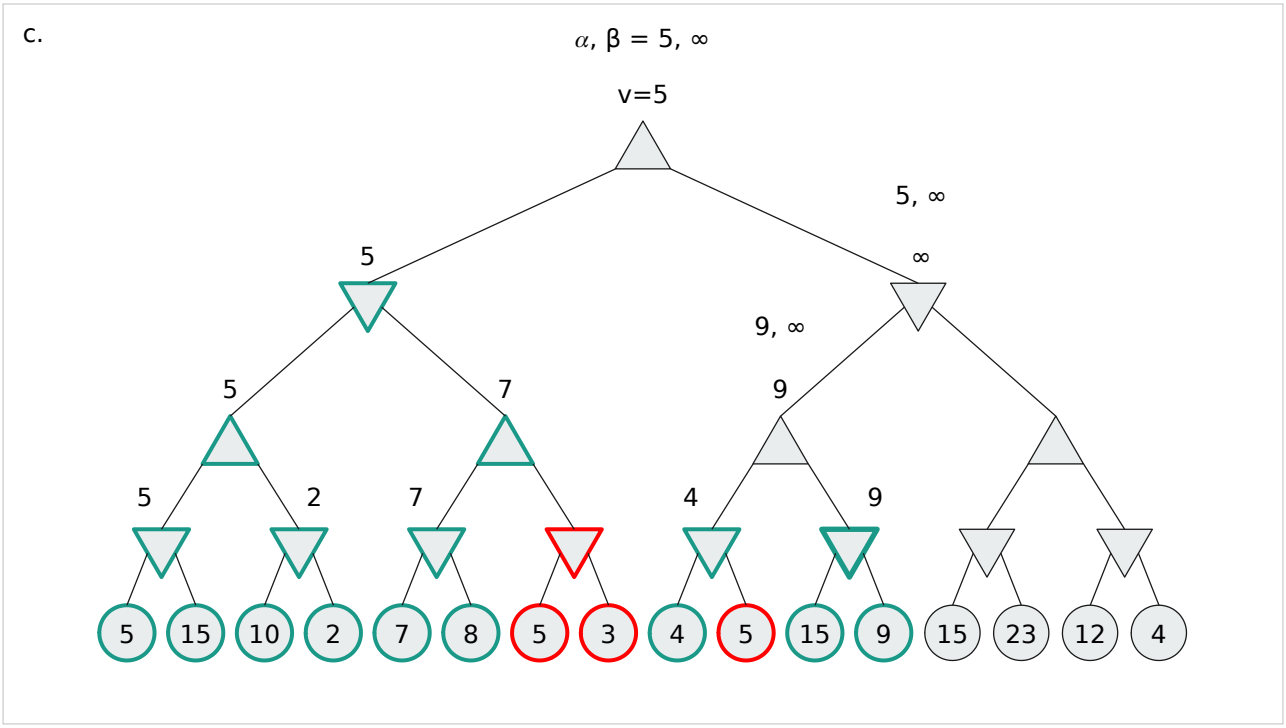
c.



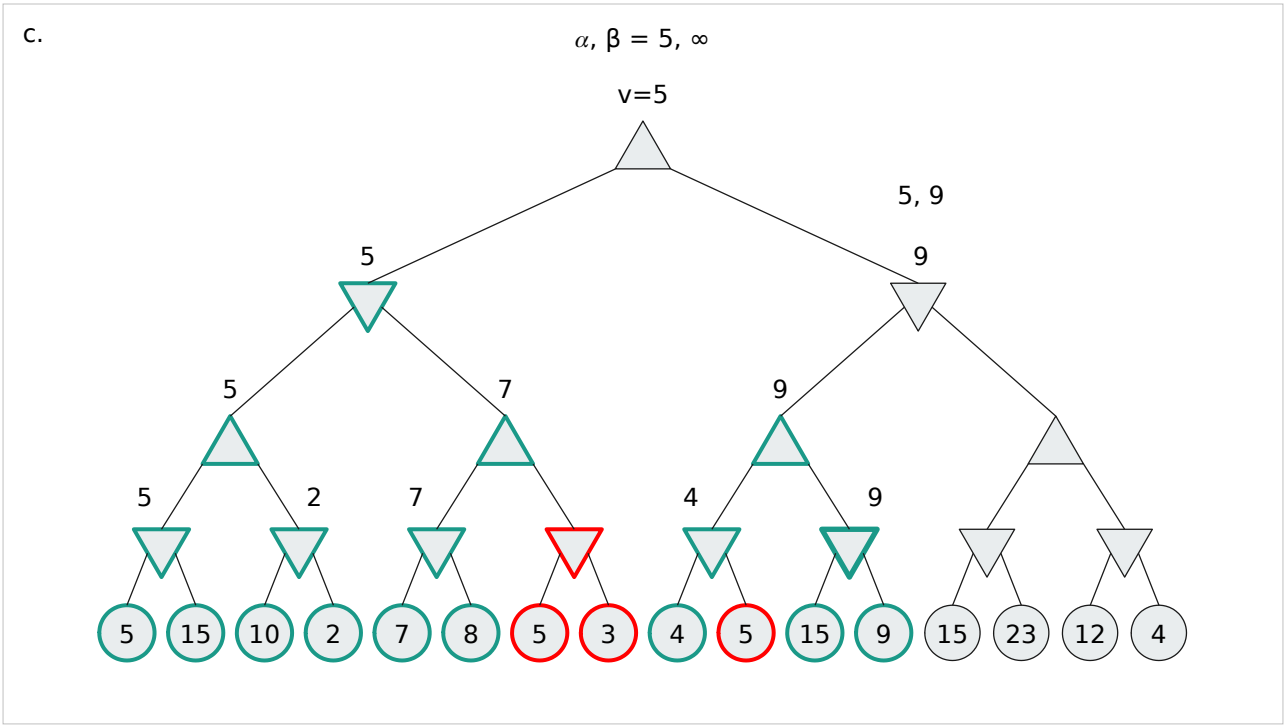
c.



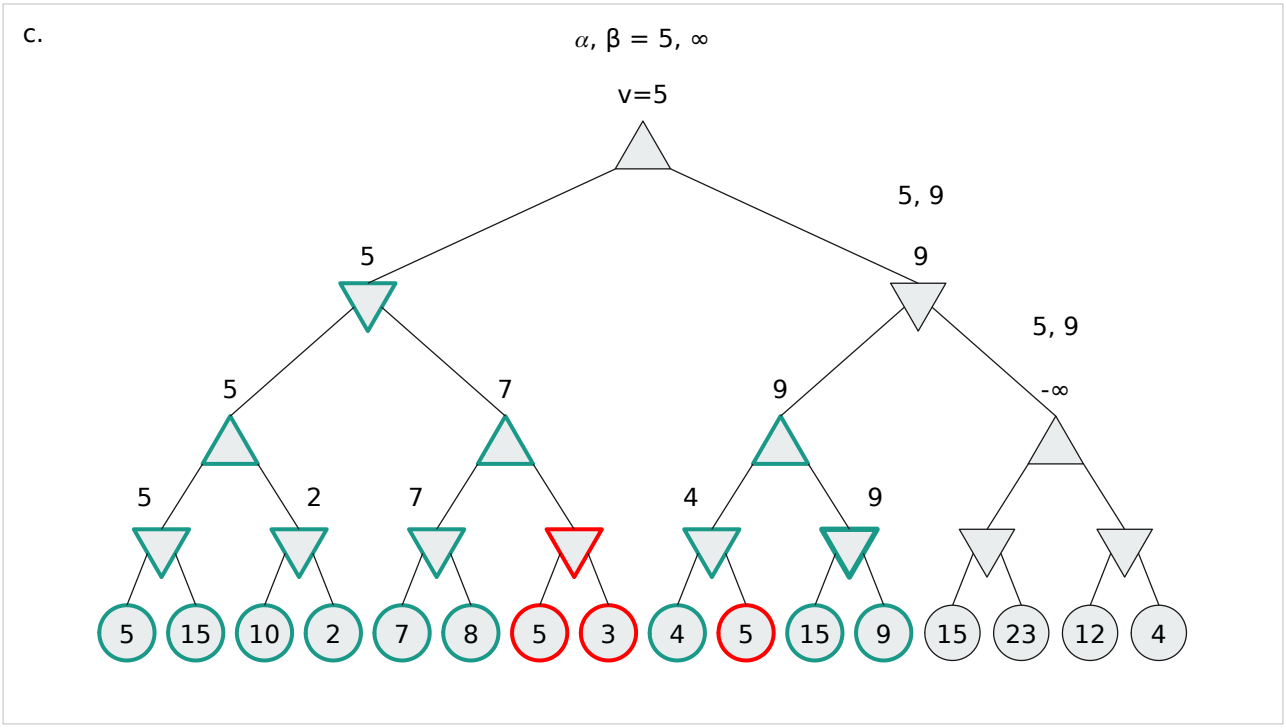
c.



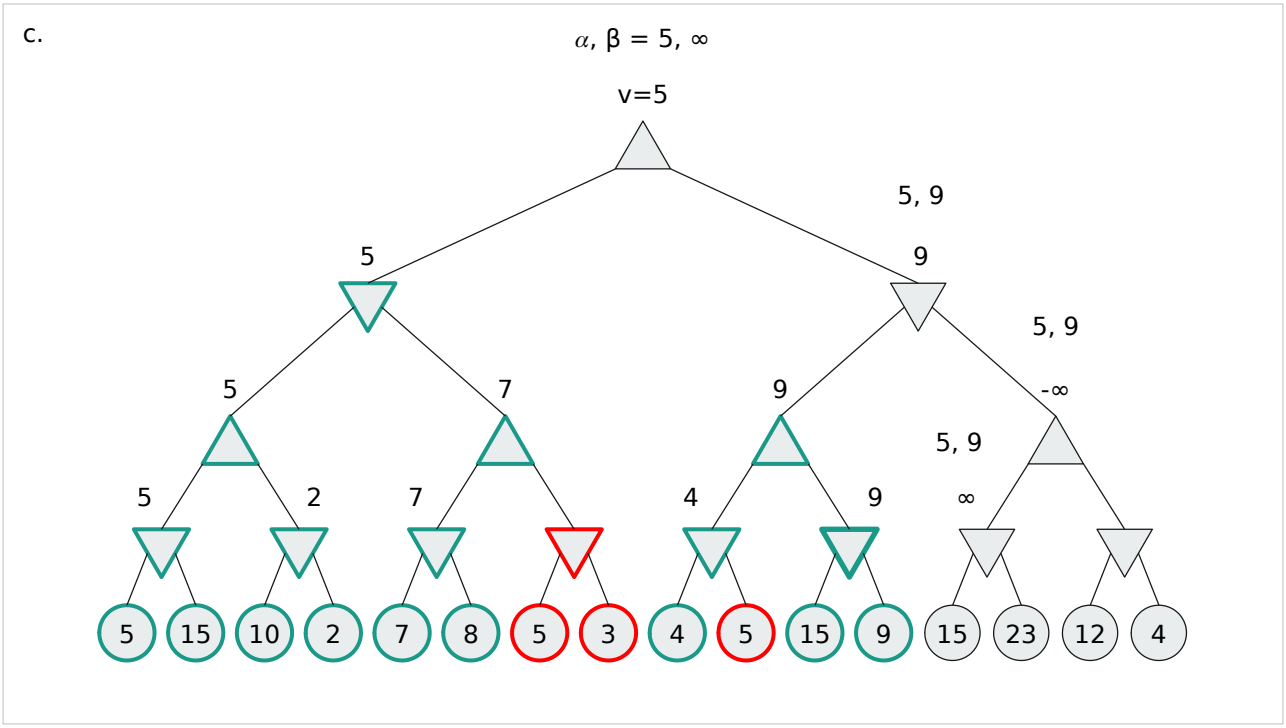
c.



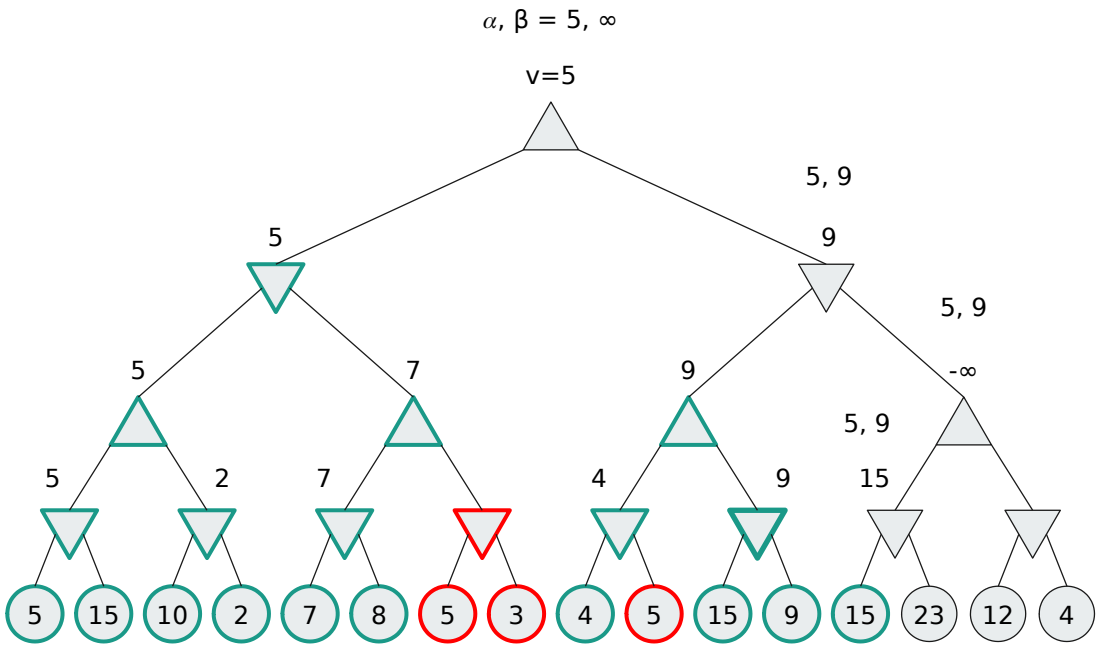
c.



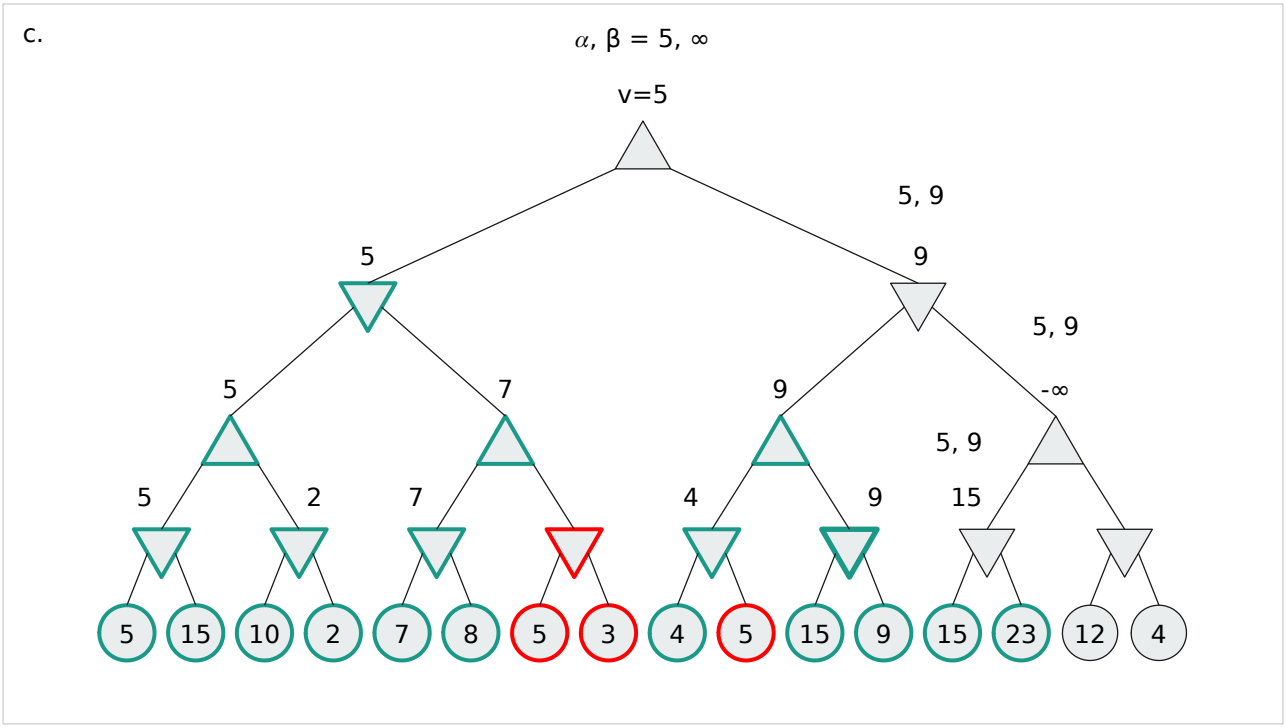
c.



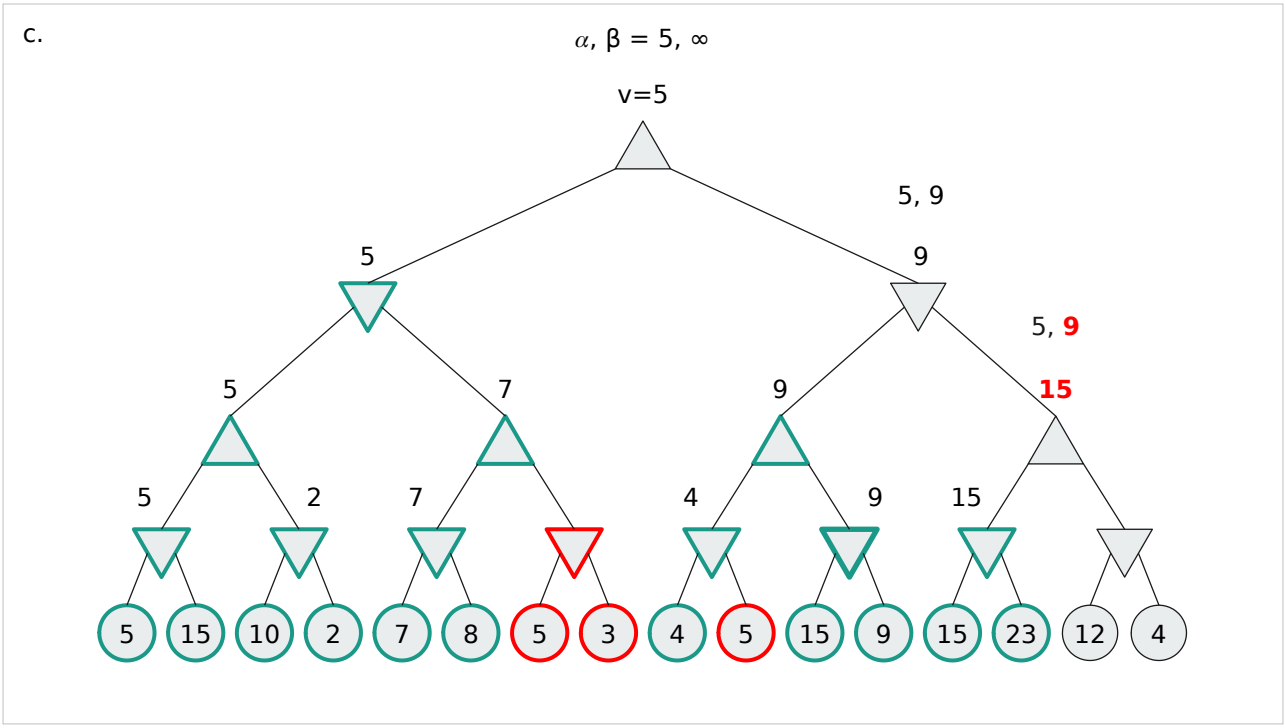
c.



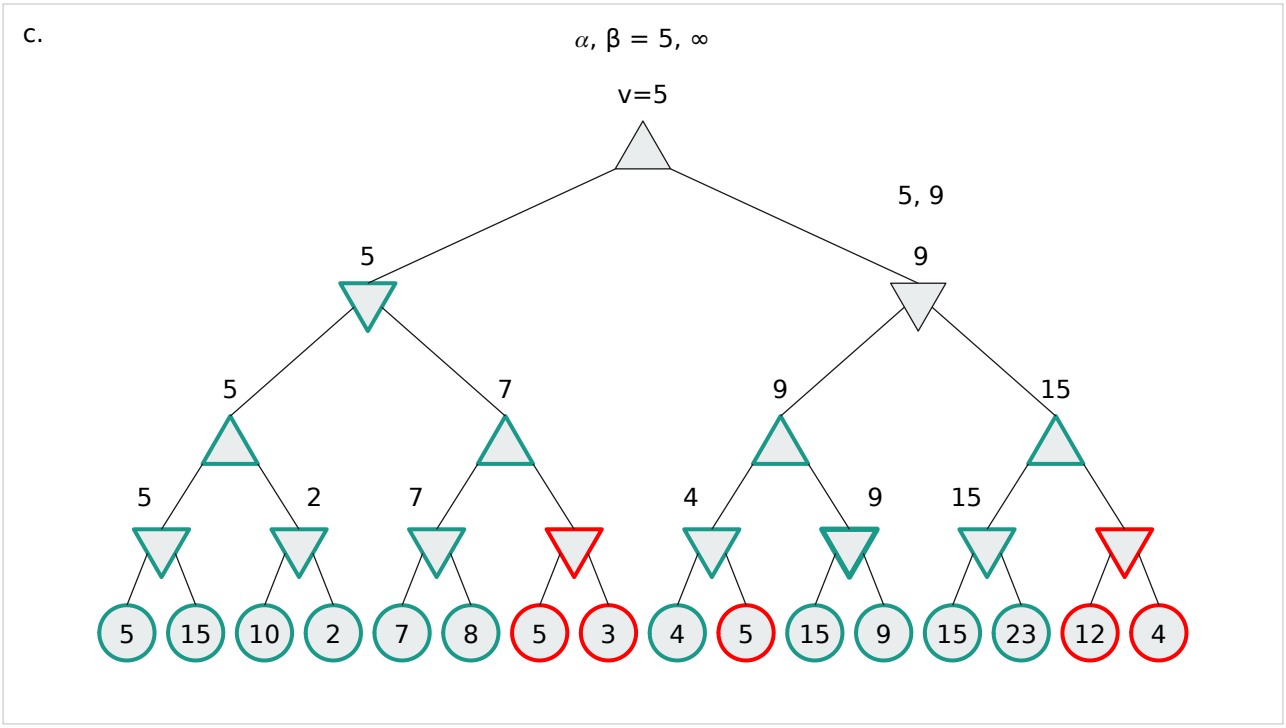
c.



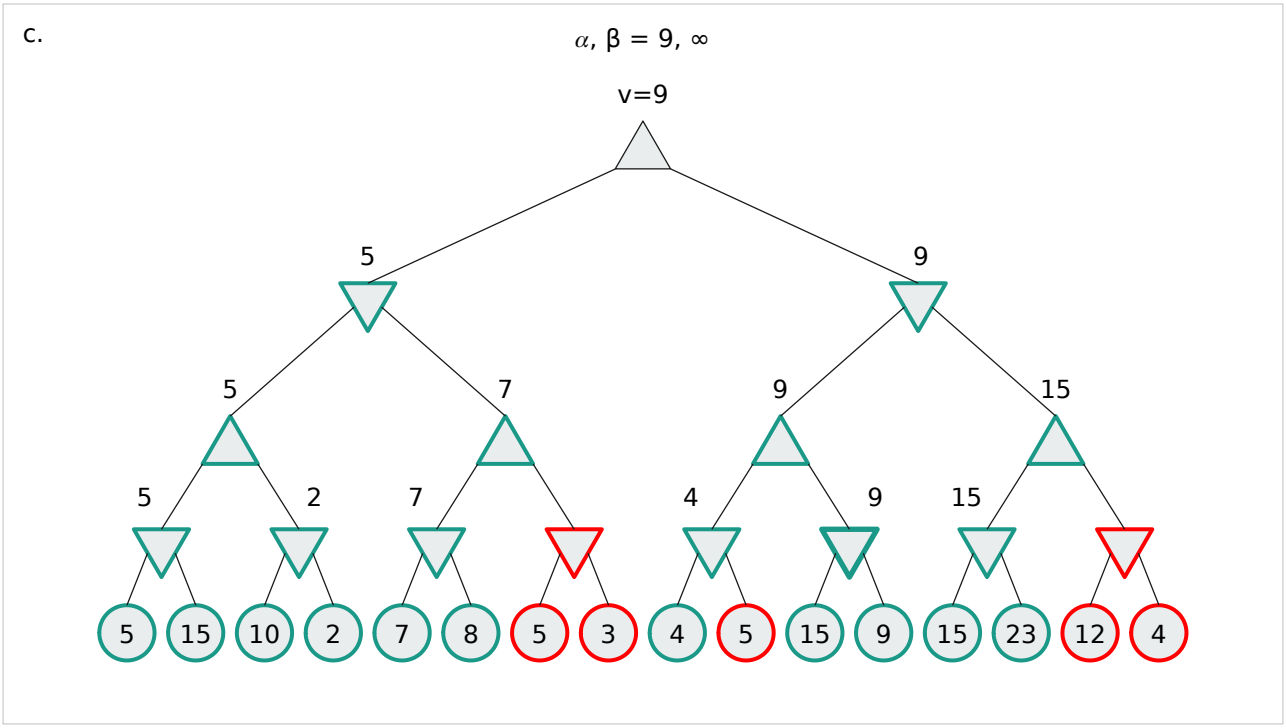
c.



c.

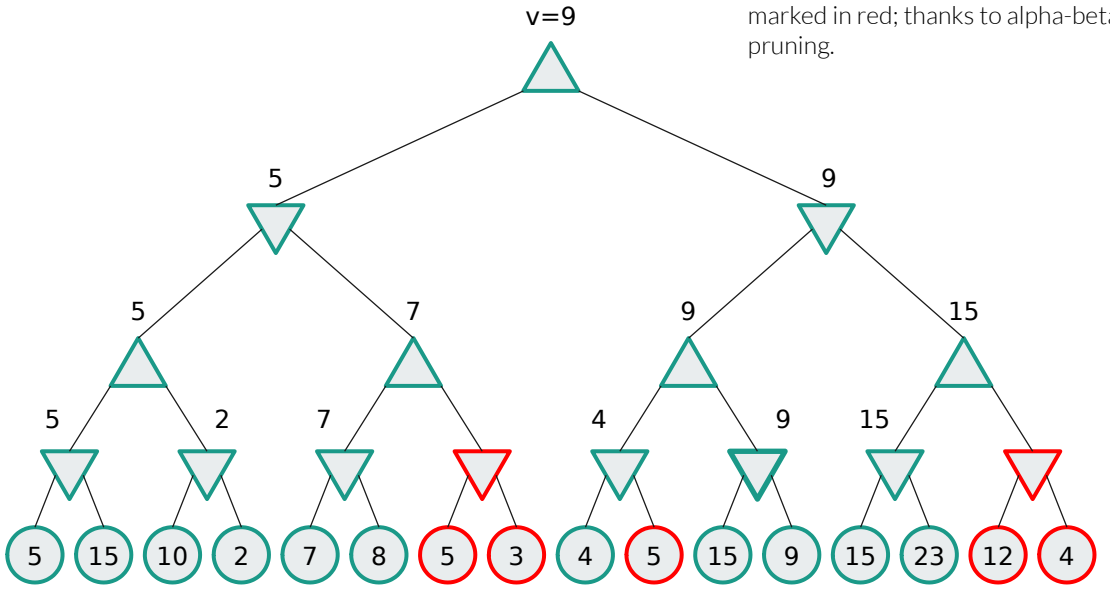


c.



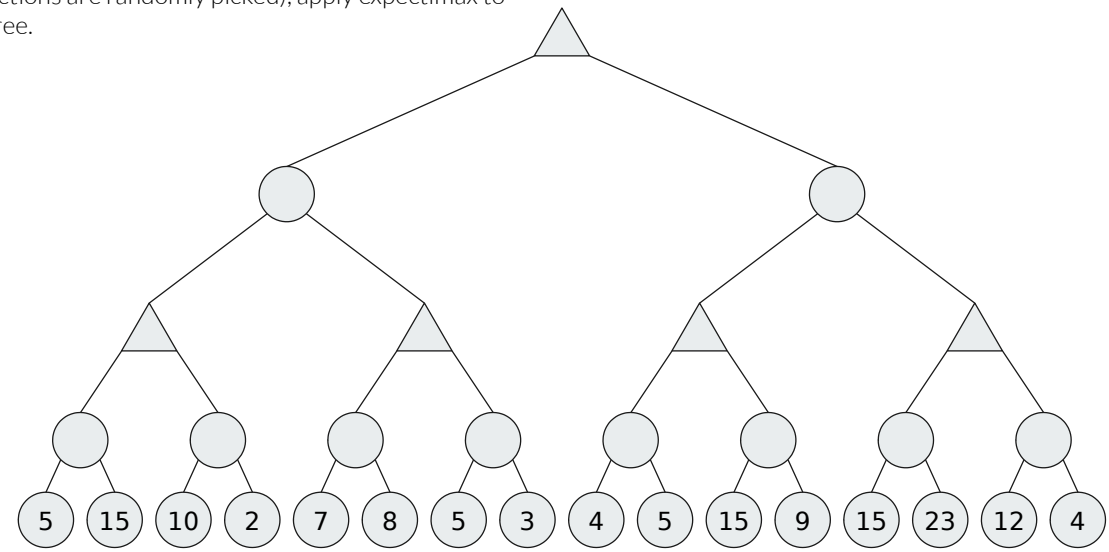
c.

In the end, we were able to solve this problem without evaluating the nodes marked in red; thanks to alpha-beta pruning.



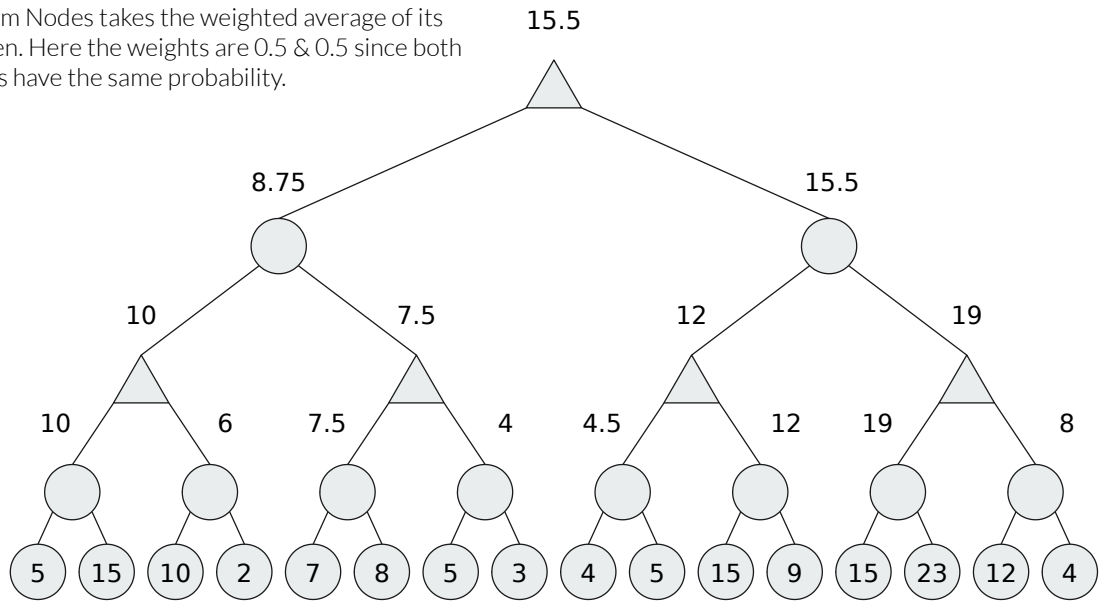
extra.

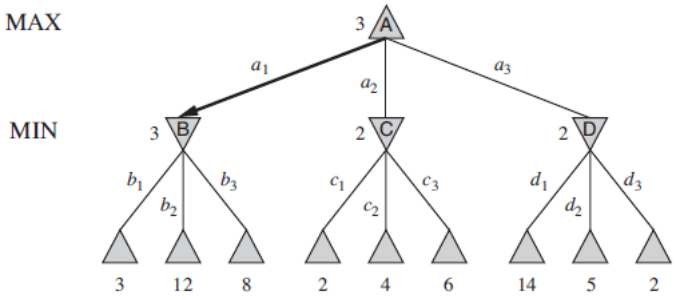
Assuming that the 2nd player is replaced by a fair coin flip (actions are randomly picked), apply expectimax to this tree.



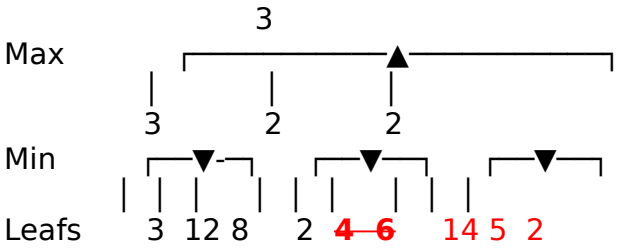
extra.

Random Nodes takes the weighted average of its children. Here the weights are 0.5 & 0.5 since both actions have the same probability.





Apply Alpha Beta Pruning:



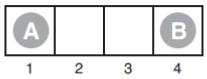


Figure 5.17 The starting position of a simple game. Player A moves first. The two players take turns moving, and each player must move his token to an open adjacent space in either direction. If the opponent occupies an adjacent space, then a player may jump over the opponent to the next open space if any. (For example, if A is on 3 and B is on 2, then A may move back to 1.) The game ends when one player reaches the opposite end of the board. If player A reaches space 4 first, then the value of the game to A is $+1$; if player B reaches space 1 first, then the value of the game to A is -1.

5.8 Consider the two-player game described in Figure 5.17.

a. Draw the complete game tree, using the following conventions:

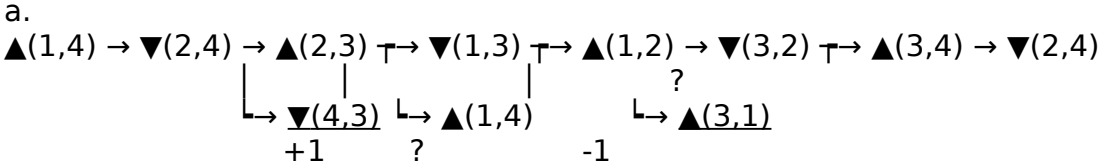
- Write each state as (s_A, s_B), where s_A and s_B denote the token locations.
- Put each terminal state in a square box and write its game value in a circle.
- Put *loop states* (states that already appear on the path to the root) in double square boxes. Since their value is unclear, annotate each with a "?" in a circle.

b. Now mark each node with its backed-up minimax value (also in a circle). Explain how you handled the "?" values and why.

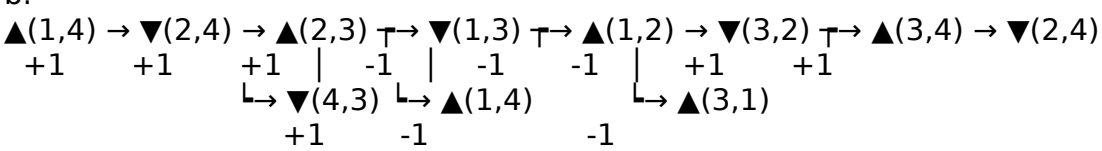
c. Explain why the standard minimax algorithm would fail on this game tree and briefly sketch how you might fix it, drawing on your answer to (b). Does your modified algorithm give optimal decisions for all games with loops?

d. This 4-square game can be generalized to n squares for any $n > 2$. Prove that A wins if n is even and loses if n is odd.

a.



b.



Since there are two possible values only {-1, 1}, we can do:

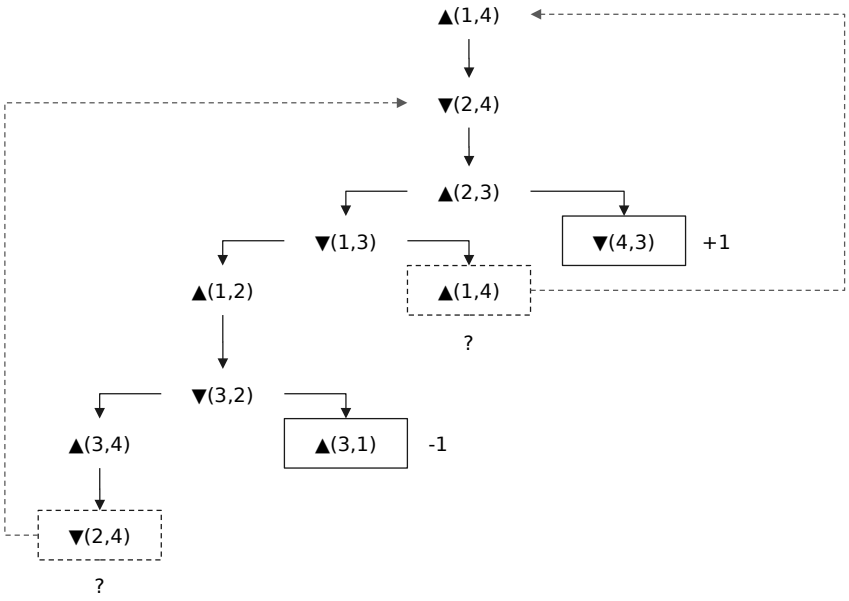
max(?, ?) = ?, max(?, -1) = ?, max(?, 1) = 1
max(-1, -1) = -1, max(-1, 1) = 1
max(1, 1) = 1

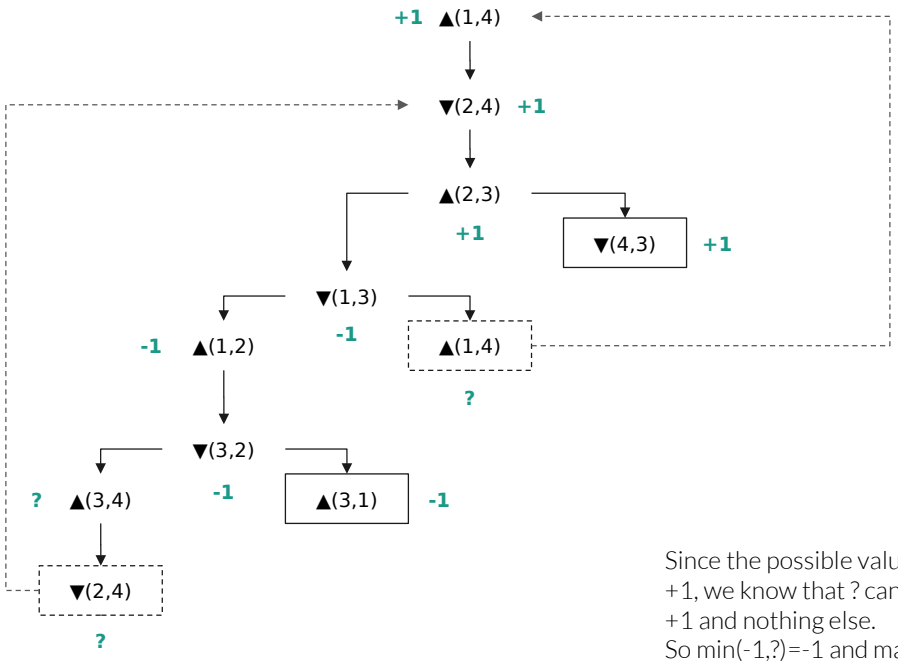
min(?, ?) = ?, min(?, -1) = -1, min(?, 1) = ?
min(-1, -1) = -1, min(-1, 1) = -1
min(1, 1) = 1

c. The standard minimax algorithm would get stuck in an infinite loop. One way to fix it is to store states along the path to the root and whenever we visit a loop node, we return with "?" and handle it as shown in "b". This will not handle all cases since we may end up propagating "?" to the root and fail to get an answer. Also we have no way to handle "?" if the known values are not binary (not just -1 & 1).

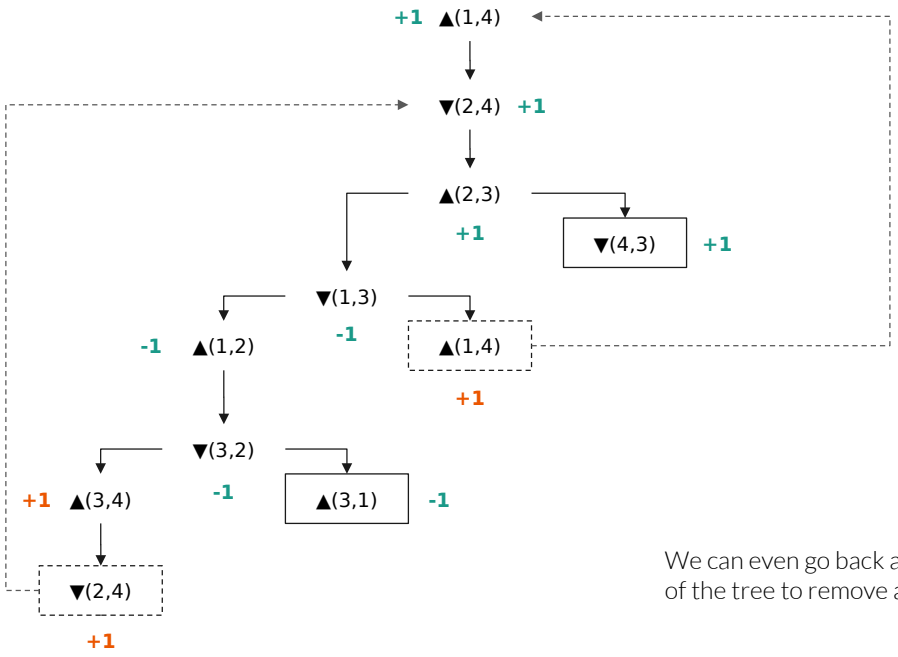
d.

This question is a little tricky. One approach is a proof by induction on the size of the game. Clearly, the base case n=3 is a loss for A and the base


▲(1,4)
▼(2,4)
▲(2,3)
▼(1,3)
▼(4,3)
+1
▲(1,2)
▲(1,4)
?
▼(3,2)
▲(3,4)
▲(3,1)
-1
▼(2,4)
?



Since the possible values are only -1 and +1, we know that ? can either hold -1 or +1 and nothing else.
So min(-1,?)=-1 and max(1,?)=1

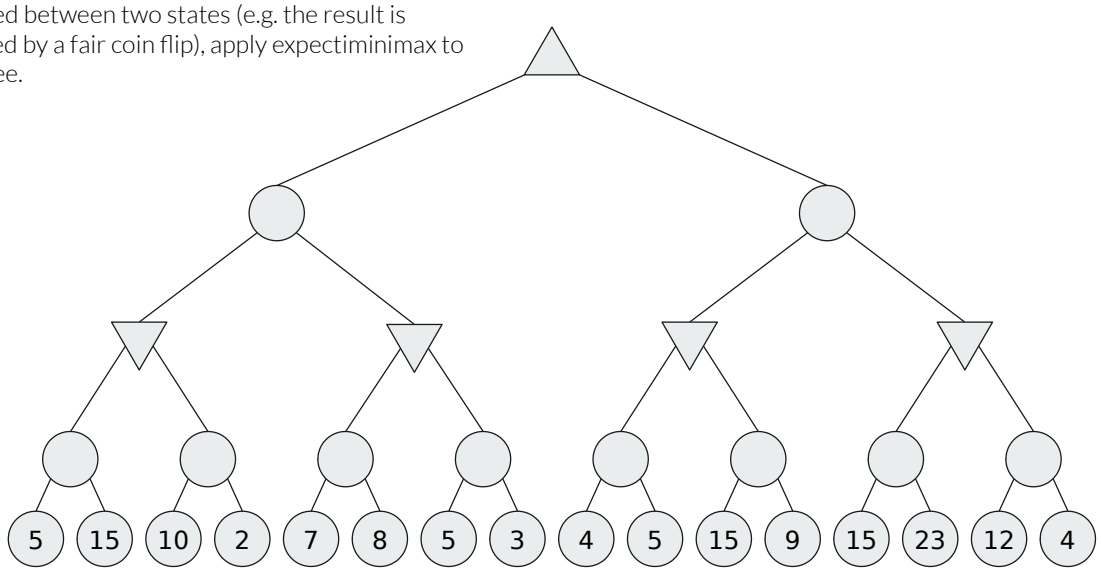


We can even go back and update the rest of the tree to remove all "?" if needed.

More Questions

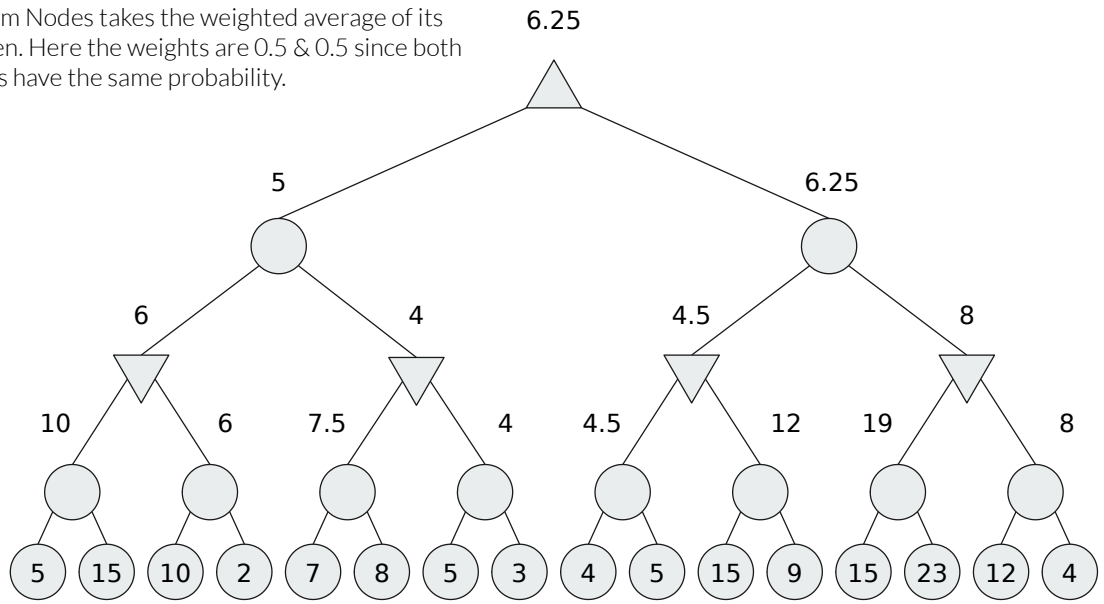
extra.

Assuming that the result of each action is randomly selected between two states (e.g. the result is affected by a fair coin flip), apply expectiminimax to this tree.



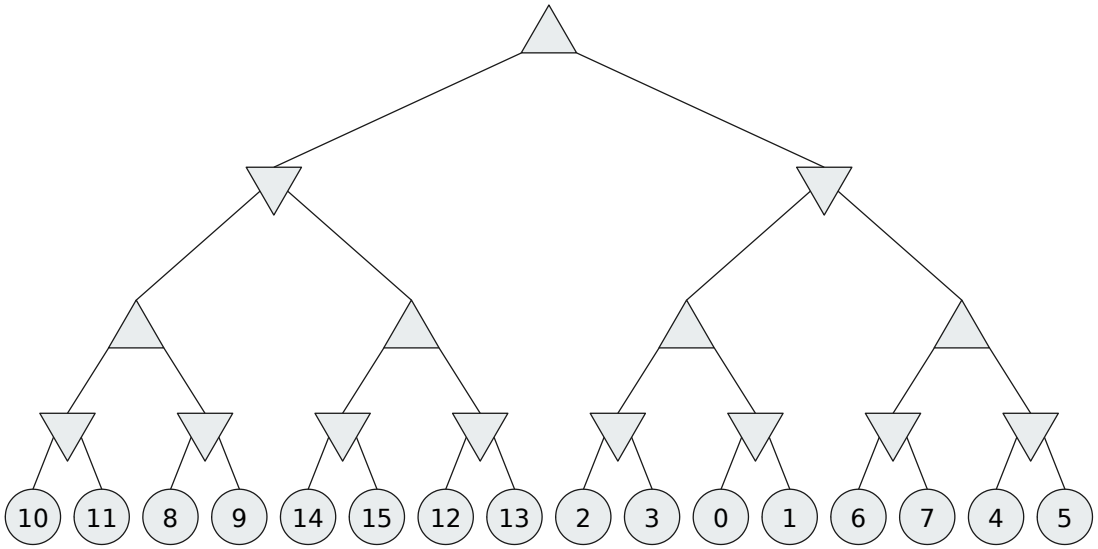
extra.

Random Nodes takes the weighted average of its children. Here the weights are 0.5 & 0.5 since both actions have the same probability.

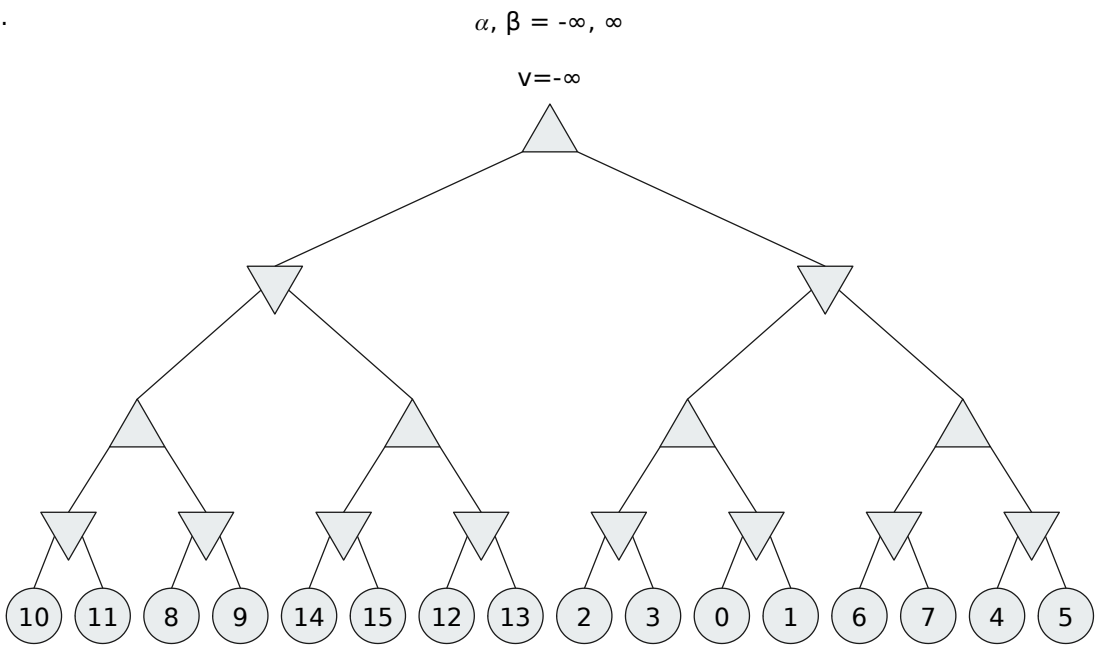


extra.

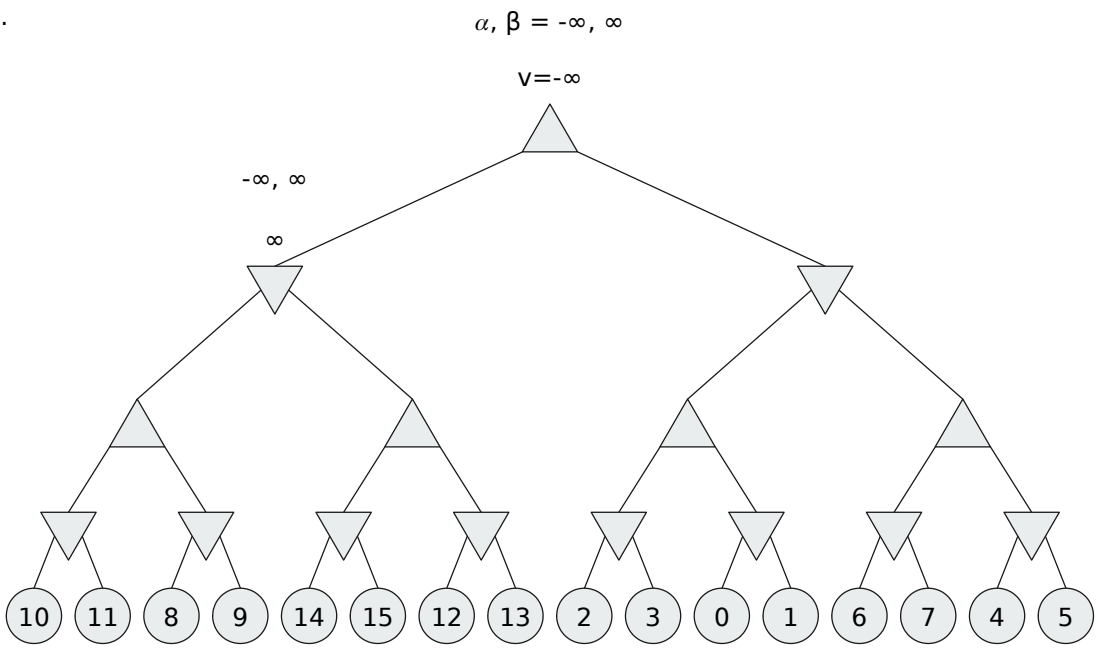
Apply Alpha Beta Pruning to this tree



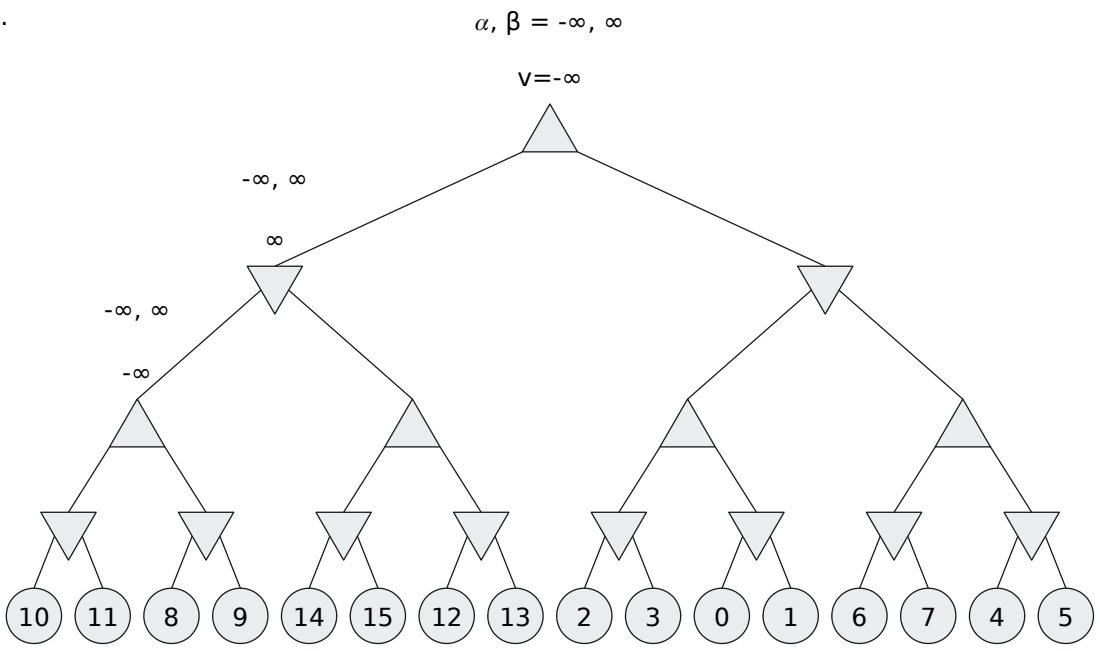
extra.



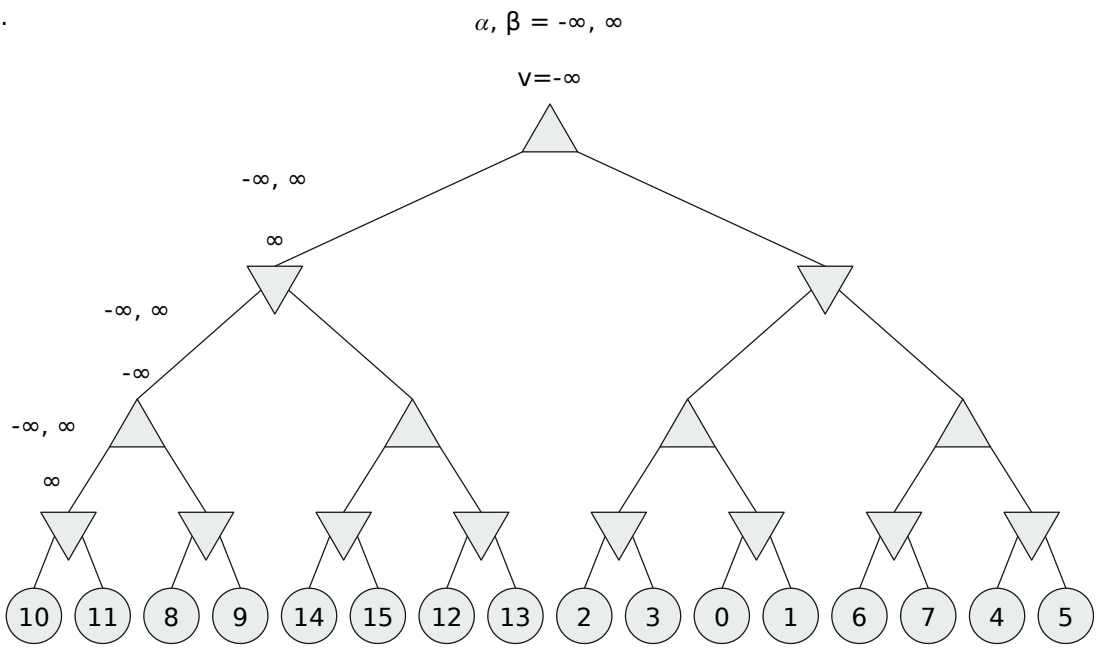
extra.



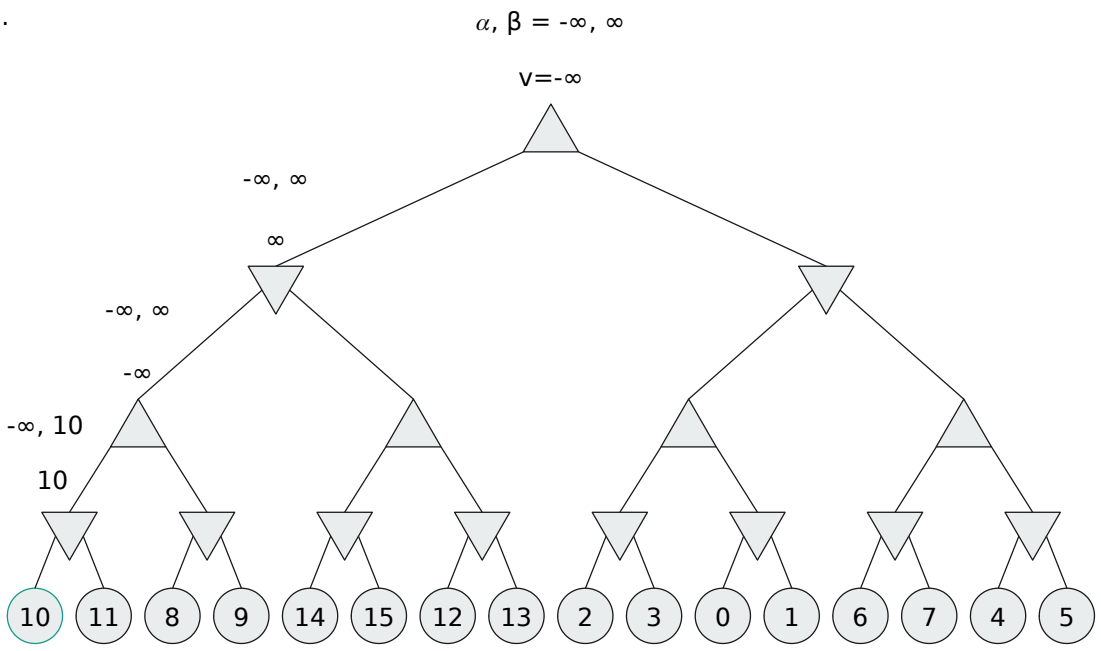
extra.



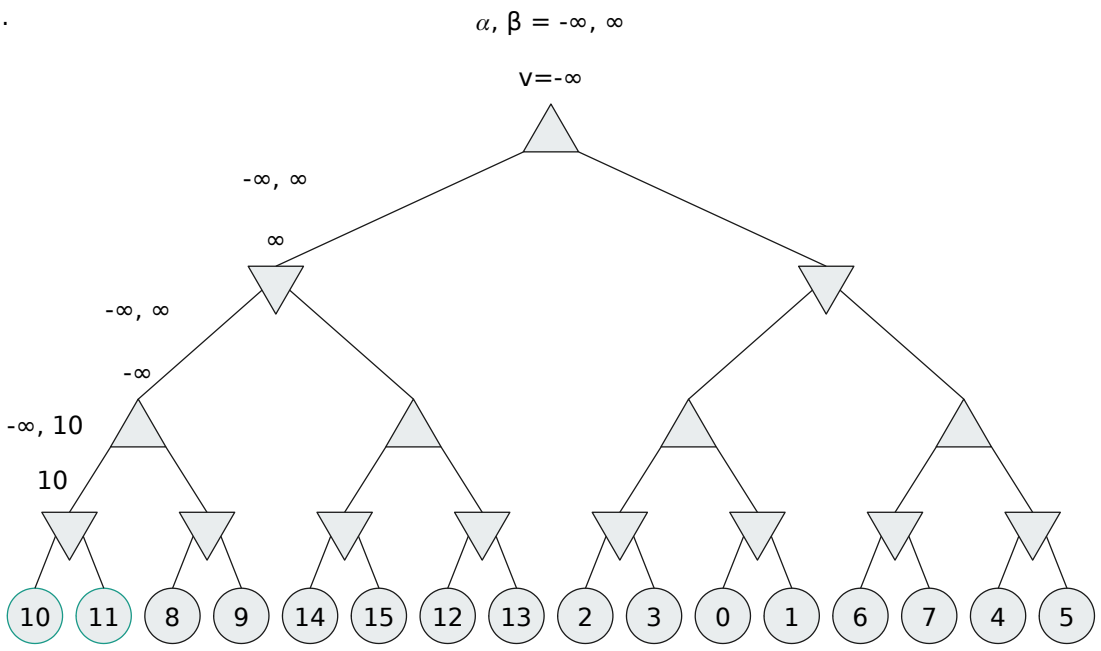
extra.



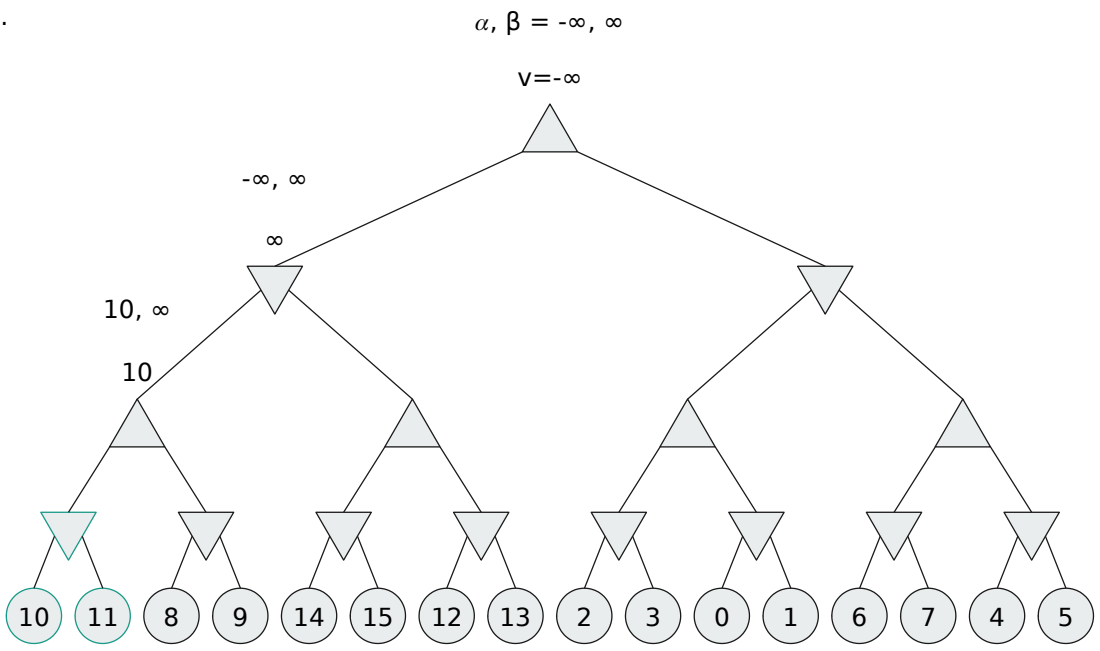
extra.



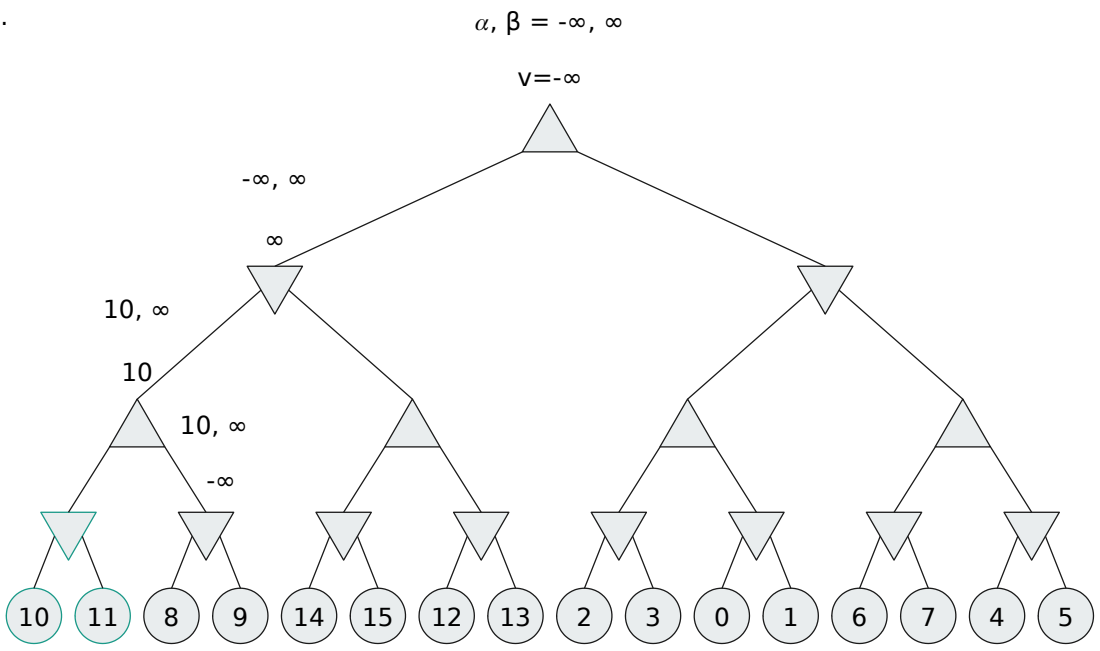
extra.



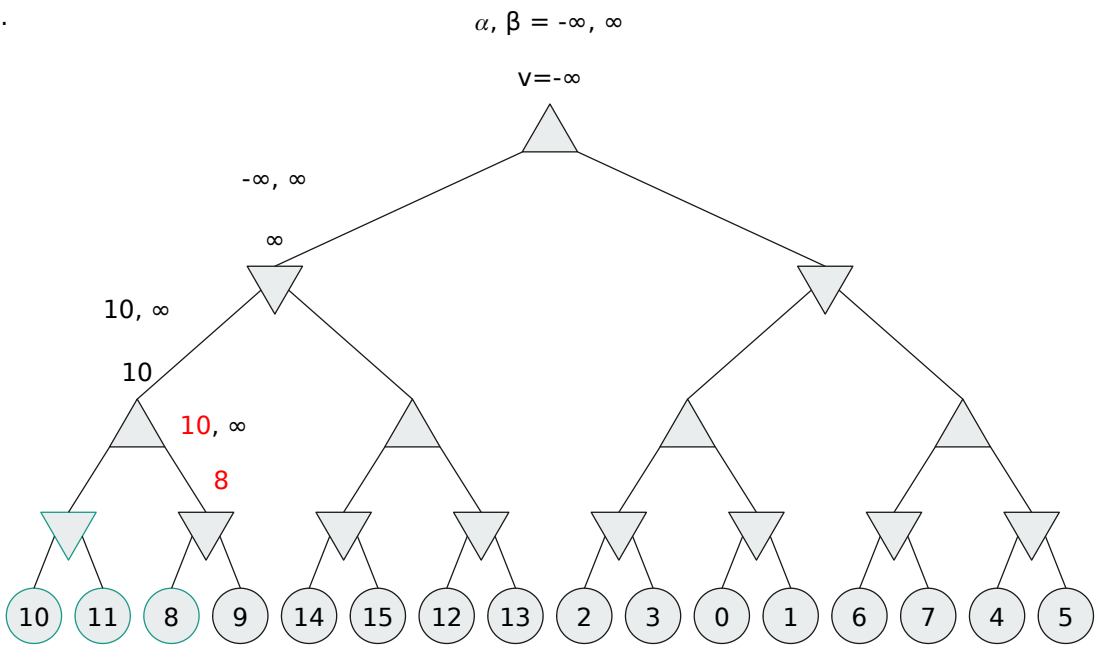
extra.



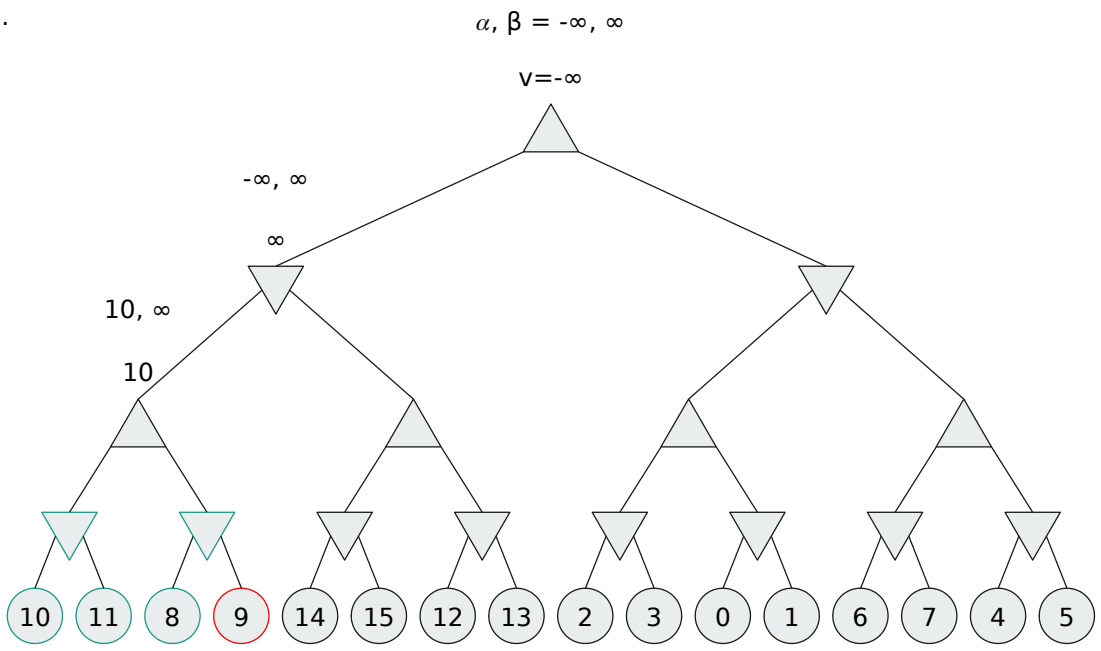
extra.



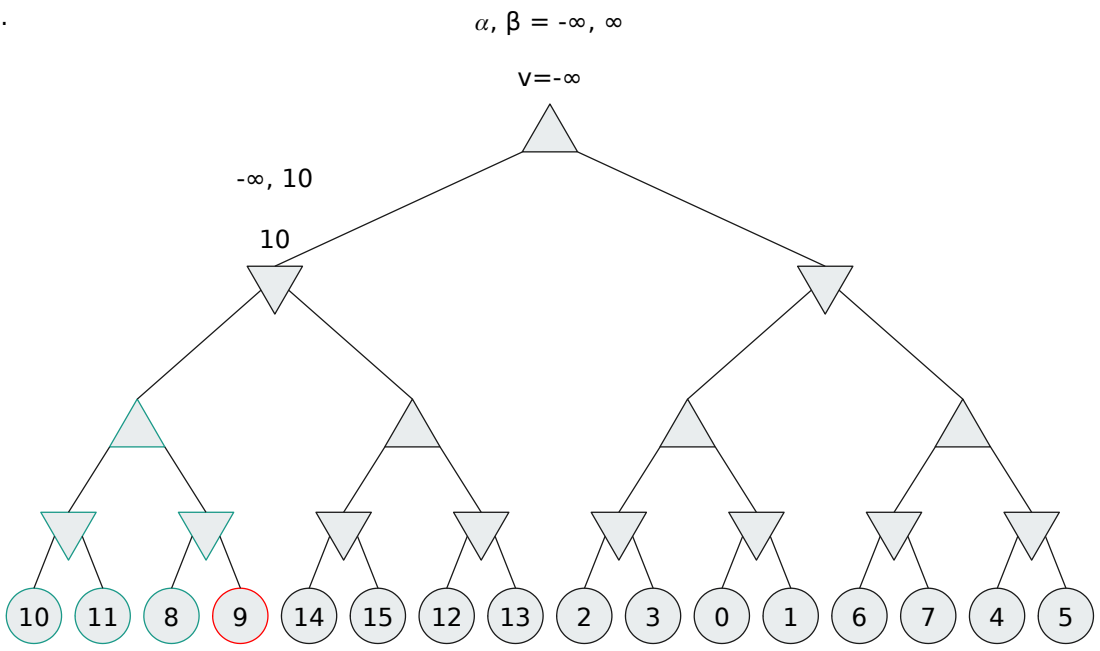
extra.



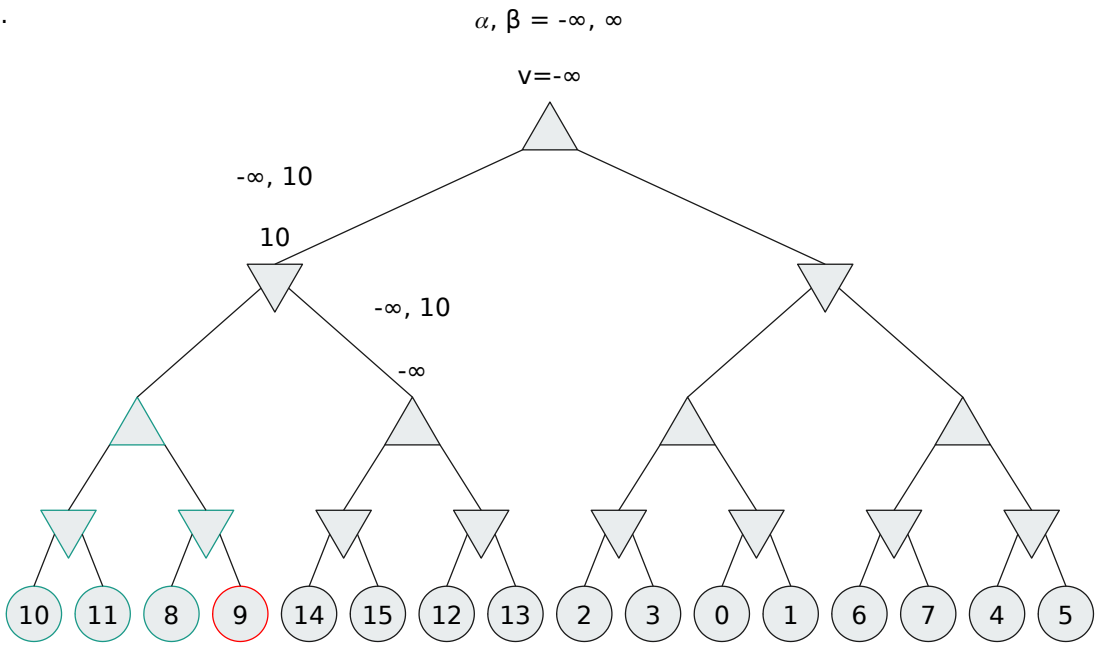
extra.



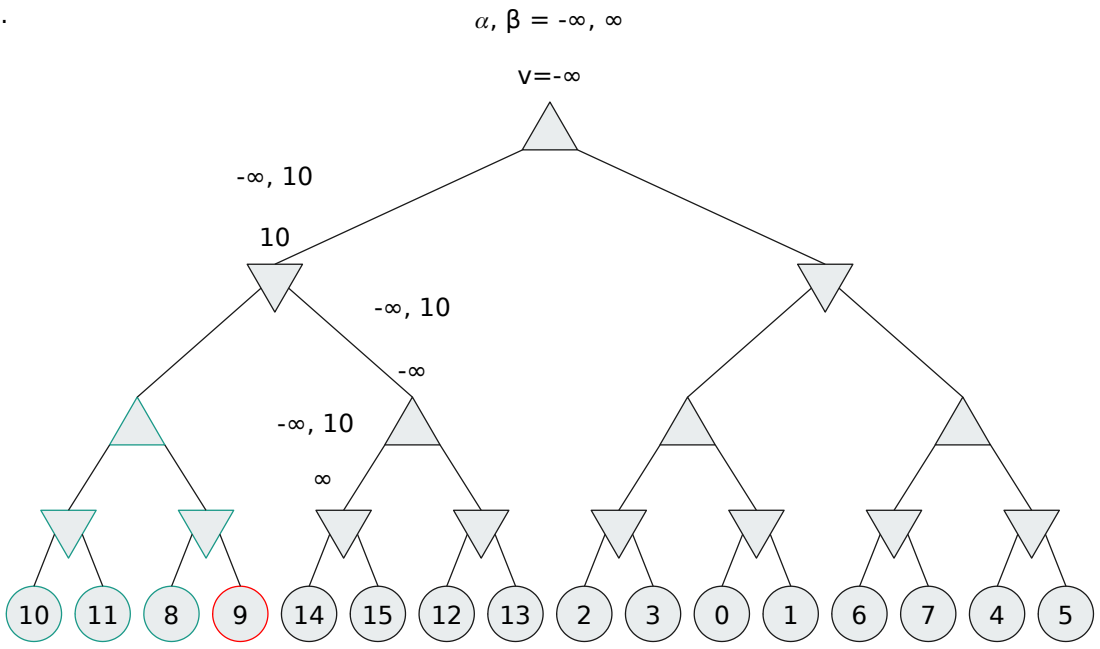
extra.



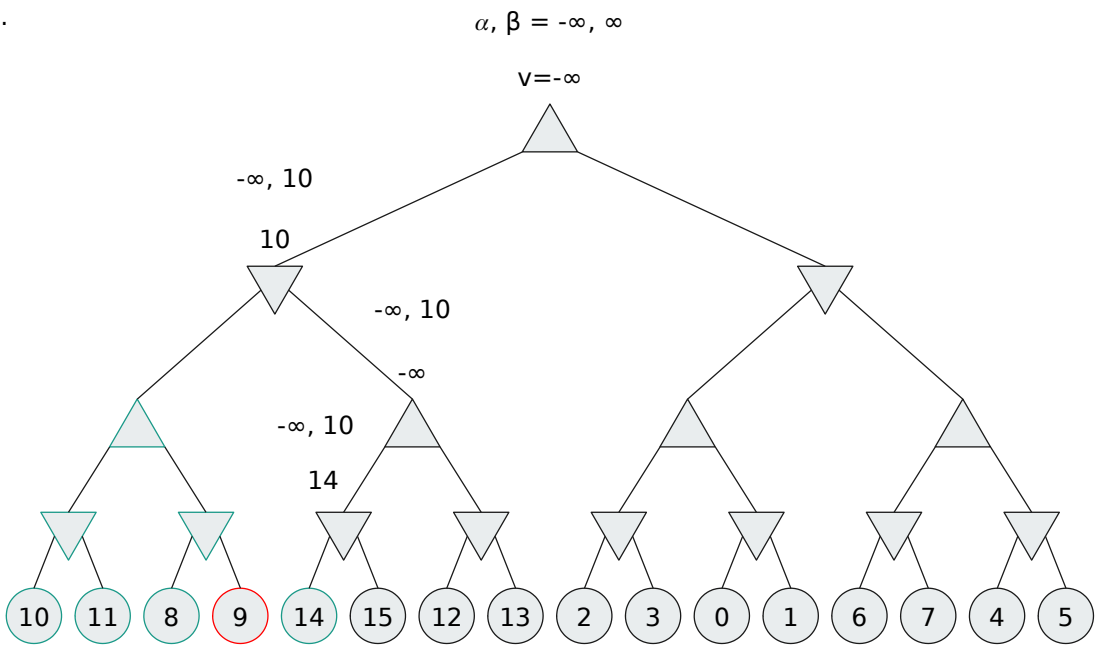
extra.



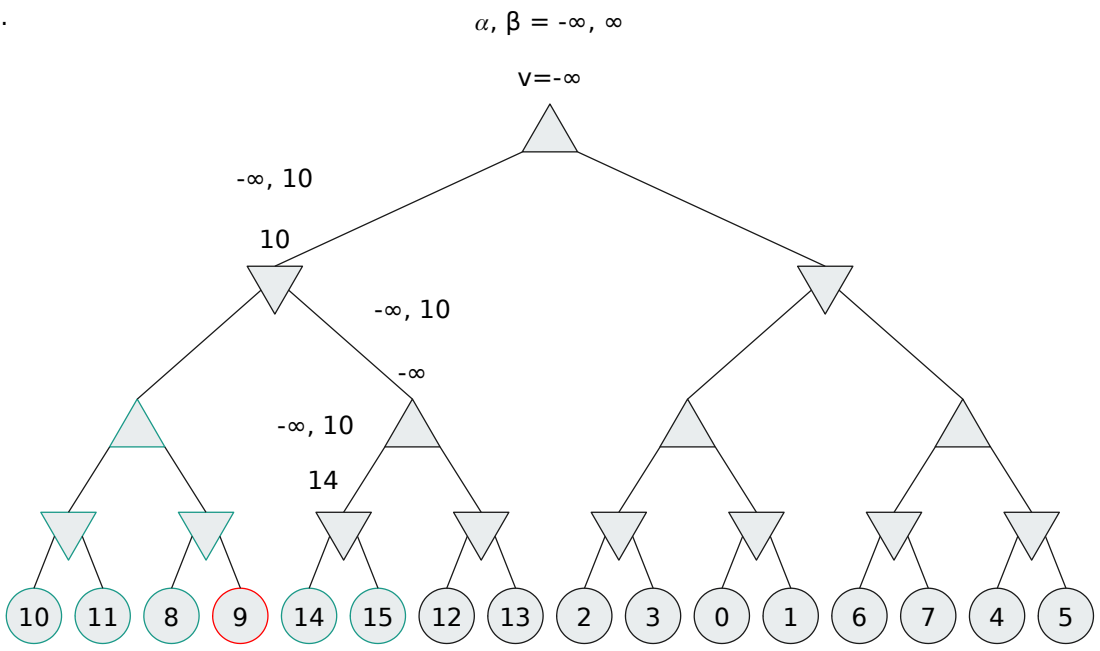
extra.



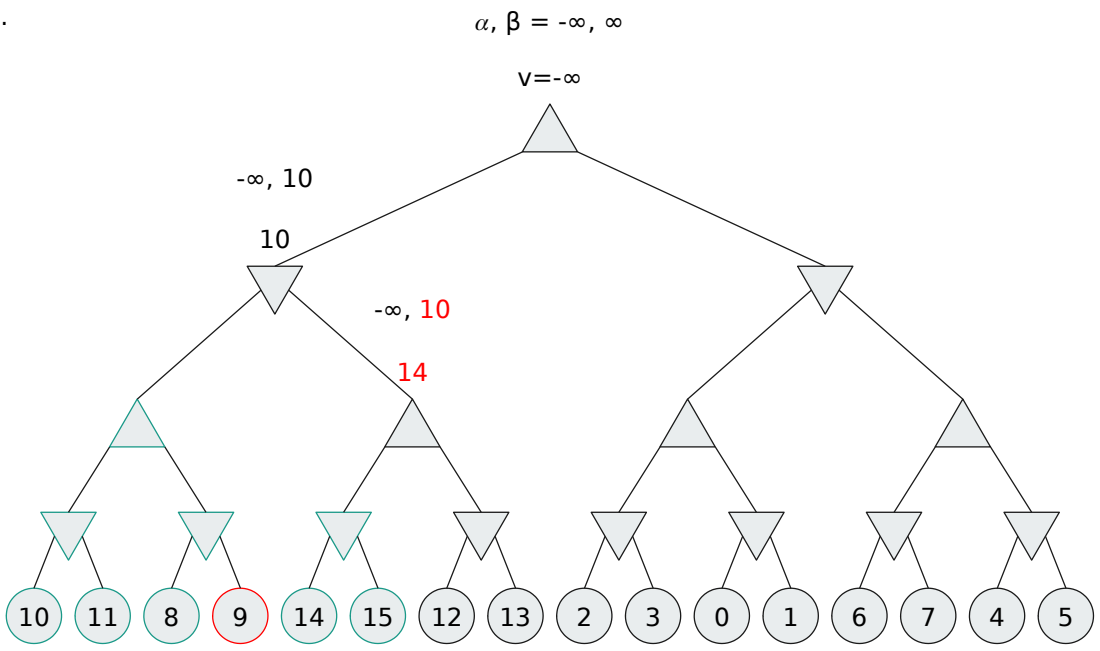
extra.



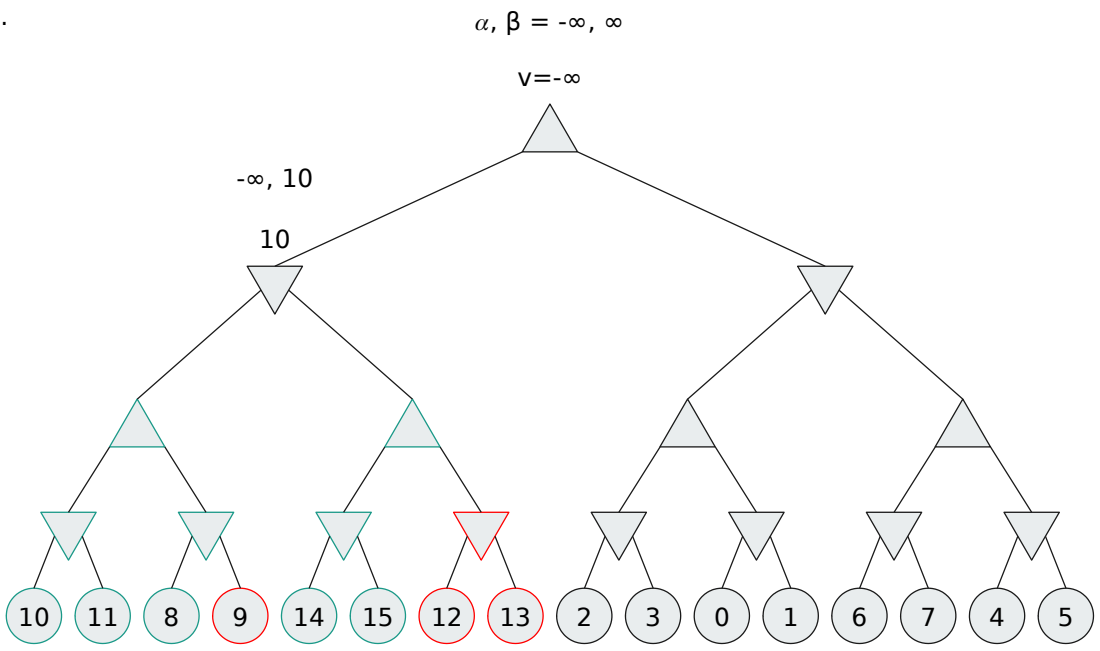
extra.



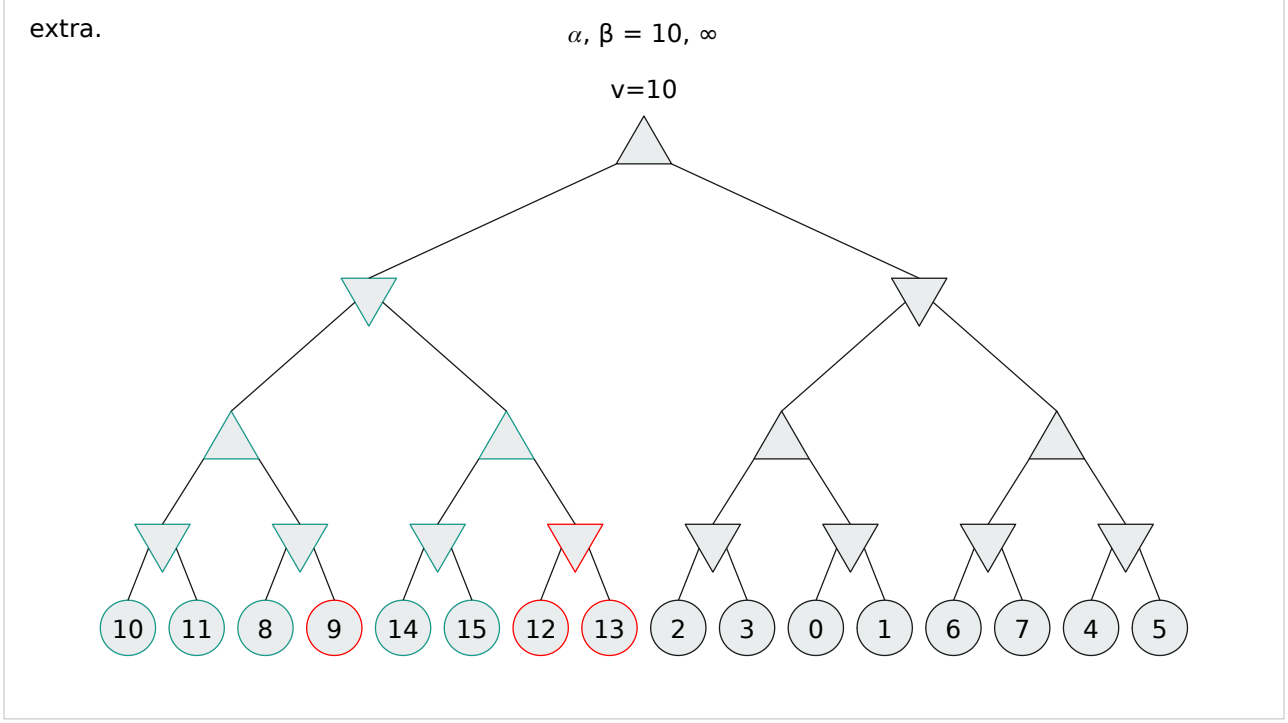
extra.



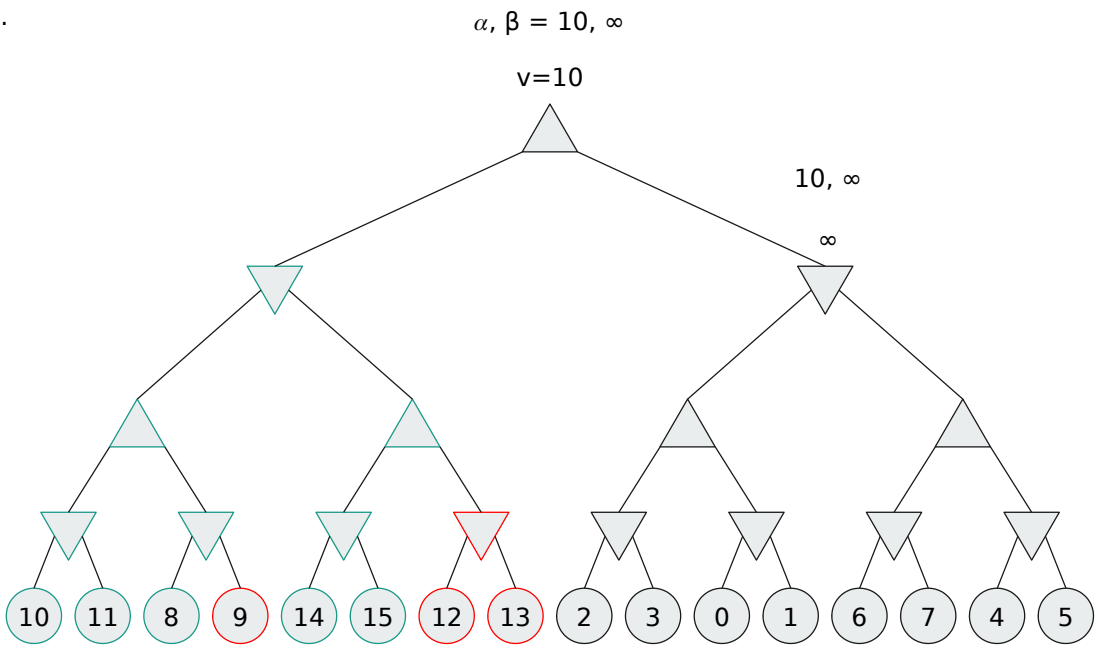
extra.



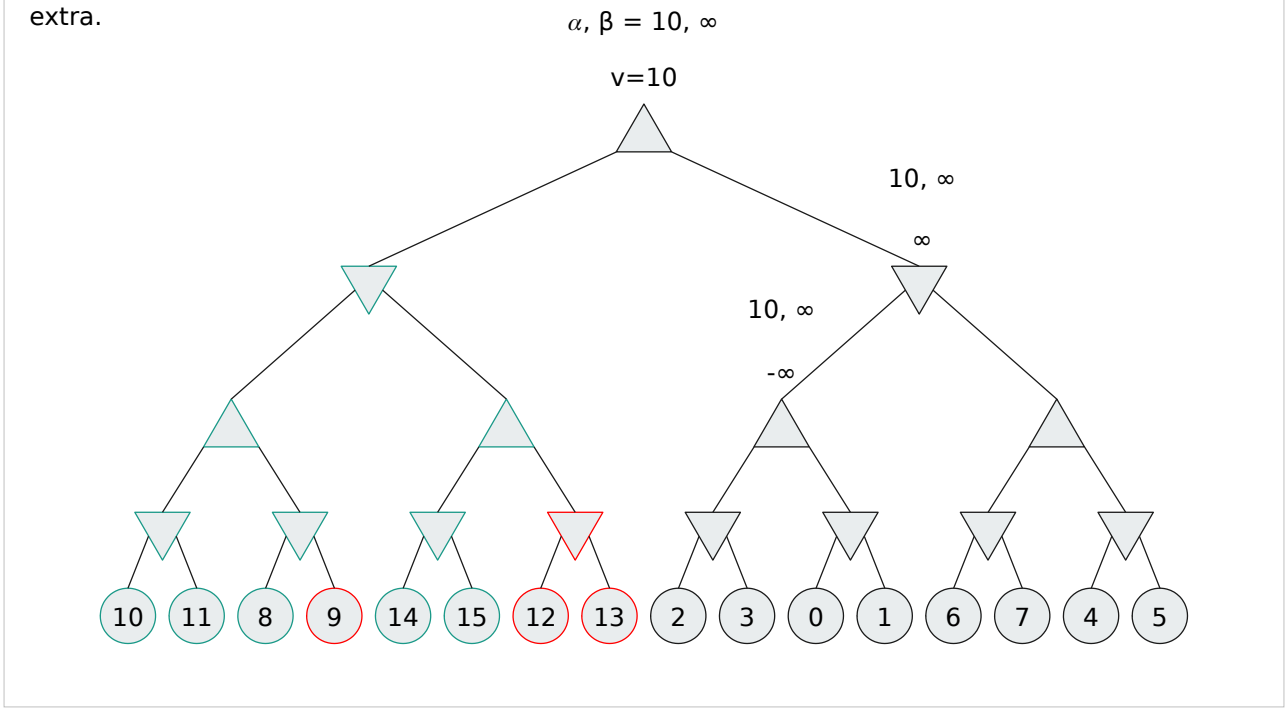
extra.



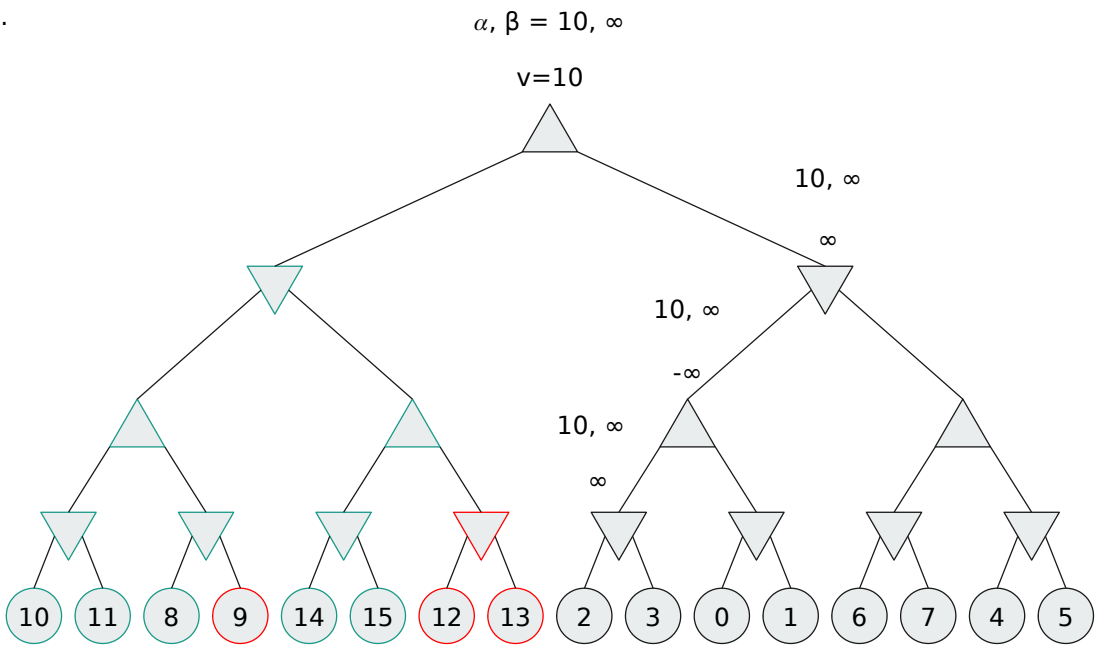
extra.



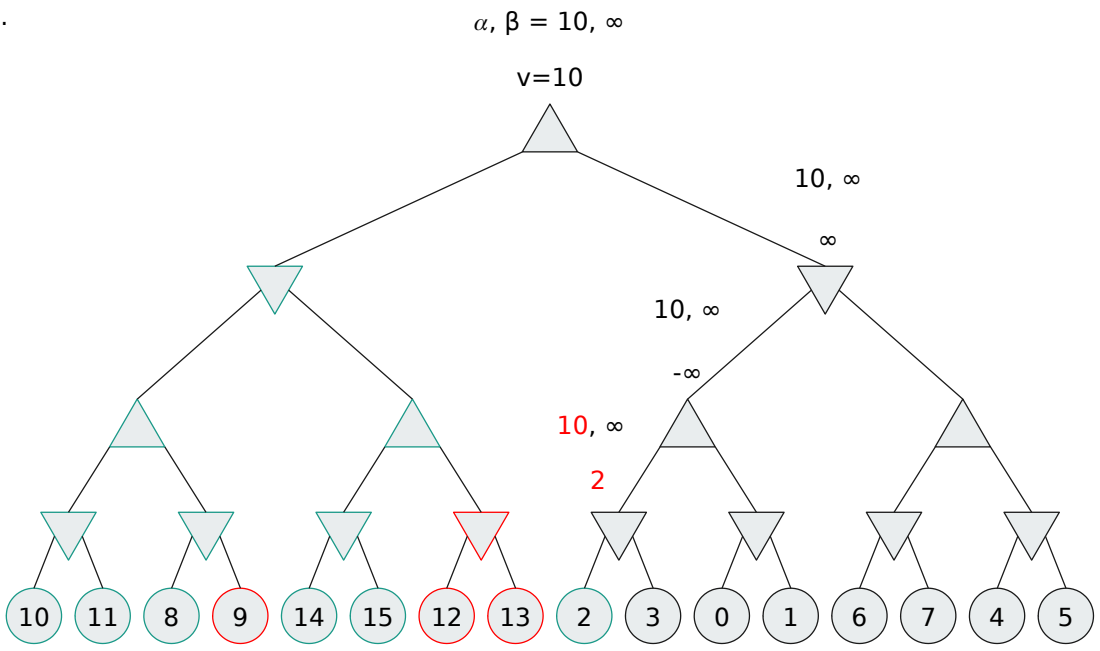
extra.



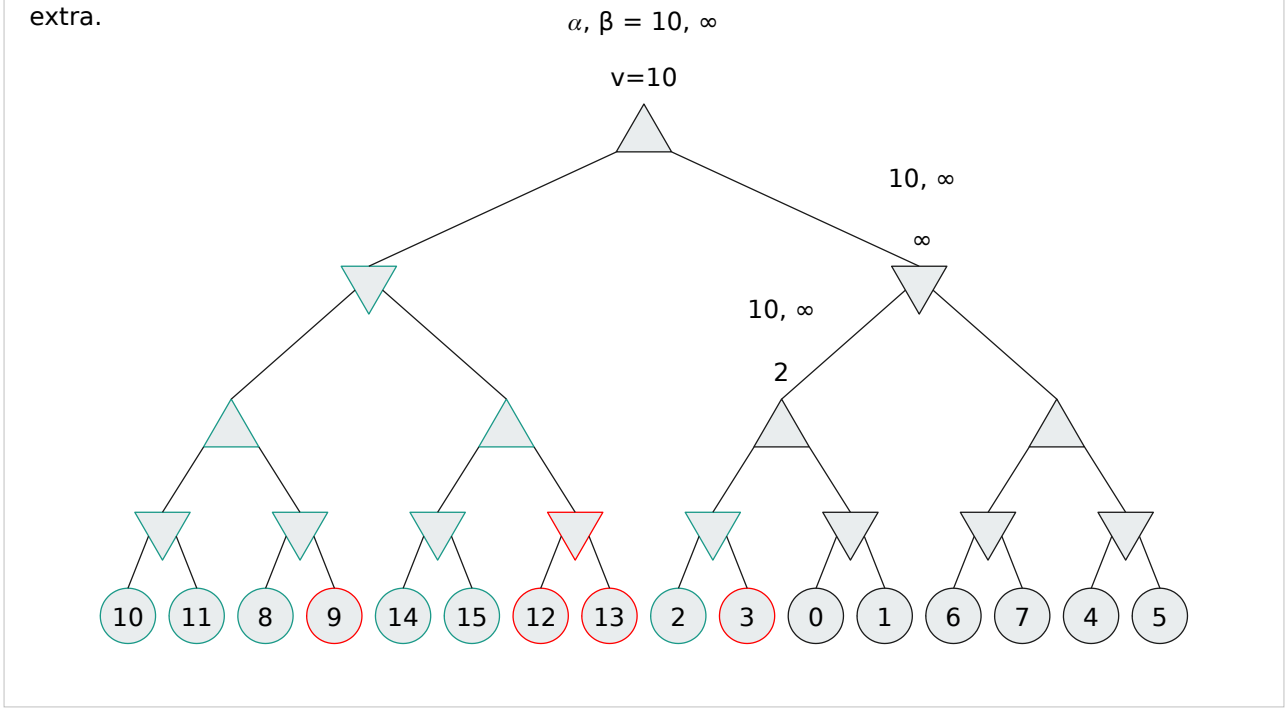
extra.



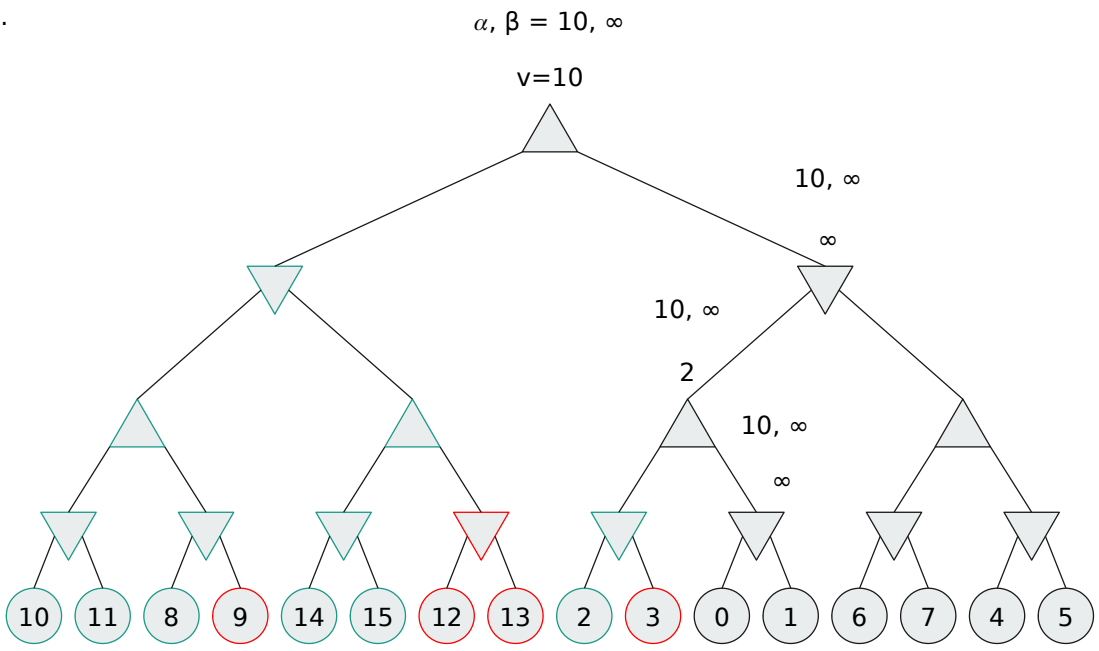
extra.



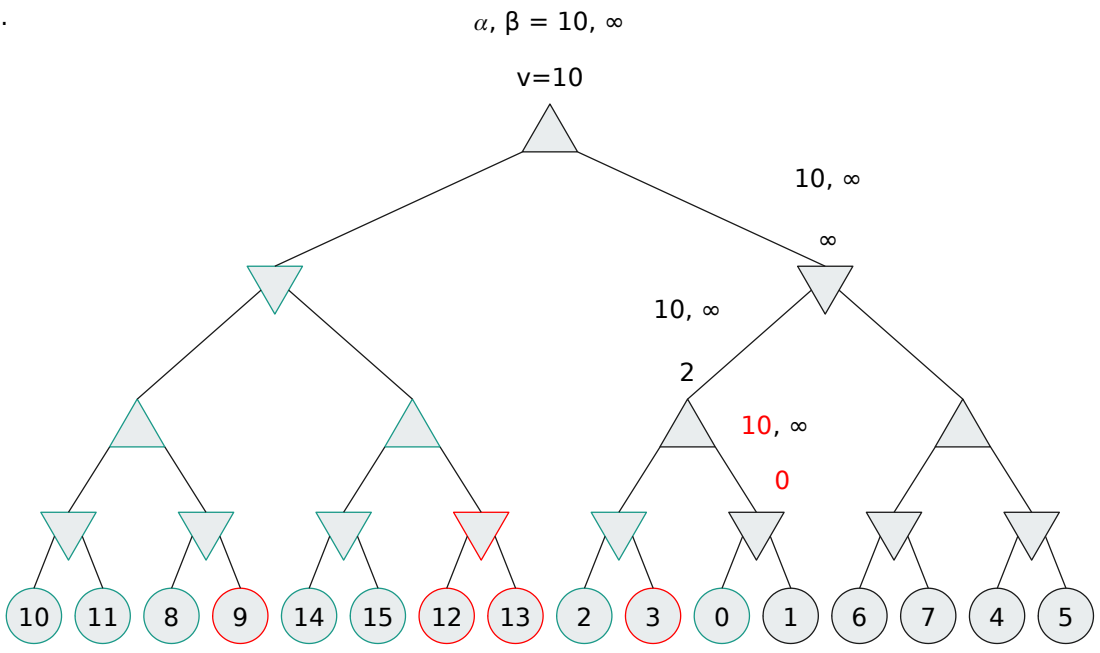
extra.



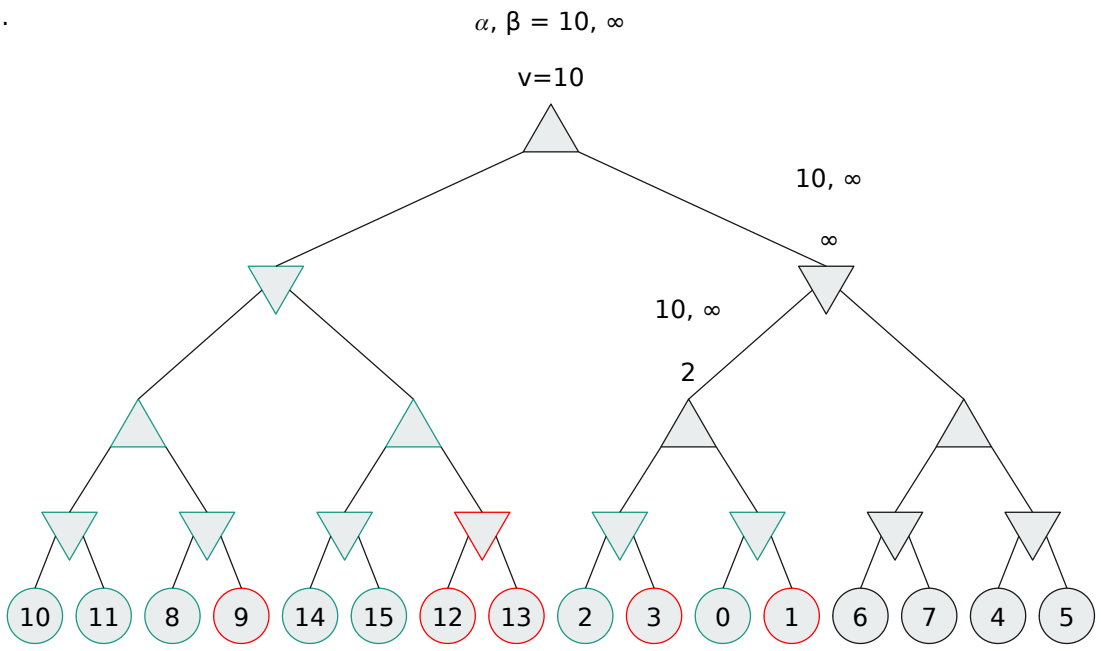
extra.



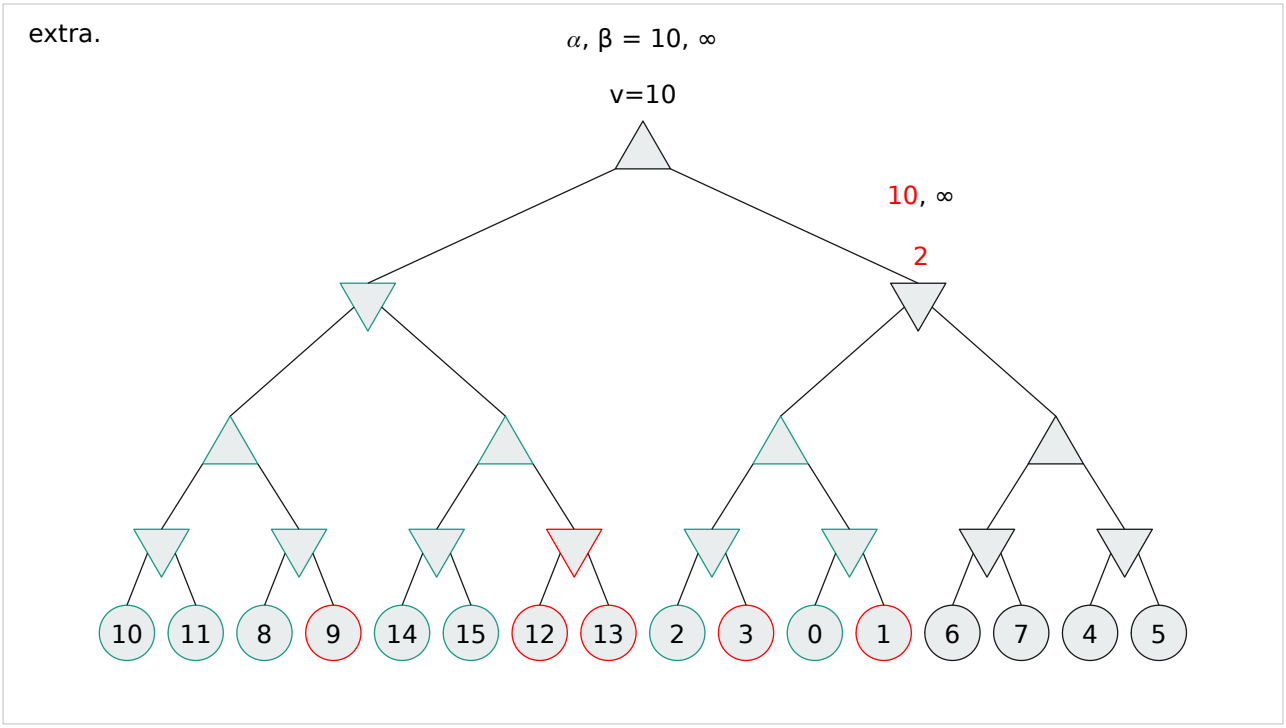
extra.



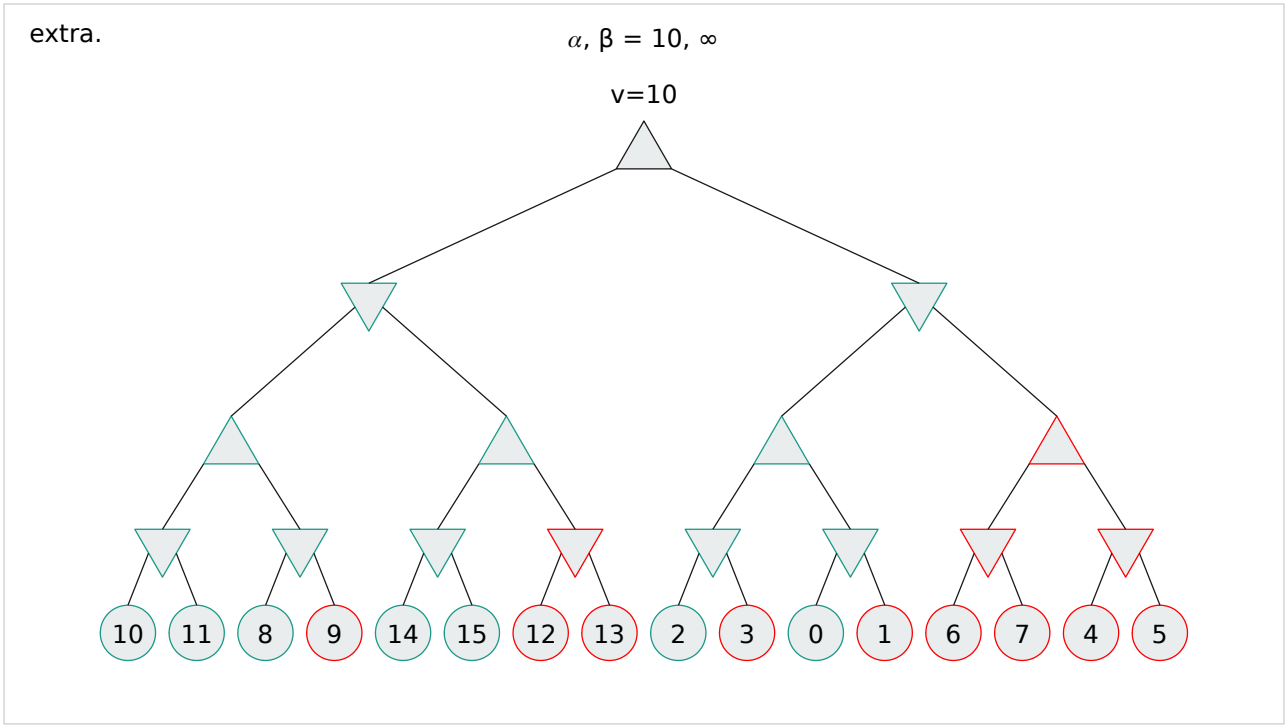
extra.



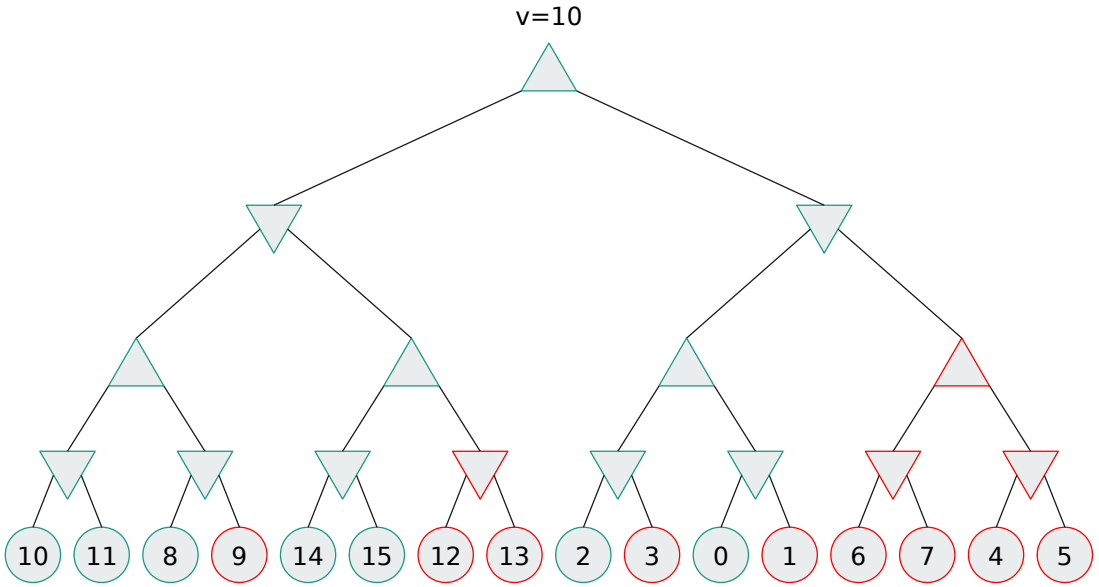
extra.



extra.



extra.



NEXT WEEK
QUIZ on “GAMES”