CS 461
Artificial Intelligence

Dr. Hashim Yasin

Adversarial Search

Adversarial Search

- **Competitive** environments, in which the agents' goals are in conflict, giving rise to **adversarial search** problems—often known as **games**

Why do AI researchers study game playing?

- It's a **good reasoning problem**, formal and nontrivial.
- Offer an opportunity to study problems involving **{hostile, adversarial, competing} agents**.
- **Direct comparison** with humans and other computer programs is easy.

Adversarial Search

Mainly games of strategy with the following characteristics:

- Sequence of **moves** to play
- Rules that specify **possible moves**
- Rules that specify a **payment** for each move
- Objective is to **maximize** your payment

Games

- **Compititve: Commonly Zero Sum**
  - One player wins and the other loses
  - A zero-sum game is defined as one where the **total payoff to all players is the same for every instance** of the game. Chess is zero-sum because every game has payoff of either 0 + 1, 1 + 0 or ½, ½.
- **Perfect Information:**
  - Players knows the results of the all previous moves
  - There is one best way to win the game for all players
- **Imperfect Information**
  - Players do not know all of the previous moves

Games

- **Initial State (s_0):** The initial state, which specifies how the game is *set up at the start*.
- **Players:** defines which player has the move in a state.
- **Actions:** The set of legal moves.
- **Result (s, a):** The transition model, which defines the result of a move. The state after action a is the state s.
- **Terminal Test:** A terminal test, which is true when the game is over and false otherwise.

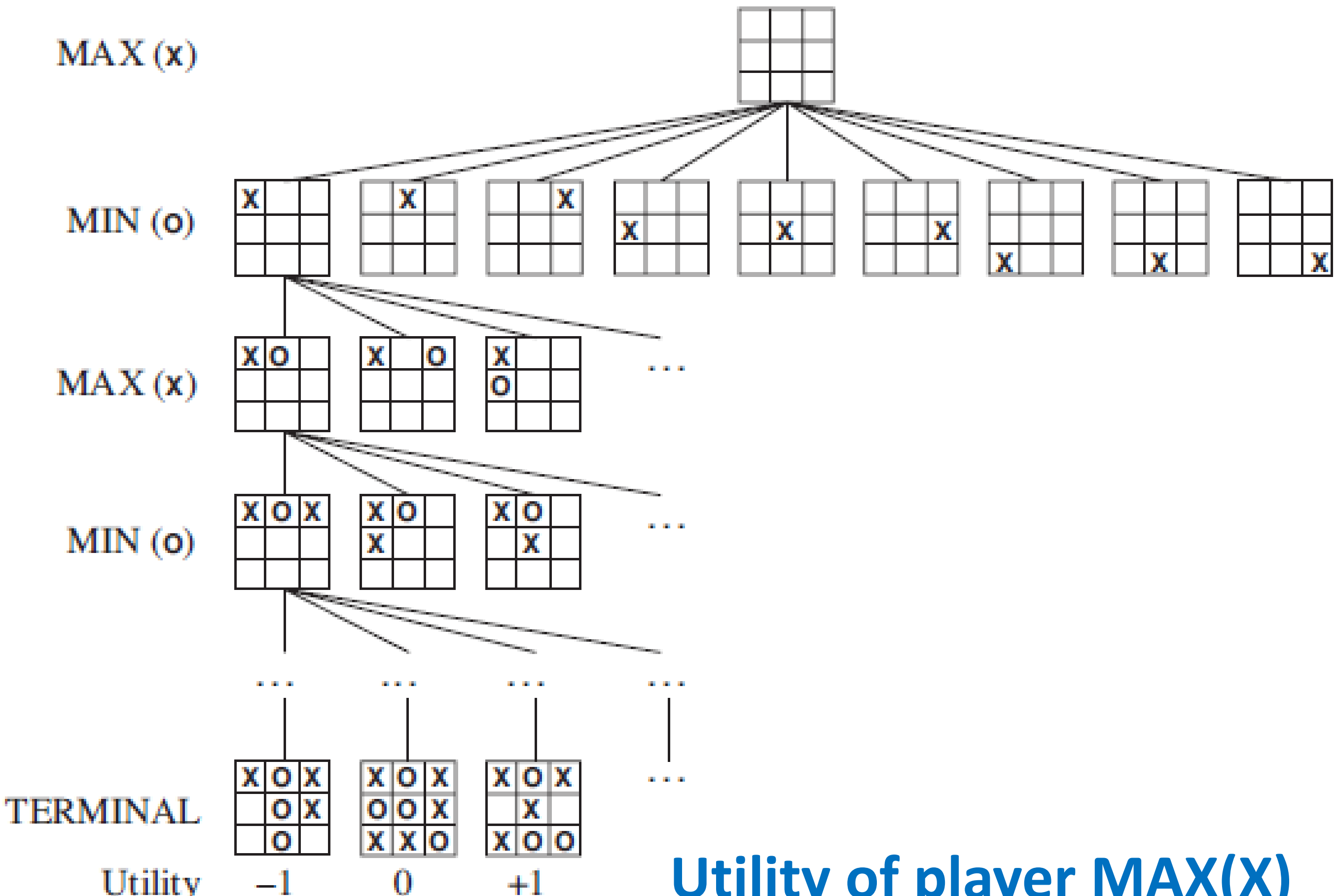
Games

- **Terminal State:** States where the game has ended are called terminal states.
- **Utility:** A utility function (also called an **objective function** or **payoff function**), defines the final numeric value for a game that ends in terminal state s for a player p.
  - In chess, the outcome is a win, loss, or draw, with values +1, 0, or ½.
  - Some games have a wider variety of possible outcomes; the payoffs in **backgammon range from 0 to +192**.

Game Tree

- The **initial state**, **actions**, and **results** define the game tree for the game.
- A **game tree** where the **nodes are the game states** and **the edges are moves**.
- The game tree is best thought of as a theoretical construct that we cannot realize in the physical world.
  - For **tic-tac-toe** the game tree is relatively small—fewer than 9! = 362, 880 terminal nodes.
  - For **chess** there are over 10^{40} nodes,

Game Tree: tic-tac-toe



Utility of player MAX(X)

Minimax

- Minimax is a method used **to evaluate game trees**.
- Given a game tree, the optimal strategy can be determined from the **minimax value** of each node.
- A static evaluator is applied to leaf nodes, and **the values are passed back up the tree to determine the best score** the computer can obtain against a rational opponent.

Minimax

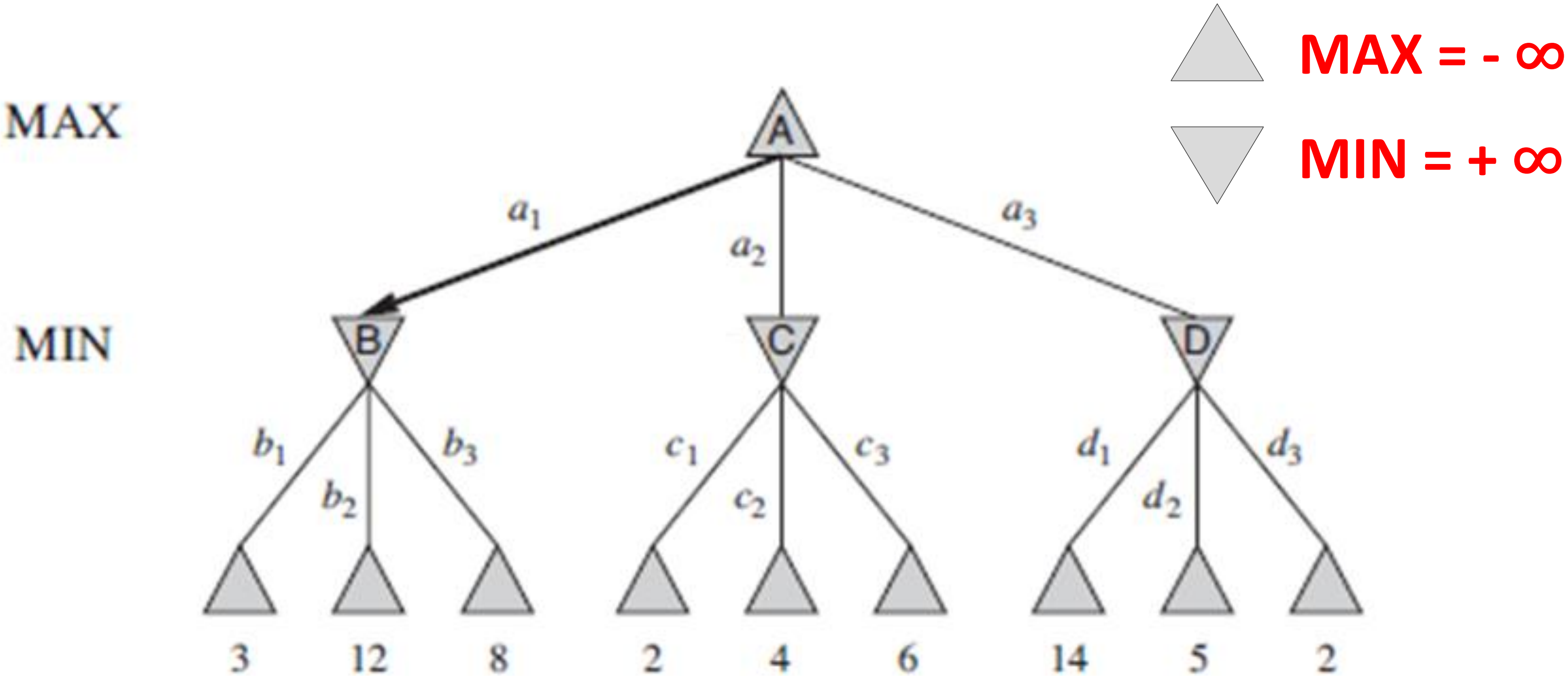
MAX

- Wants to **maximize** the result of the utility function
- ***Winning strategy*** if, on MIN's turn, a win is obtainable for MAX for all moves that MIN can make

MIN

- Wants to **minimize** the result of the utility function
- ***Winning strategy*** if, on MAX's turn, a win is obtainable for MIN for all moves that MAX can make

Minimax

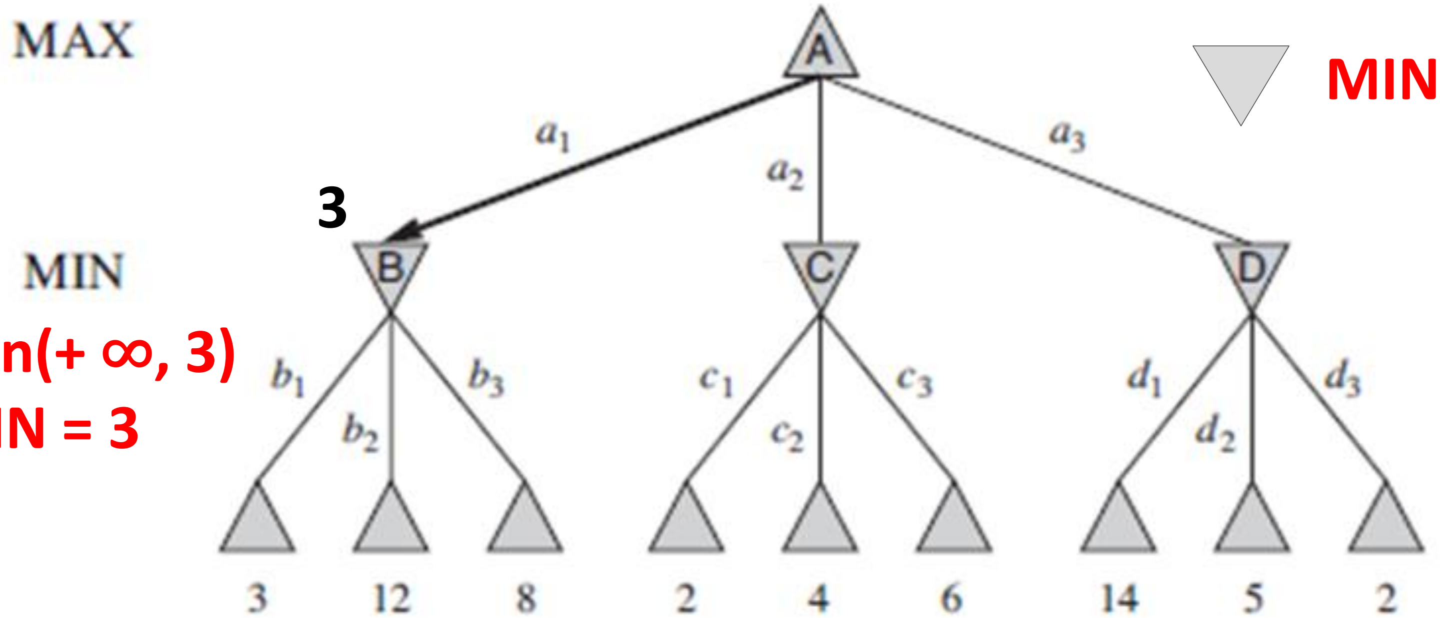


$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

Minimax

MAX = - ∞

MIN = + ∞

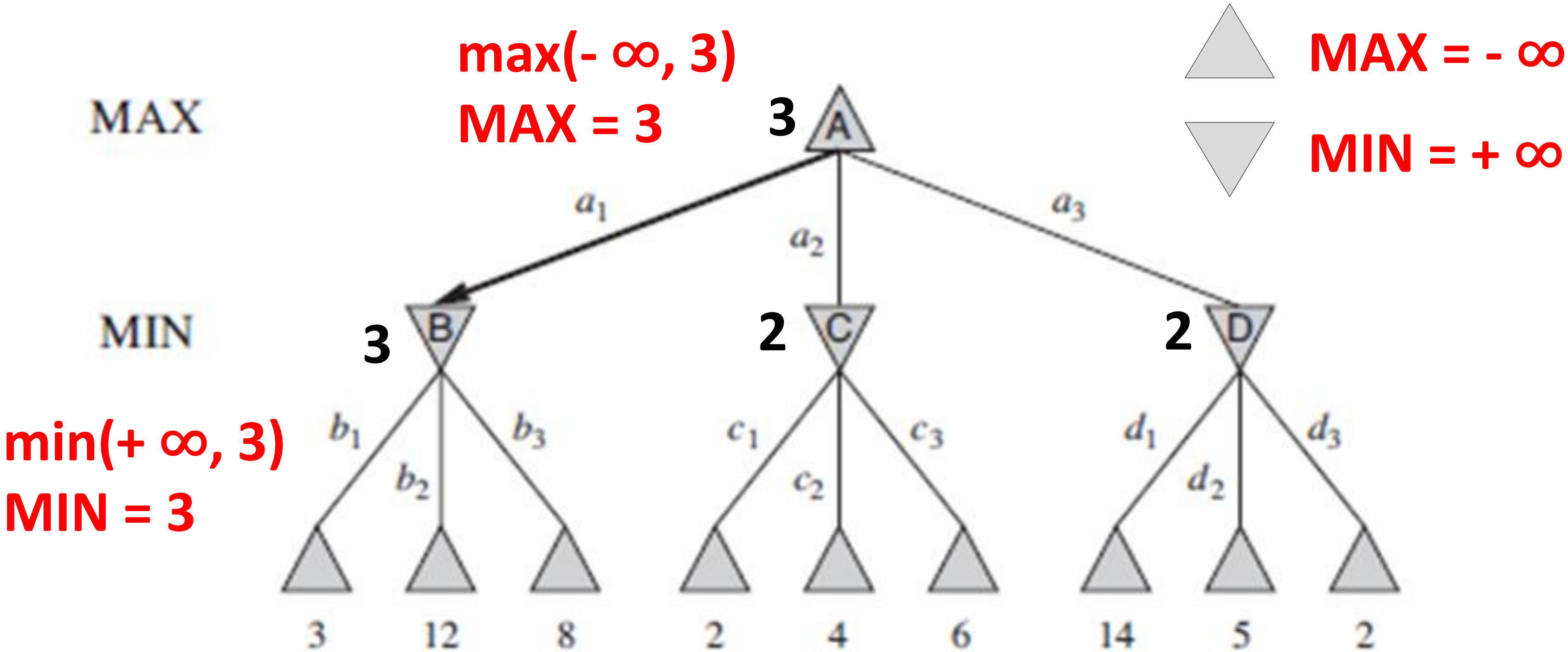


min(+ ∞, 3)

MIN = 3

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

Minimax



$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

Minimax

function MINIMAX-DECISION(*state*) **returns** *an action*
  return $\arg\max_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT(*state*, *a*))

---

function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

---

function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
  return v

Properties of Minimax

Complete?

- Yes (if tree is finite).

Optimal?

- Yes

Time complexity?

- $O(b^m)$, m is the maximum depth of the tree and b is the legal moves.

Space complexity?

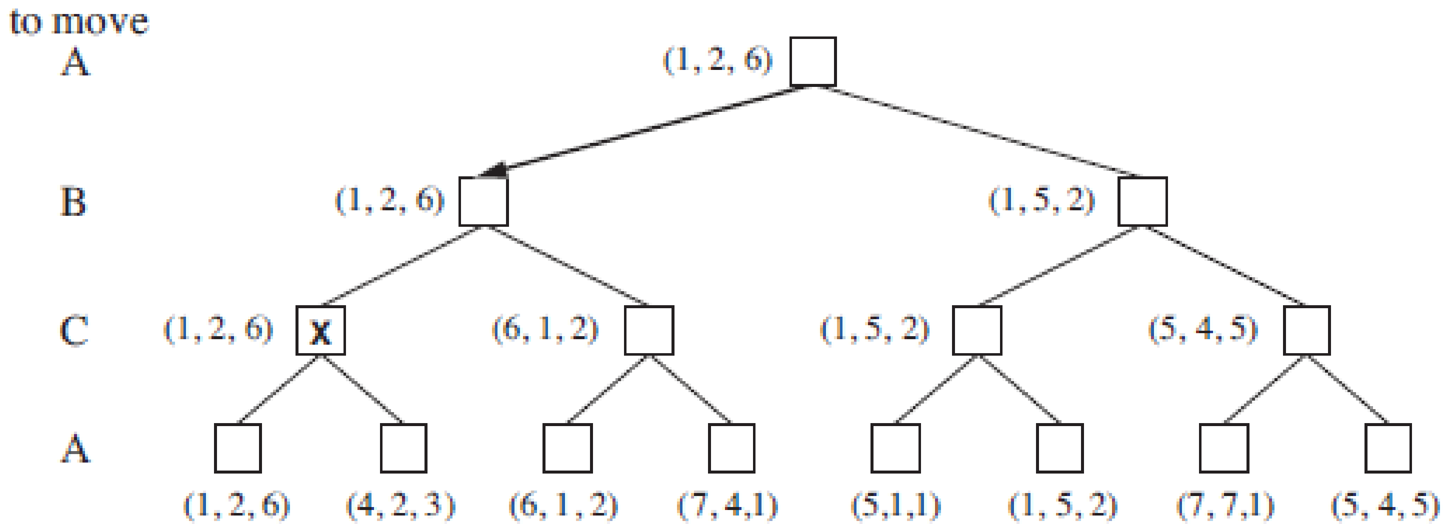
- $O(bm)$
  - (depth-first search, generate all actions at once)
- $O(m)$
  - (backtracking search, generate actions one at a time)

Multiplayer Games

- Each node must hold **a vector of values**

For example, for three players $\boldsymbol{A, B, C\ (VA, VB, VC)}$

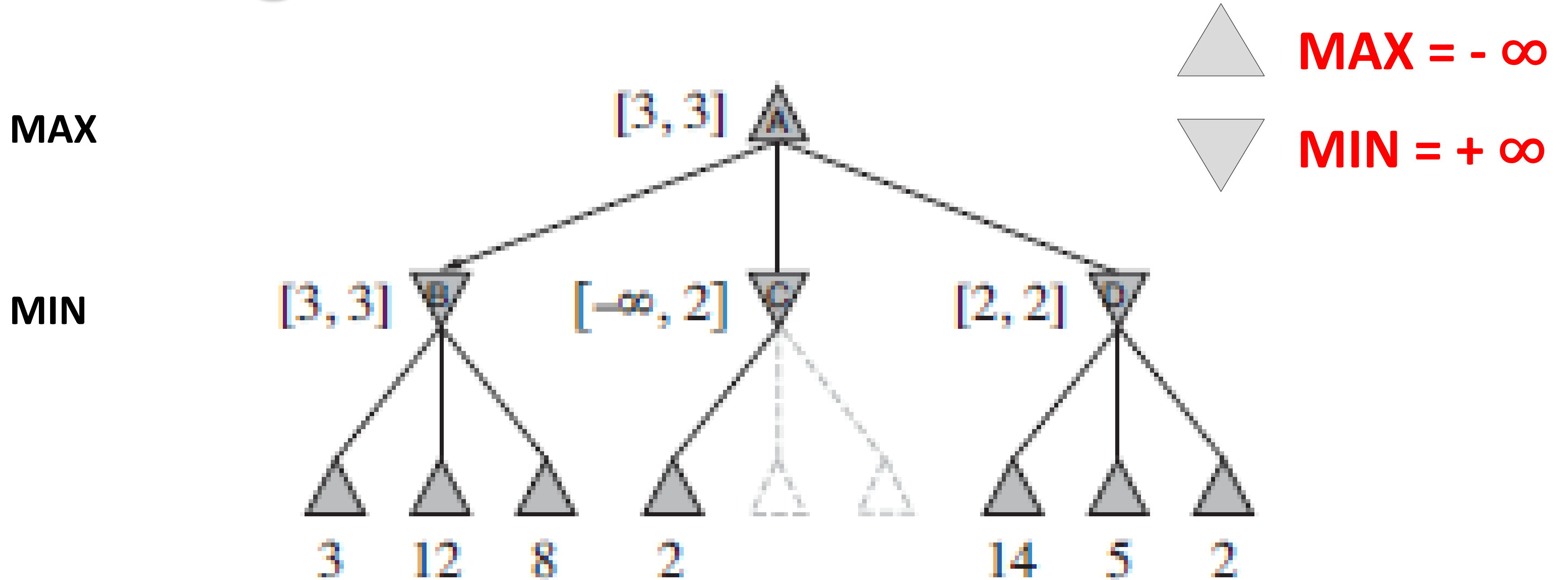
- The backed up vector at node $\boldsymbol{n}$ *will always be the one that* ***maximizes the payoff*** *of the player choosing at* $\boldsymbol{n}$



Searching Game Trees

- **Exhaustively searching** a game tree is not usually a good idea.
- Even for a simple game like
  - **tic-tac-toe** there are over **350,000 nodes** in the complete game tree.
- **An additional problem** is that the computer only gets to choose every other path through the tree and the opponent chooses the others.

Pruning



$$
\begin{aligned}
\textsc{Minimax}(root) &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\
&= \max(3, \min(2,x,y), 2) \\
&= \max(3, z, 2) \qquad \text{where } z = \min(2,x,y) \leq 2 \\
&= 3.
\end{aligned}
$$

Do we need *z*?

Pruning

- We can use a **branch-and-bound technique** to reduce the number of states that must be examined to determine the value of a tree.

Branch-and-bound Technique:

- We keep track of a **lower bound on the value of a maximizing node**, and don't bother evaluating any trees that cannot improve this bound.
- Keep track of an **upper bound on the value of a minimizing node.** Don't bother with any sub-trees that cannot improve this bound.

Minimax with Alpha-Beta Cutoffs

Alpha Cutoffs:

- Alpha is the ***lower bound on maximizing nodes***.

Beta Cutoffs:

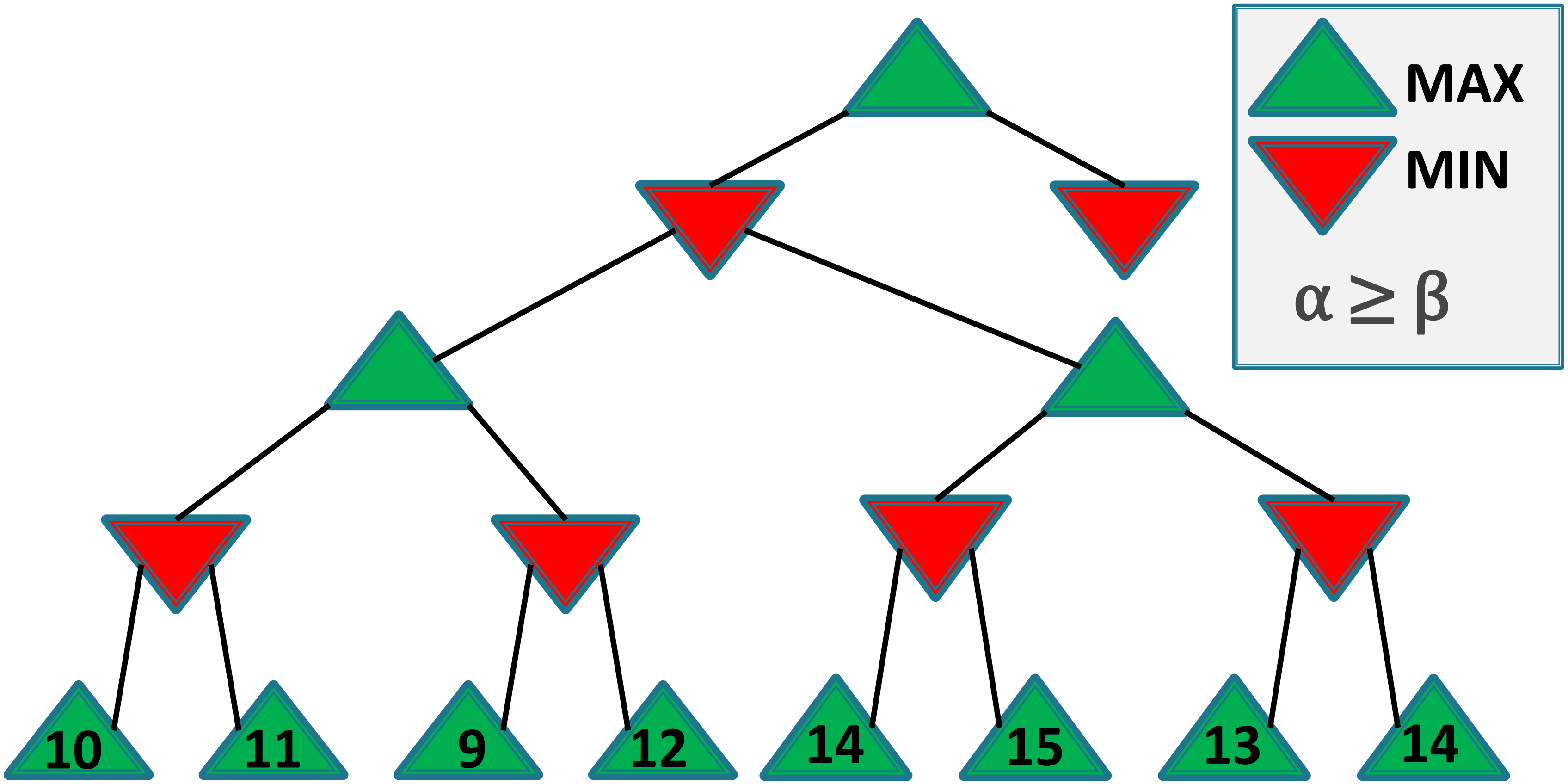
- Beta is the ***upper bound on minimizing nodes***.

- Both alpha and beta get passed down the tree during the Minimax search.

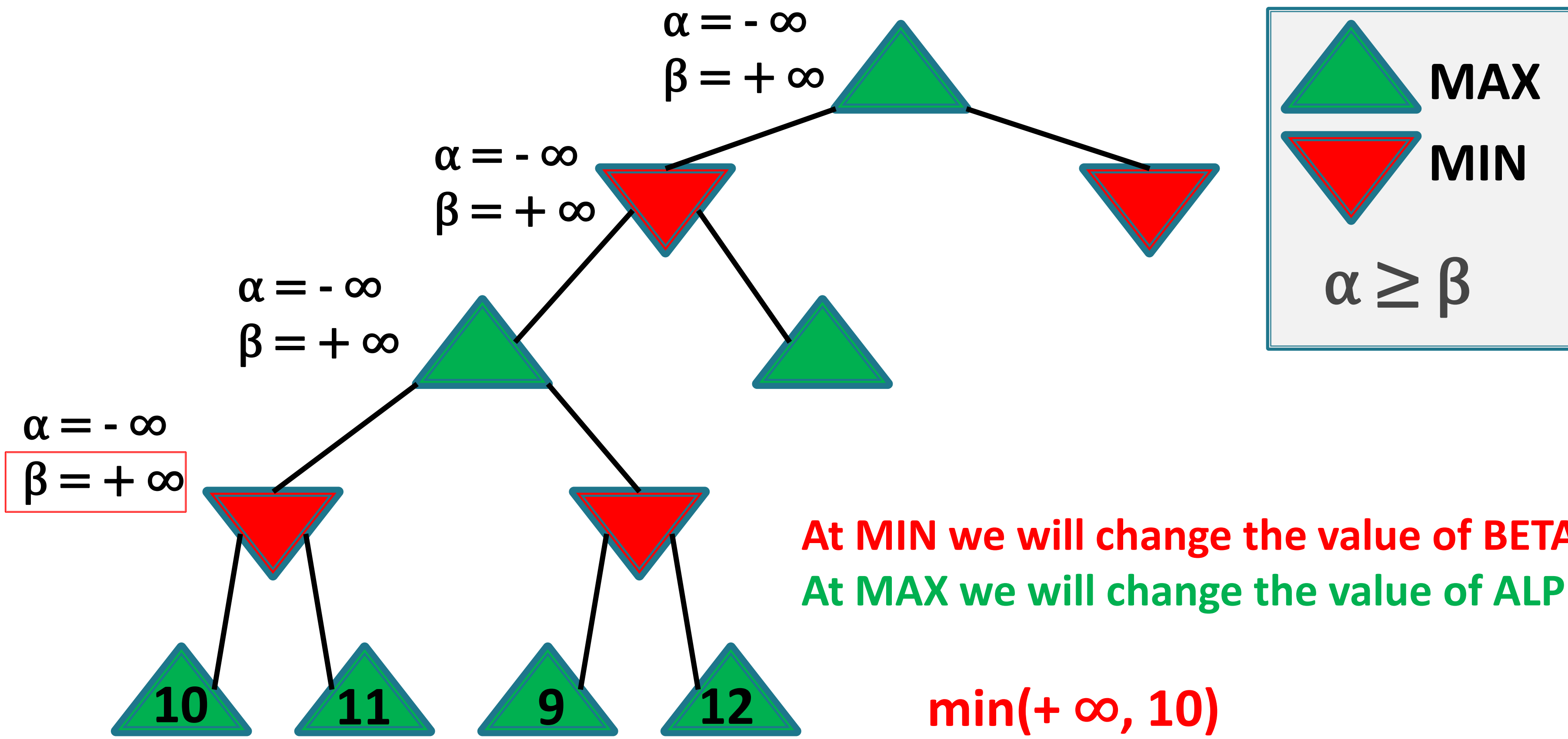
Minimax with Alpha-Beta Cutoffs

- **At minimizing nodes**, we stop evaluating children if we get a child whose value is **less than the current lower bound** ***(alpha)***.
- **At maximizing nodes**, we stop evaluating children as soon as we get a child whose value is **greater than the current upper bound** ***(beta)***.
- Some branches will never be played by rational players since they include sub-optimal decisions (for either player)

Alpha-Beta Pruning: Example



Alpha-Beta Pruning: Example

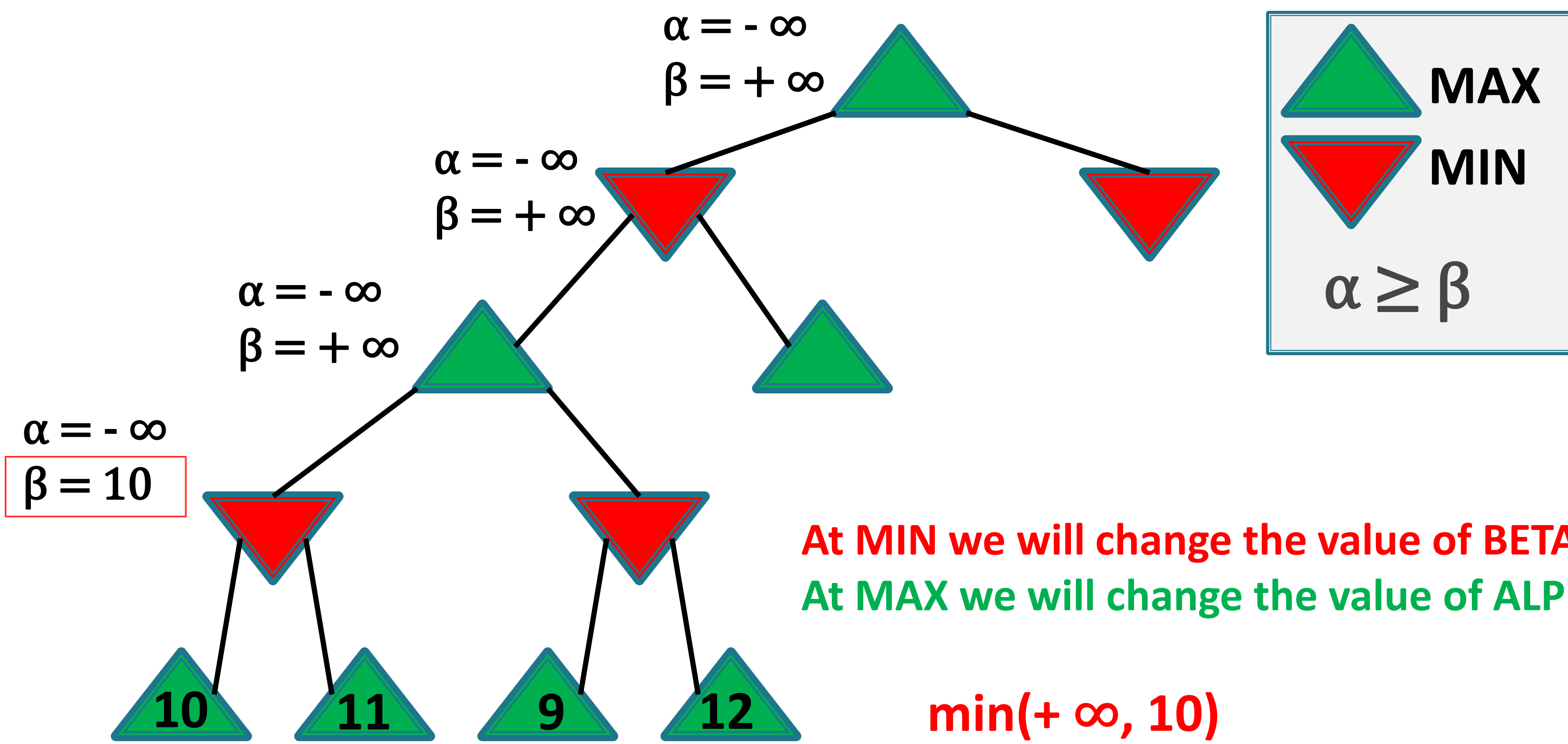


At MIN we will change the value of BETA

At MAX we will change the value of ALPHA

min(+ ∞, 10)

Alpha-Beta Pruning: Example

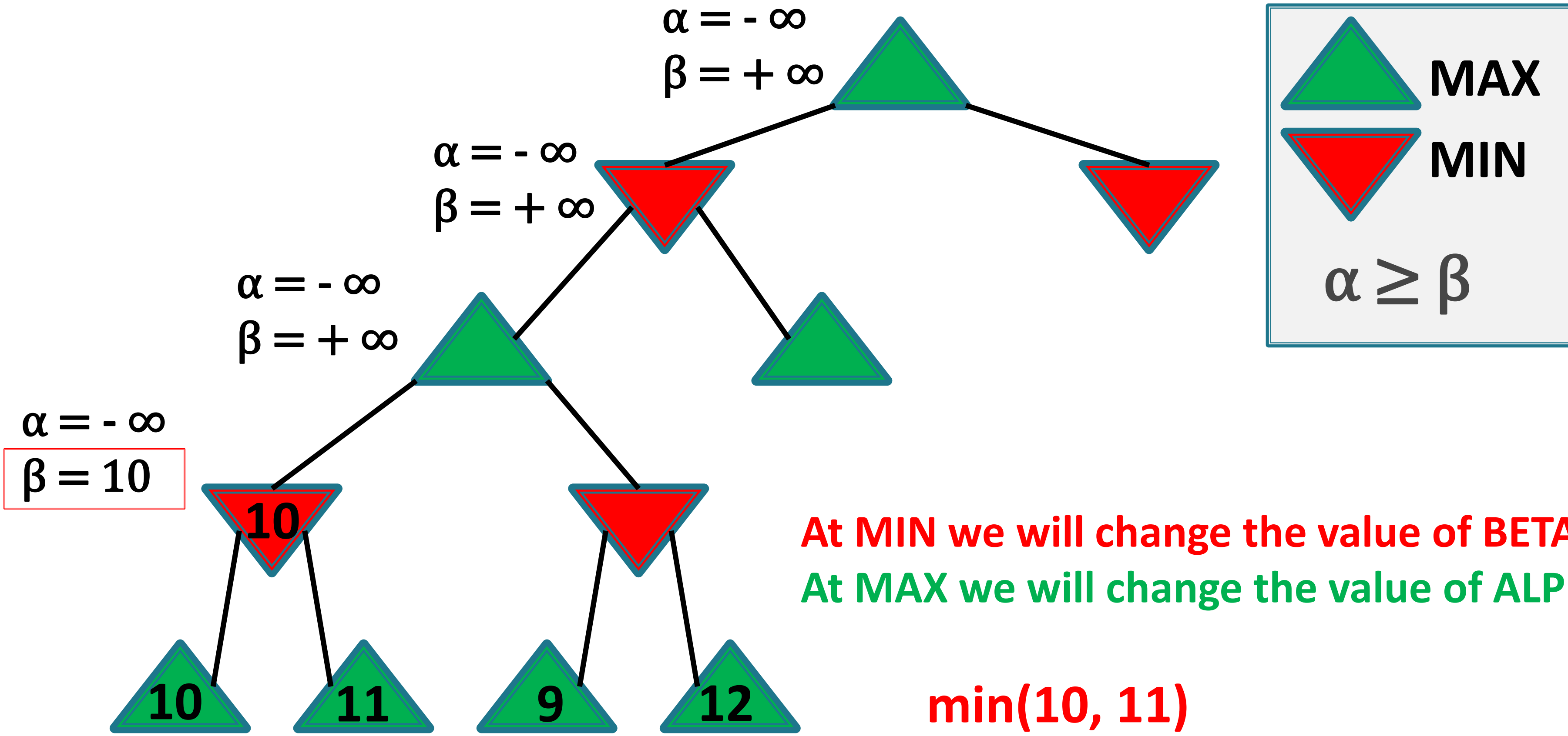


At MIN we will change the value of BETA

At MAX we will change the value of ALPHA

min(+ ∞, 10)

Alpha-Beta Pruning: Example

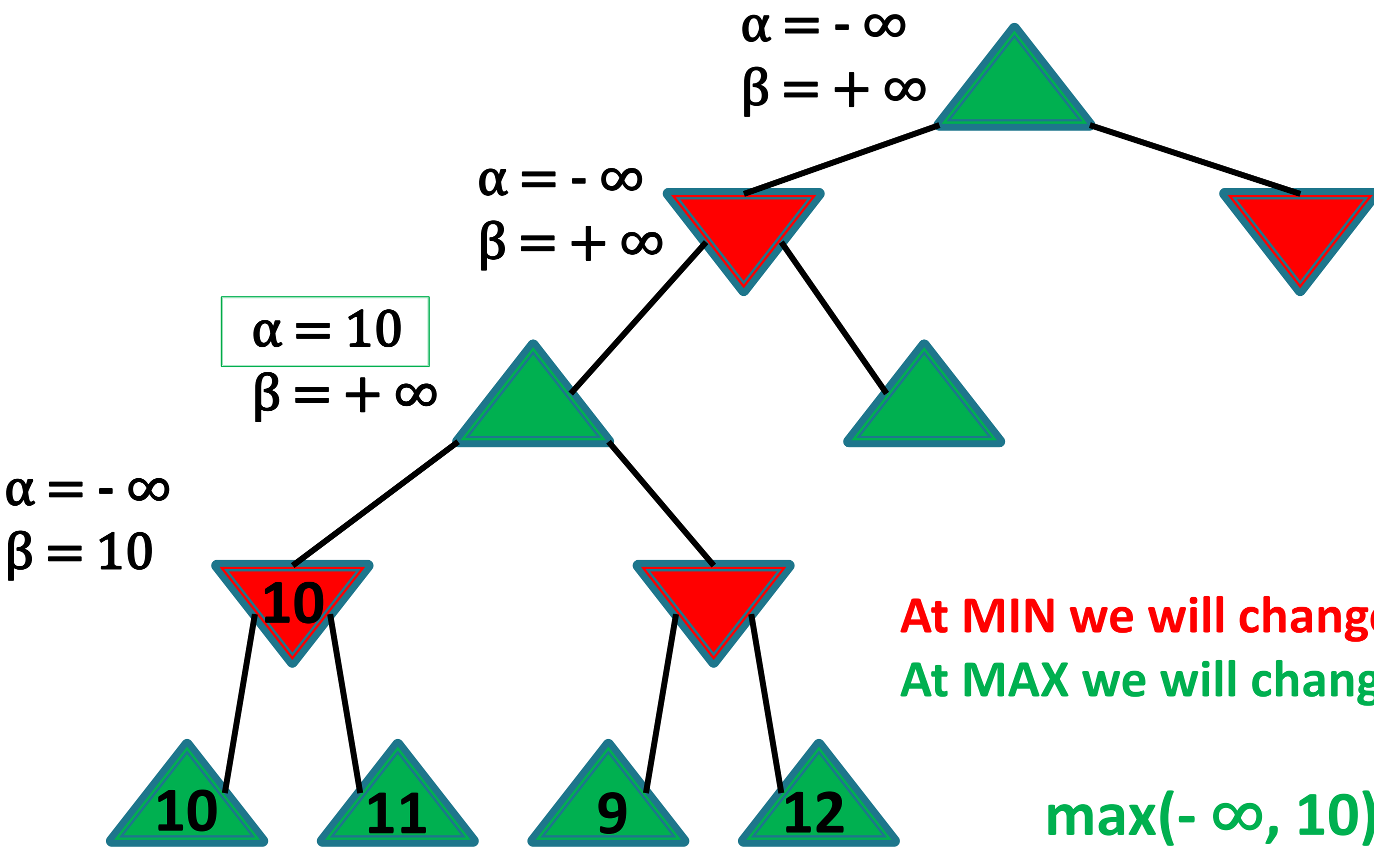


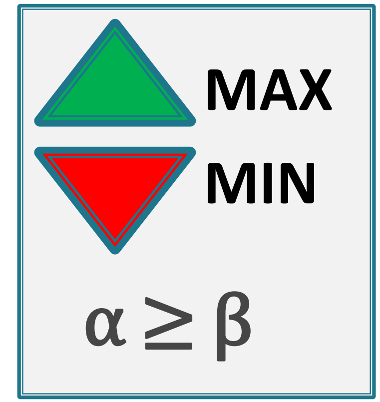
At MIN we will change the value of BETA

At MAX we will change the value of ALPHA

min(10, 11)

Alpha-Beta Pruning: Example



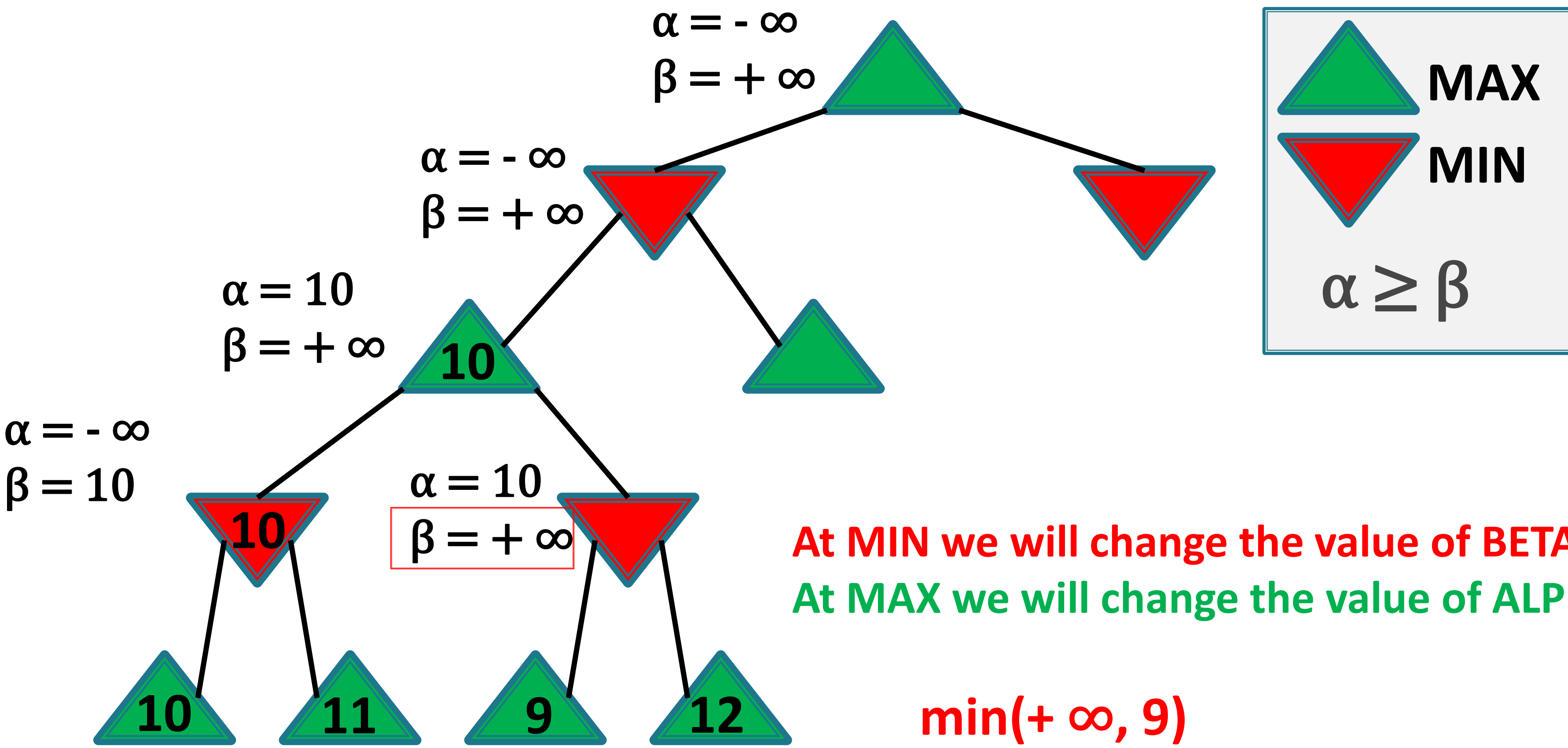


At MIN we will change the value of BETA

At MAX we will change the value of ALPHA

max(- ∞, 10)

Alpha-Beta Pruning: Example

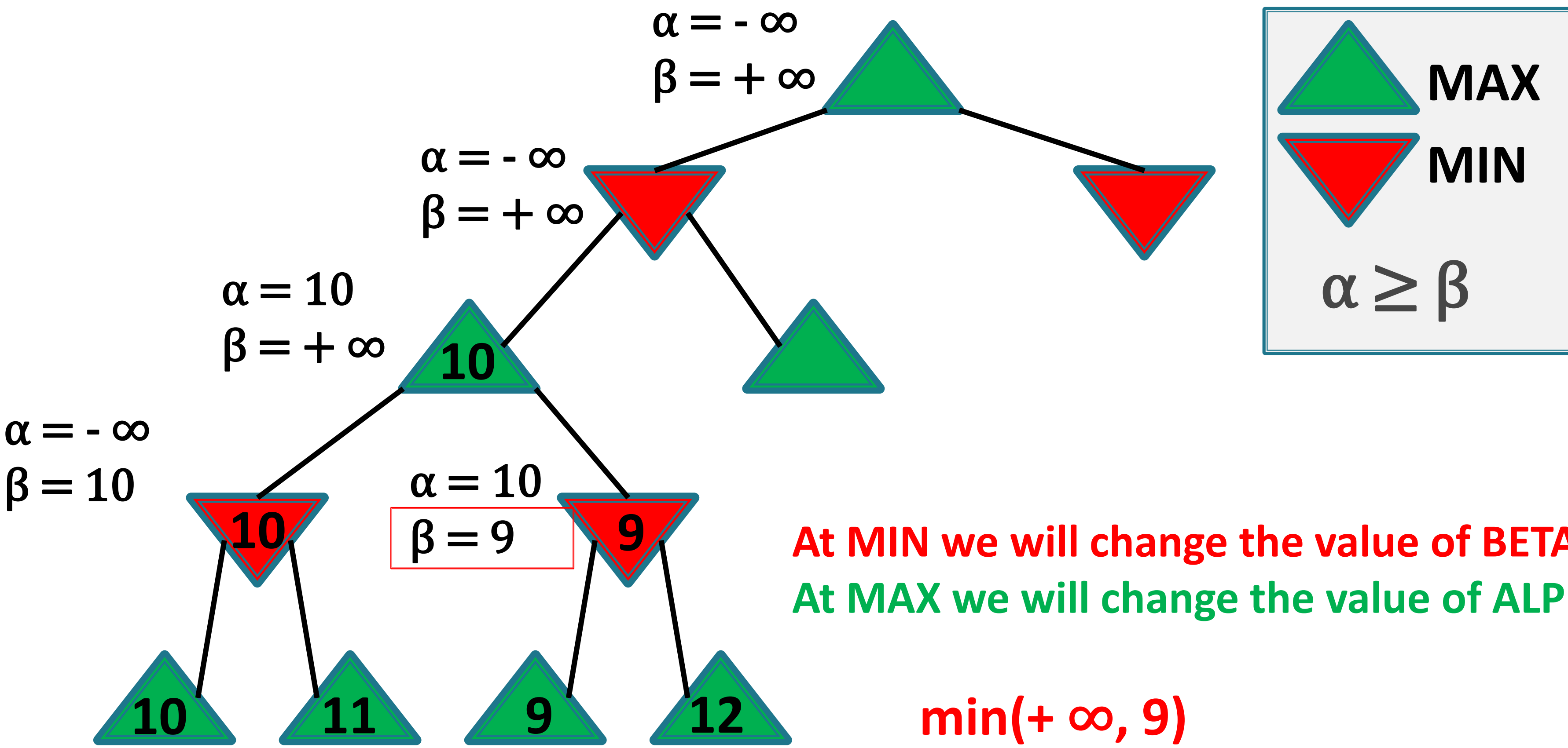


At MIN we will change the value of BETA

At MAX we will change the value of ALPHA

min(+ ∞, 9)

Alpha-Beta Pruning: Example

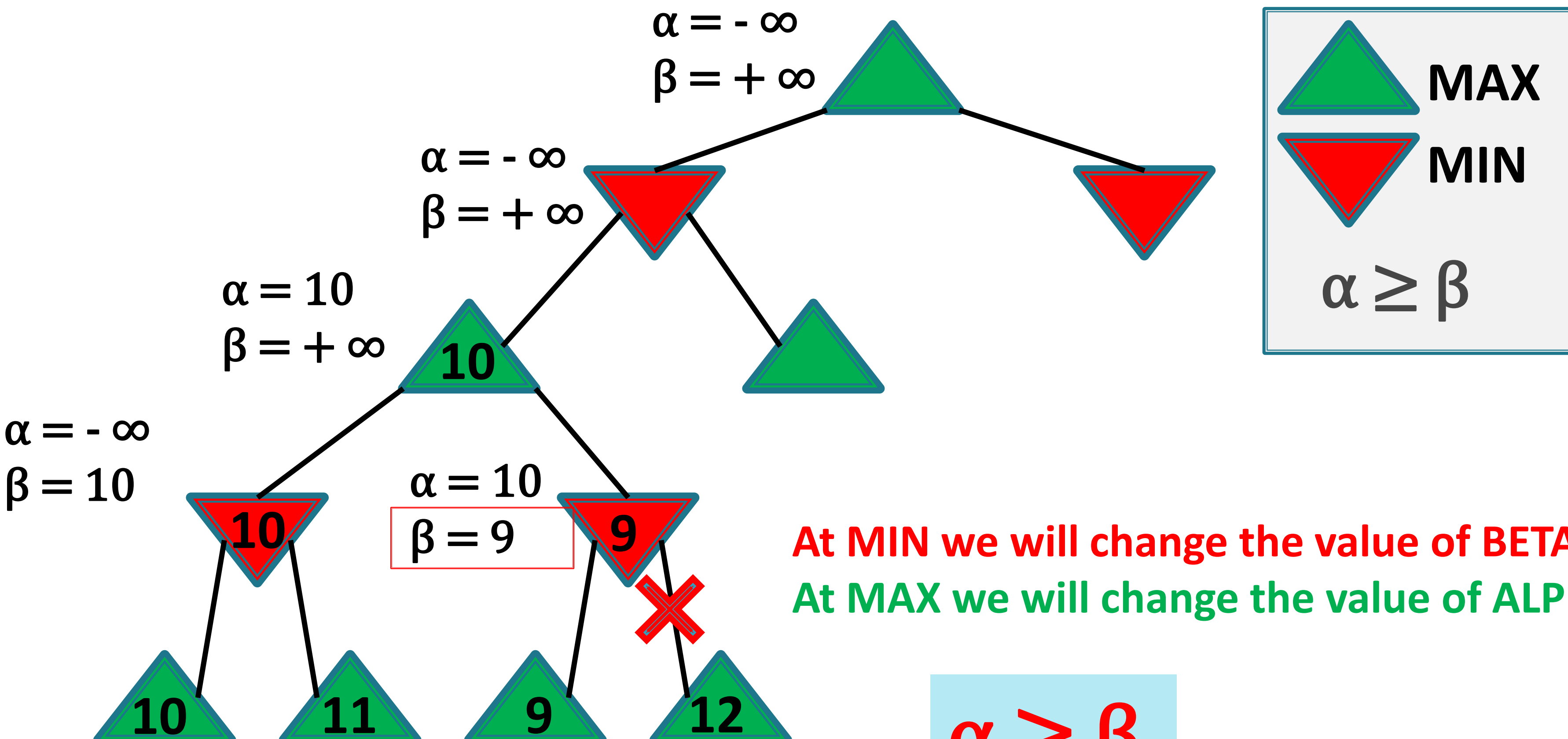


At MIN we will change the value of BETA

At MAX we will change the value of ALPHA

min(+ ∞, 9)

Alpha-Beta Pruning: Example

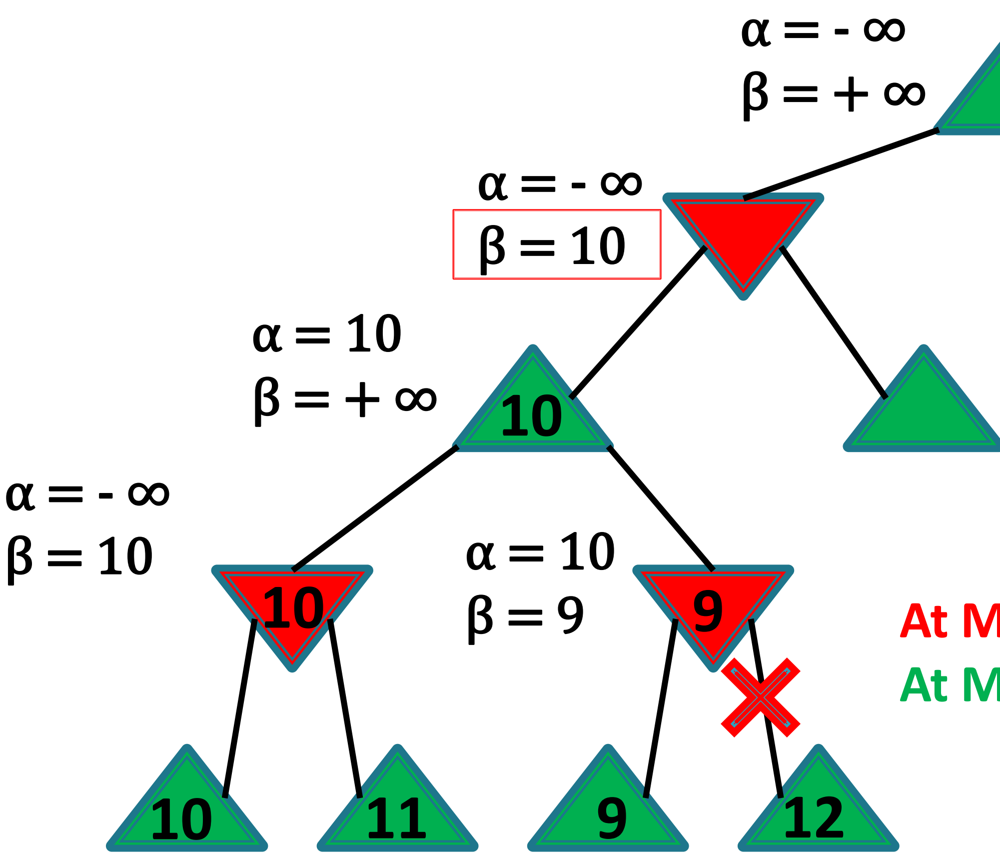


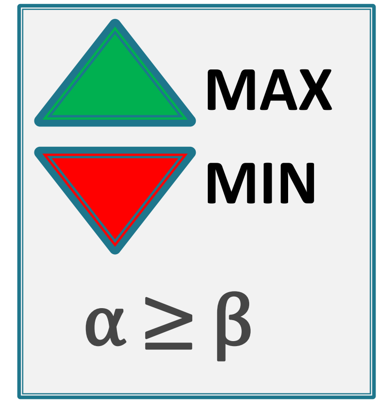
At MIN we will change the value of BETA

At MAX we will change the value of ALPHA

$\alpha \geq \beta$

Alpha-Beta Pruning: Example



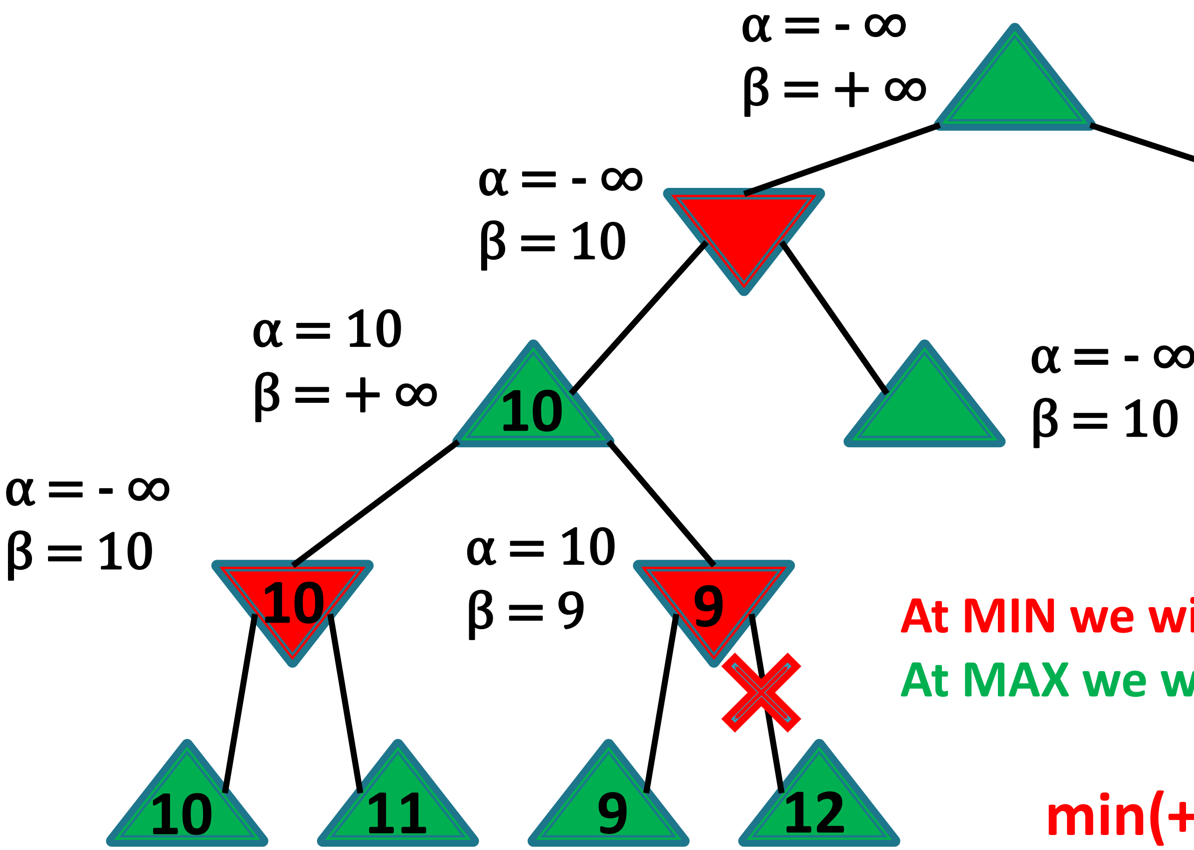


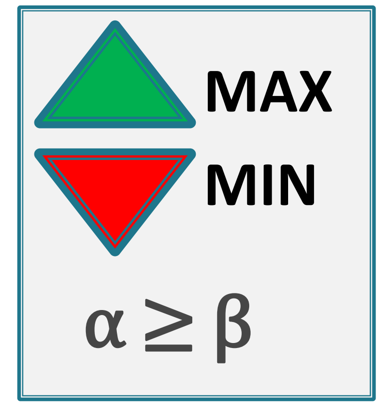
At MIN we will change the value of BETA

At MAX we will change the value of ALPHA

min(+ ∞, 10)

Alpha-Beta Pruning: Example





At MIN we will change the value of BETA

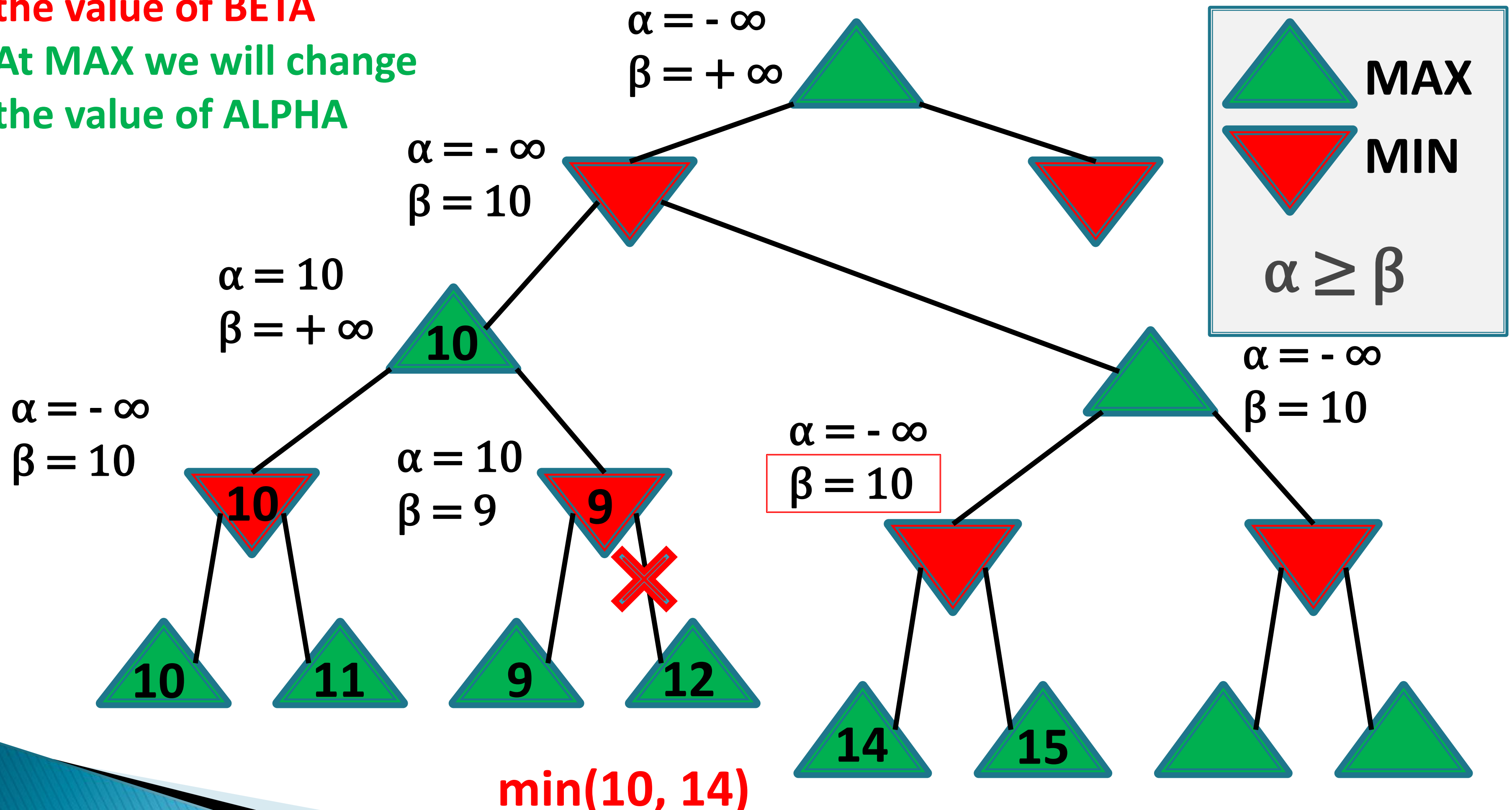
At MAX we will change the value of ALPHA

min(+ ∞, 10)

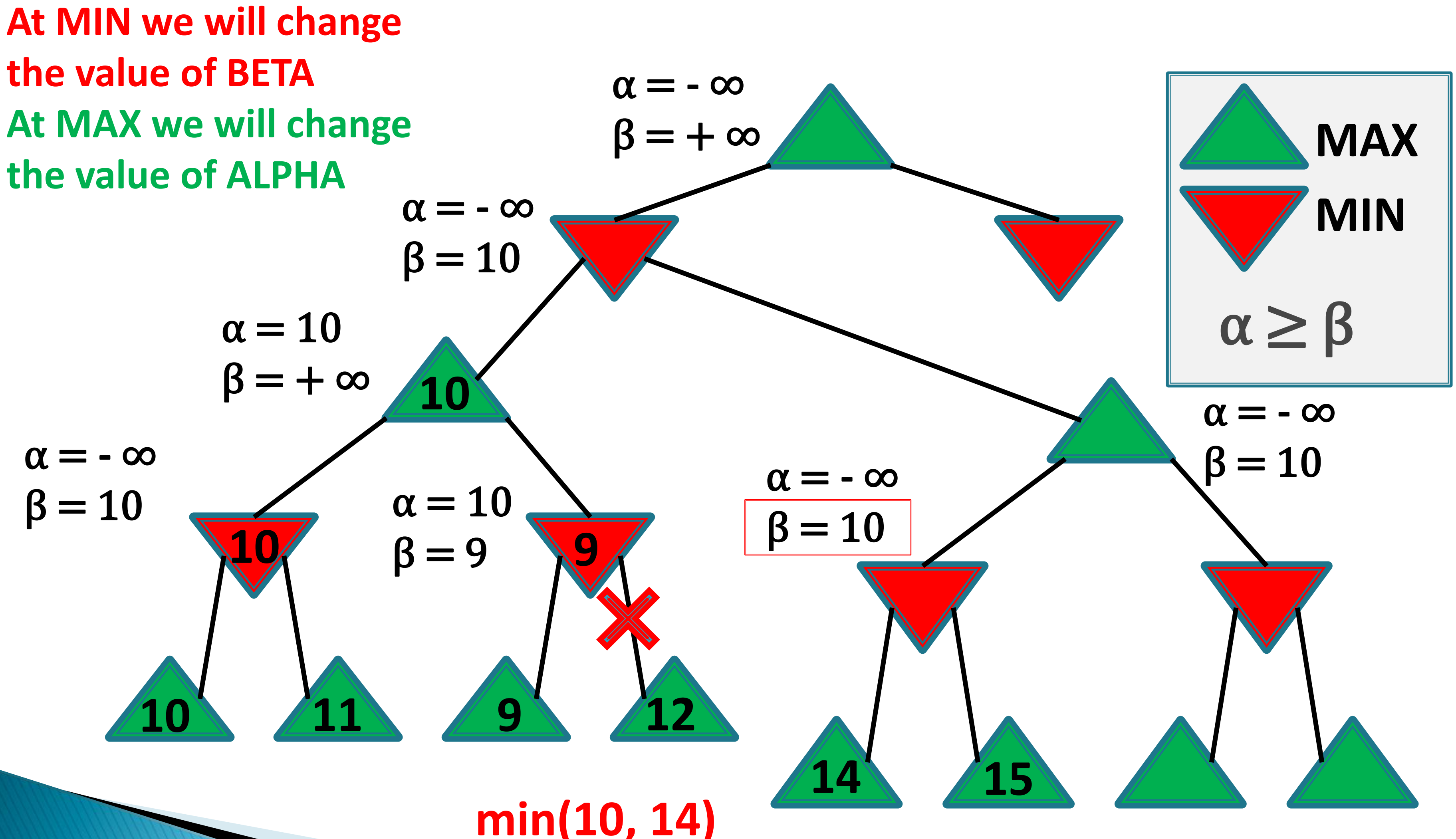
Alpha-Beta Pruning: Example

At MIN we will change the value of BETA

At MAX we will change the value of ALPHA



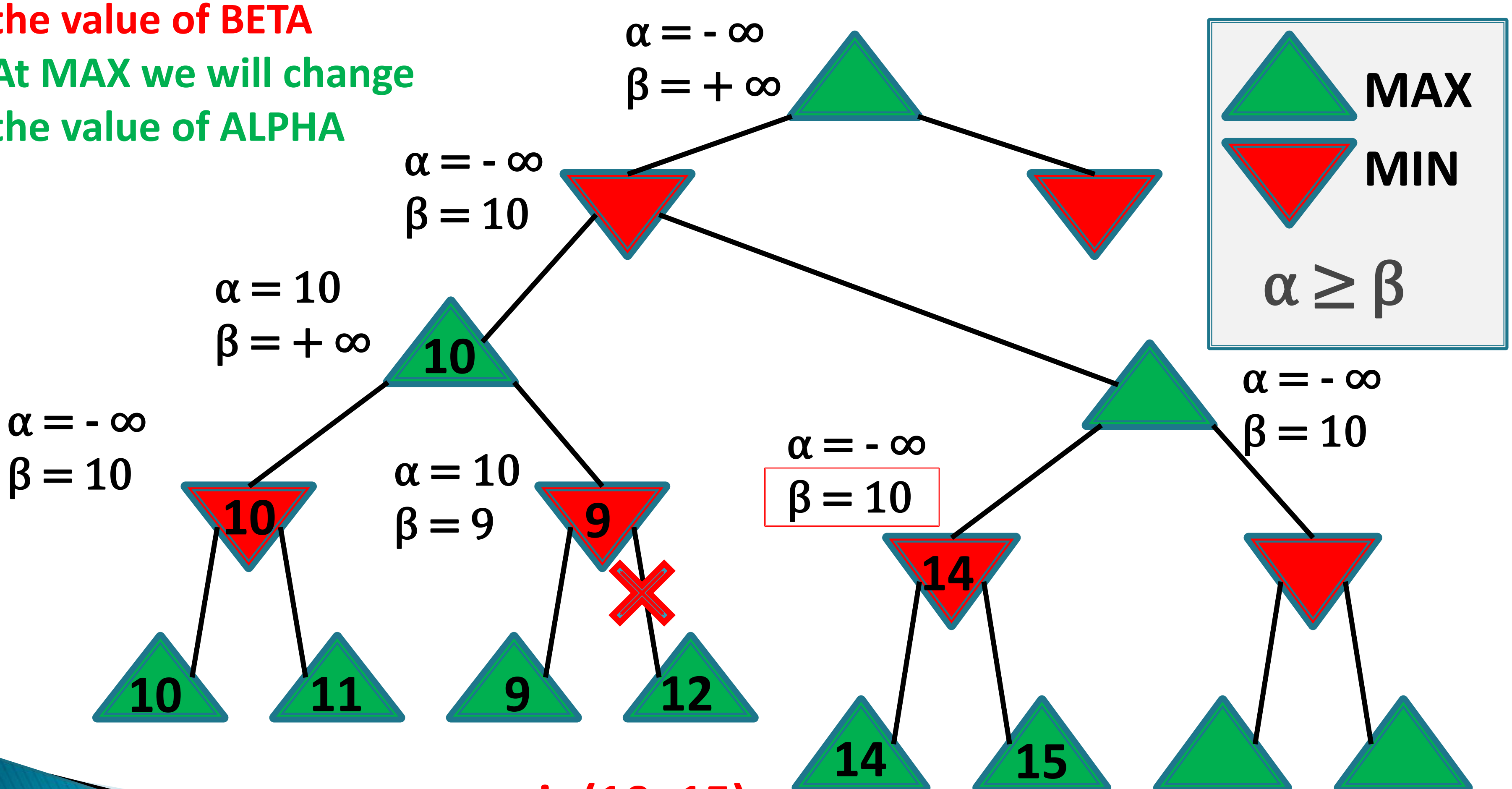
Alpha-Beta Pruning: Example



Alpha-Beta Pruning: Example

At MIN we will change the value of BETA

At MAX we will change the value of ALPHA

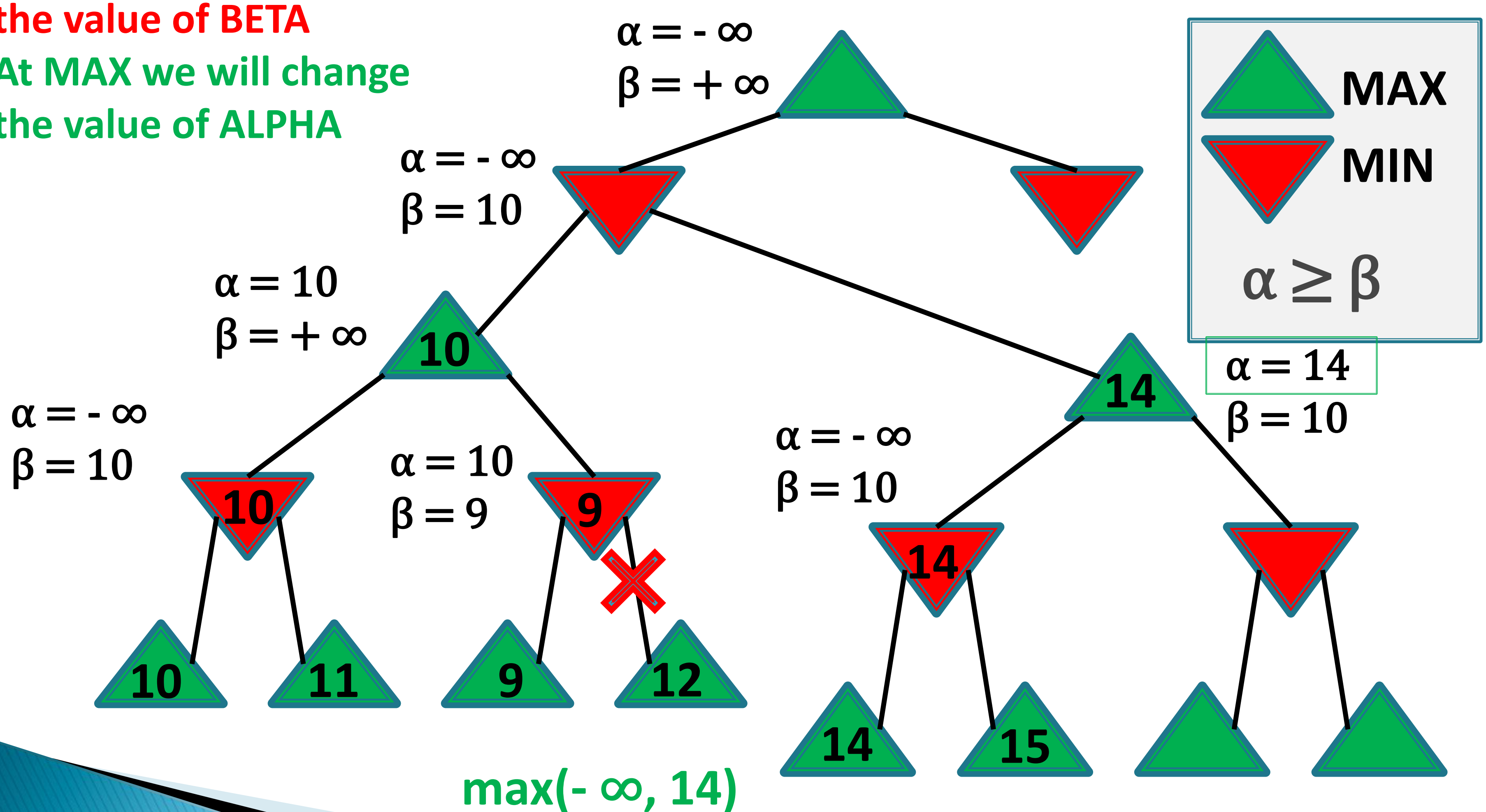


min(10, 15)

Alpha-Beta Pruning: Example

At MIN we will change the value of BETA

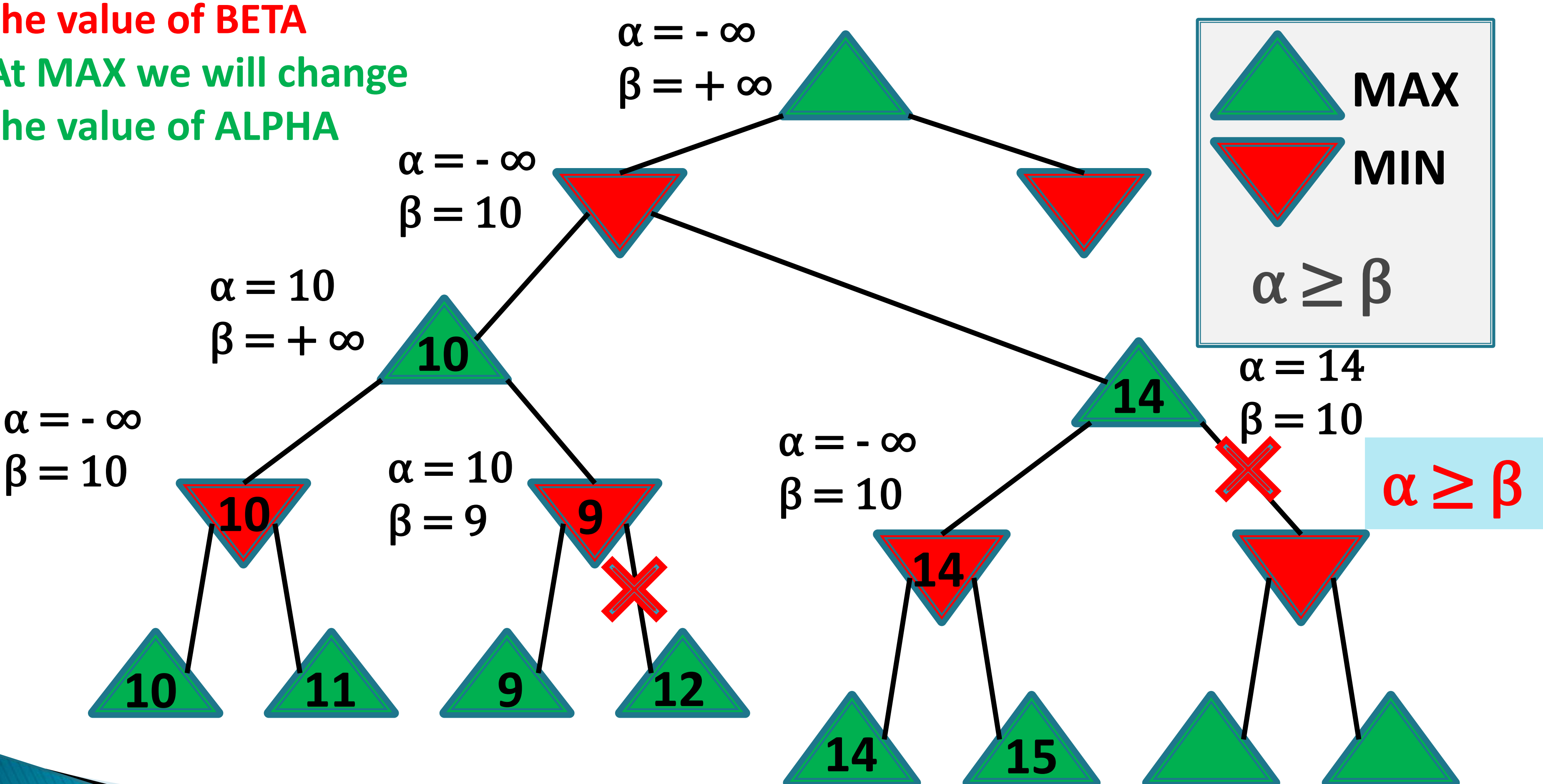
At MAX we will change the value of ALPHA



Alpha-Beta Pruning: Example

At MIN we will change the value of BETA

At MAX we will change the value of ALPHA



Alpha-Beta Pruning: Effectiveness

- The effectiveness **depends on the order** in which children are visited.

- **In the best case**, the effective branching factor will be reduced from $\boldsymbol{b}$ *to* $\boldsymbol{sqrt(b)}$.

- **In an average case** (random values of leaves) the branching factor is reduced to $\frac{\boldsymbol{b}}{\log \boldsymbol{b}}$.

Reading Material

- **Artificial Intelligence,** A Modern Approach

  Stuart J. Russell and Peter Norvig
  - **Chapter 5.**