# COMP 3200
# Artificial Intelligence


**Lecture 9**
Two-Player Games
Mini-Max Search
Alpha-Beta Pruning

# Multi-Player Games

- So far, we have only looked at problems with a single agent (single agent search)
- Heuristic search can also be applied to environments (and games) with multiple agents (or players)
- Games can have a number of properties, similar to single agent environments

# Fully vs. Partially Observable

# Deterministic vs. Stochastic

# Dynamic vs. Static


# Game Properties: Players

- How many players are in the game?
- Examples of Game Players:
  - Chess, Go: 2 Players
  - Baseball: 18 Players, 2 Teams
  - Starcraft: 1v1, 2v2, 8 Person FFA
  - Poker: 10 People at a table

# Game Properties: Payoffs

- What does each player hope to achieve?
- Game Payoff Examples:
  - Prisoner's Dilemma: Maximize Reward
  - Poker: Maximize Profits
  - Chess: Win the game
- Zero Sum Game:
  - Each player's gain or loss of payoff/utility is equally balanced by the utility of the other players
  - Win / Lose games are zero sum

# Most Traditional Board Games

- Two-Player
- Zero Sum
- Alternating Move
- Perfect Information
- Deterministic
- Discrete

# The Chess-Playing Computer

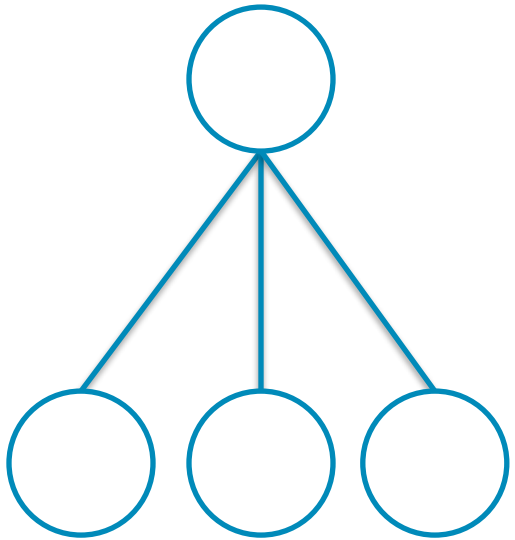

# Game-Playing Computer

- How can we design an algorithm to play a two-player alternating move game?

1. Analysis / Strategy / Tactics?
2. Thousands of If-Then Statements?
3. Try Actions (Look-Ahead) and Evaluate?

# Look-Ahead and Evaluate

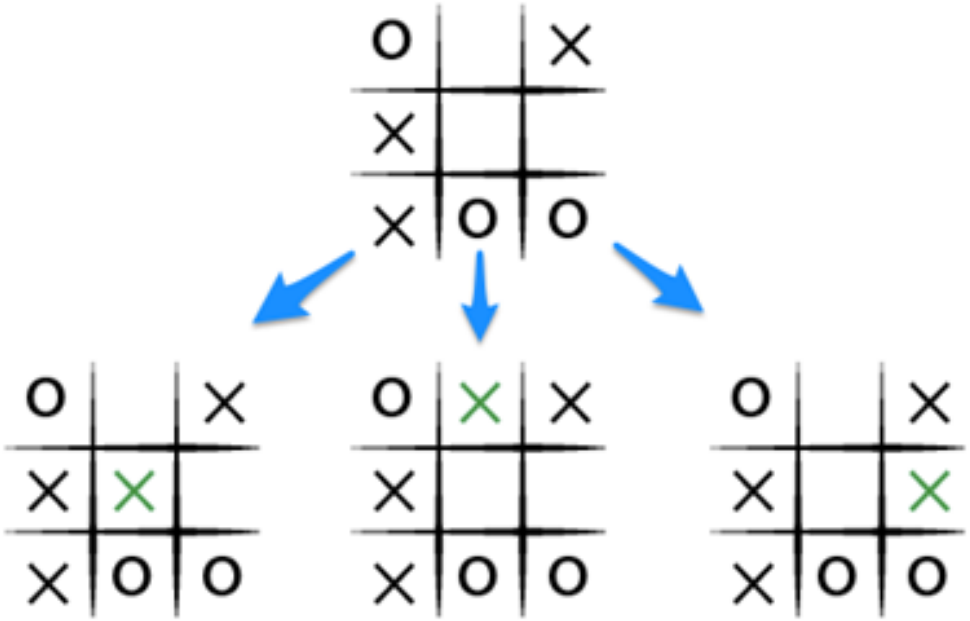
- Generate a list of actions from a given state
- Evaluate those actions based on features of the resulting states
- Do the action which has the highest evaluation



# Lookahead and Evaluate

```
1.  Function LookaheadAndEvaluate(state)
2.      maxVal = -infinity
3.      bestAction = null
4.      for (action : state.getLegalActions())
5.          child = state.doAction(action)
6.          val = eval(child)
7.          if (val > maxVal)
8.              maxVal = val
9.              bestAction = action
10.     return bestAction
```

# Tic-Tac-Toe Example



# Game-Playing Computer

- How can we design an algorithm to play a two-player alternating move game?

1. Analysis / Strategy / Tactics
2. Thousands of If-Then Statements
3. Try Actions (Look-Ahead) and Evaluate
4. Search the Entire Game Tree?

# How big is the game tree?

- Some number of actions at each state
  - b = “Branching Factor”
- Game ends after some number of moves
  - d = Search Depth
- Game tree = $b^d$

# State / Action Space

- State Space
  - Number of possible configurations of the environment for a given problem
- Action Space
  - Number of actions possible from a state
  - Given as average or worst-case
- Used as an estimation of complexity

Source: https://www.pipmodern.com/post/complexity-state-space-game-tree

# Tic-Tac-Toe

## State-Space Complexity Calculation

# Chess

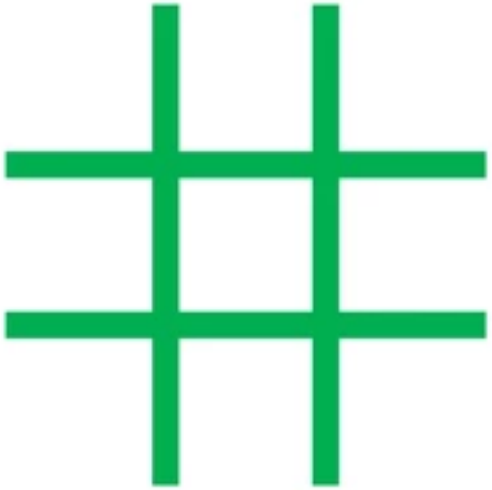
## State-Space Complexity Calculation

# Go

## State-Space Complexity Calculation

# STATE SPACE COMPLEXITY

## TIC-TAC-TOE



4

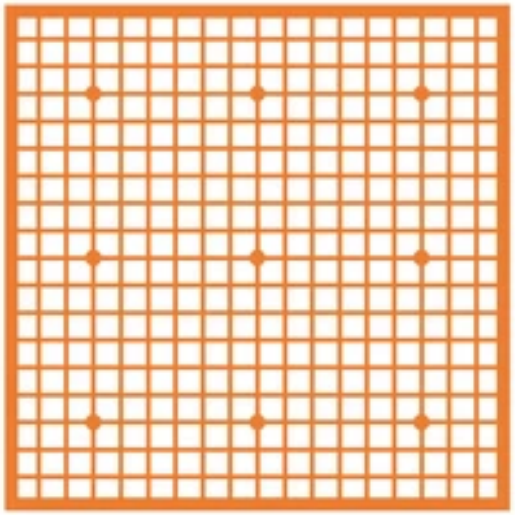
(TEN THOUSAND)

## CHESS

43

(TEN TREDECILLION)

## GO

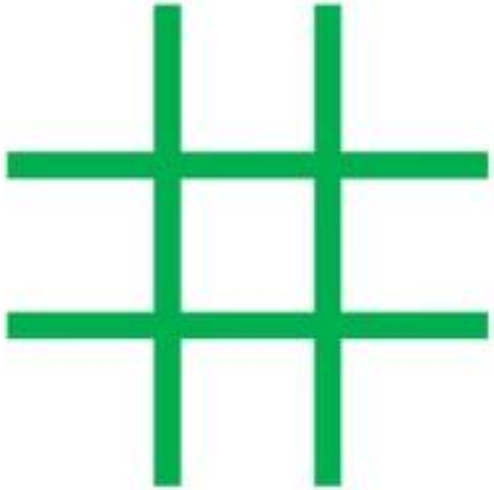


172

(TEN SEXQUINQUAGINTILLION)

# GAME-TREE COMPLEXITY

## TIC-TAC-TOE



5

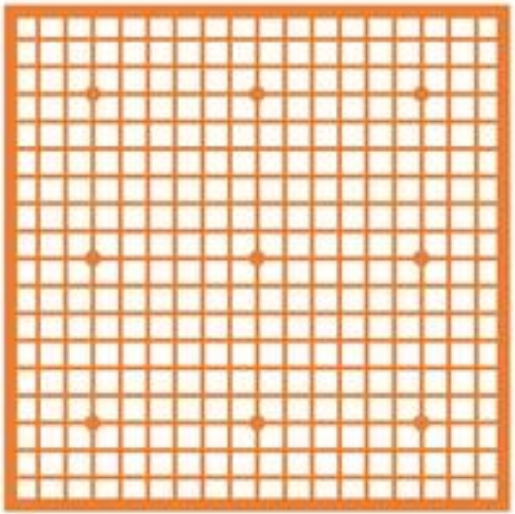
(ONE HUNDRED THOUSAND)

## CHESS

123

(ONE QUADRAGINTILLION)

## GO



360

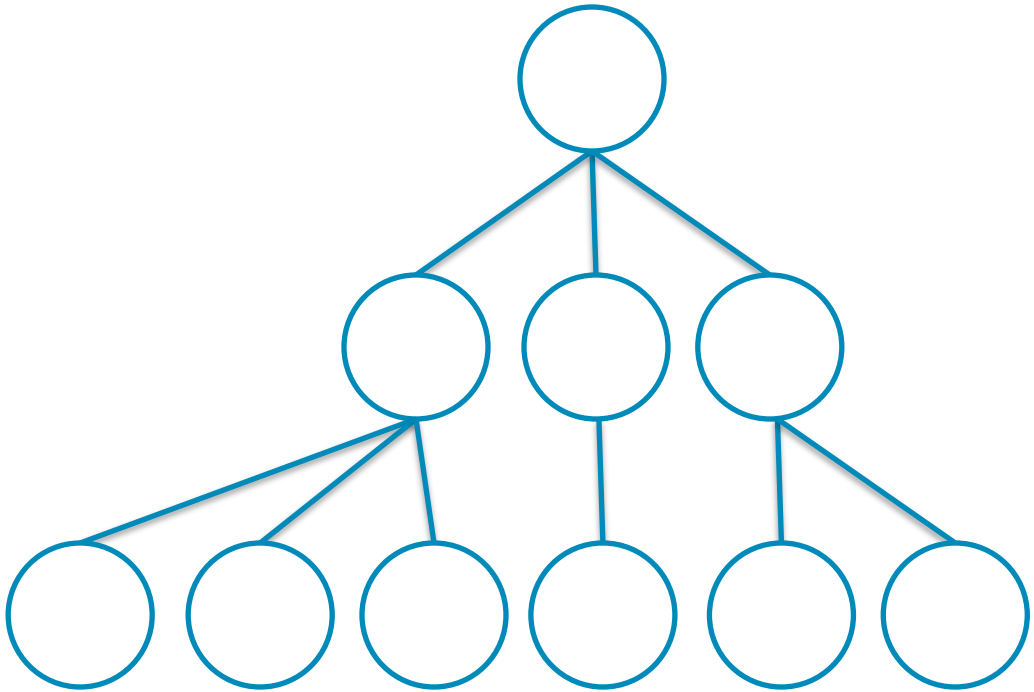
(ONE CENNOVEMDECILLION)

# Game-Playing Computer

- How can we design an algorithm to play a two-player alternating move game?

1. Analysis / Strategy / Tactics
2. Thousands of If-Then Statements
3. Try Actions (Look-Ahead) and Evaluate
4. Search the Entire Game Tree
5. Look-Ahead as Far as Possible

# Look Ahead as Far as Possible

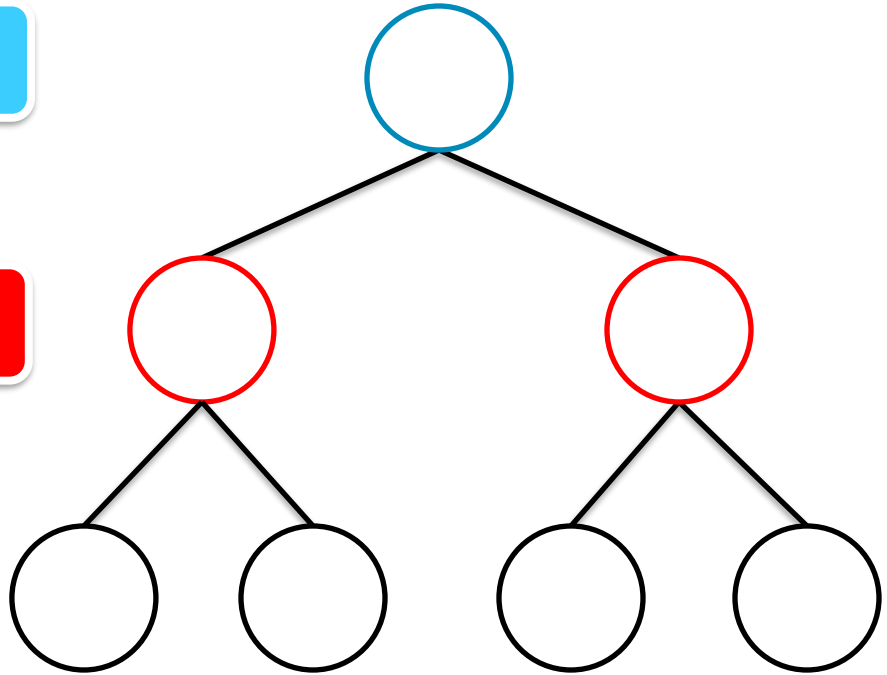


# Competing Players

Player One

Player Two

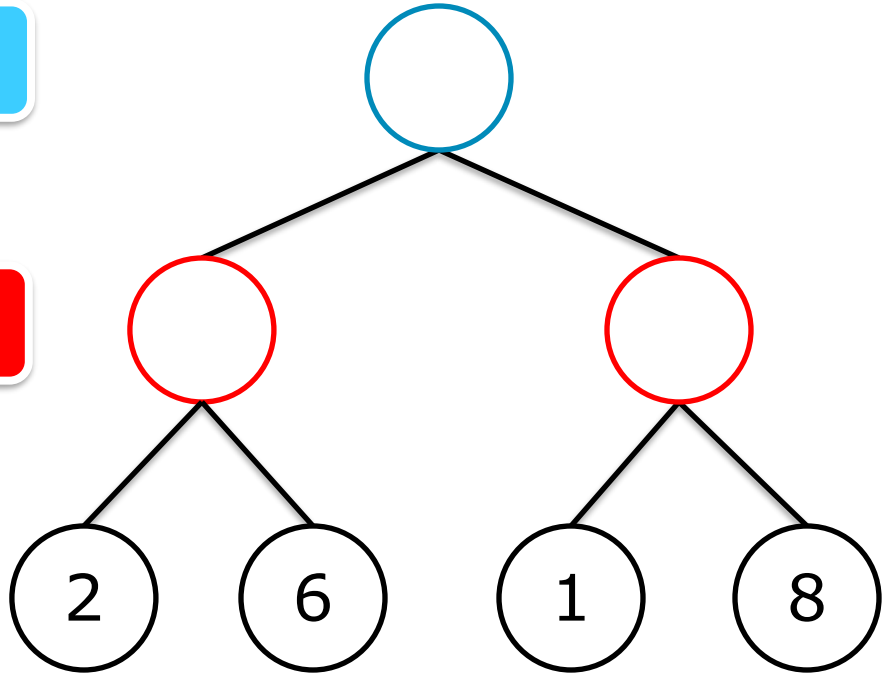
Eval



# Competing Players

Player One

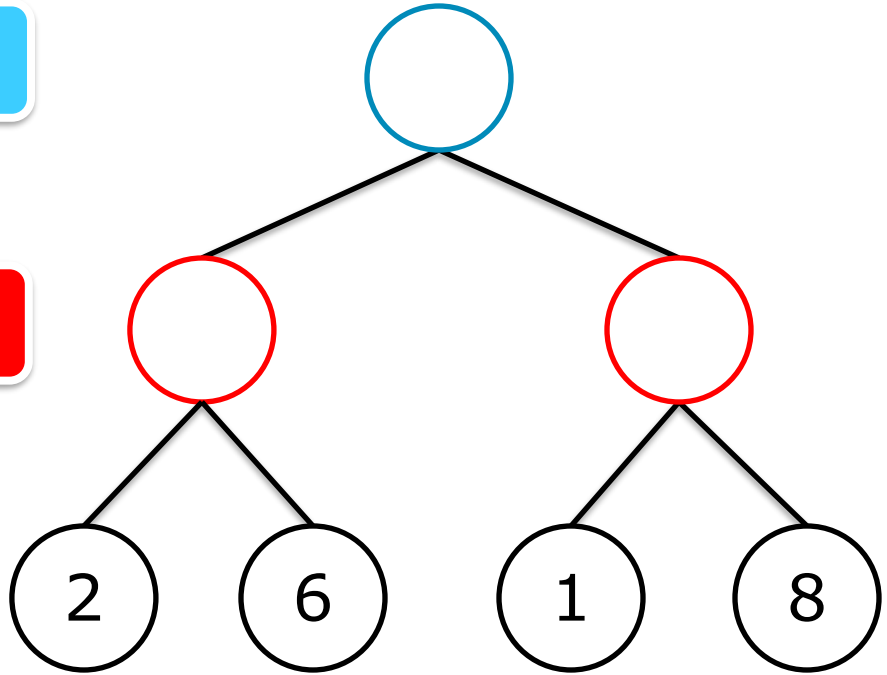
Player Two



# Competing Players

Max Player

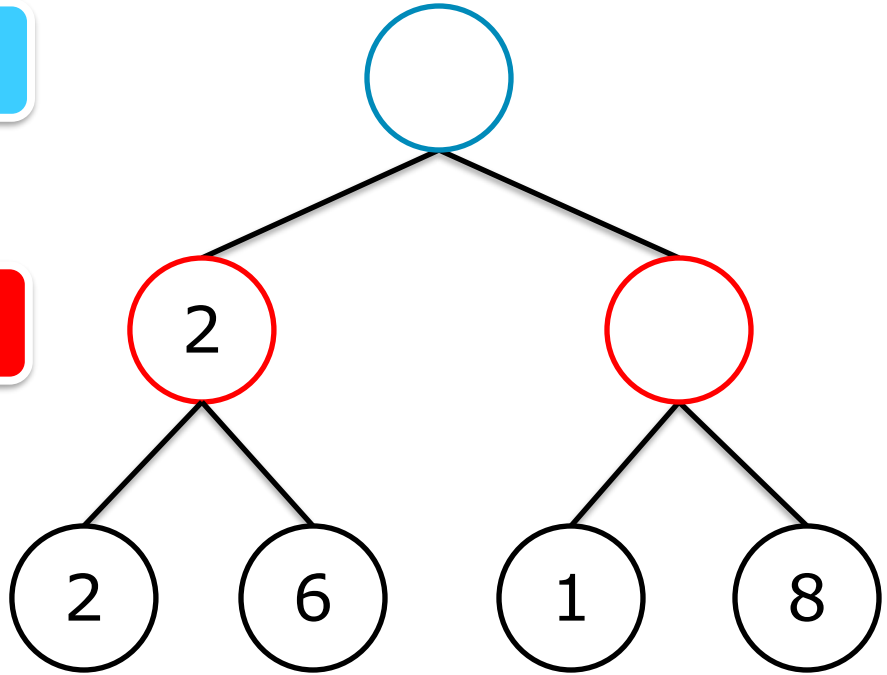
Min Player



# Competing Players

Max Player

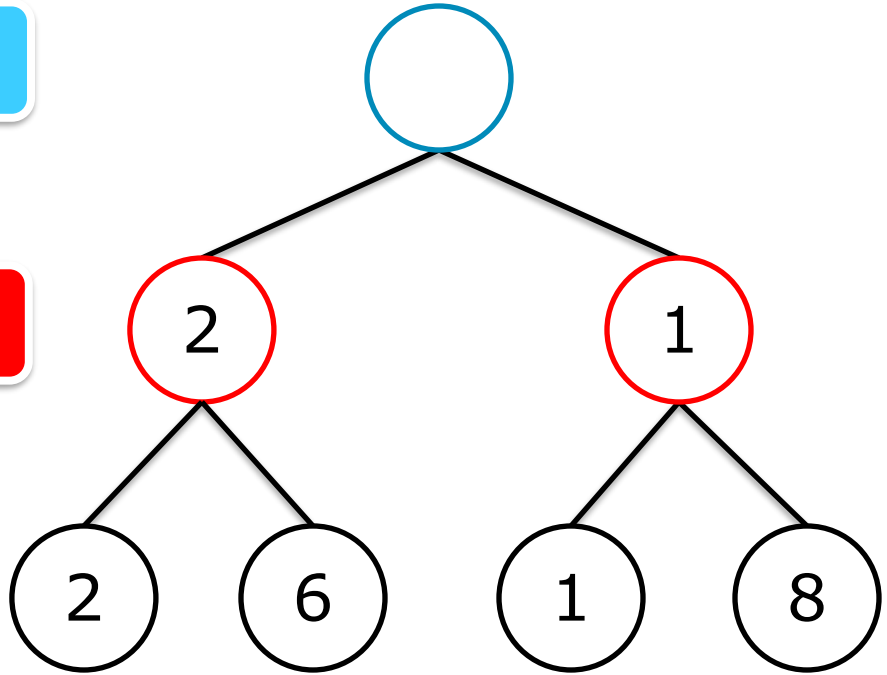
Min Player



# Competing Players

Max Player

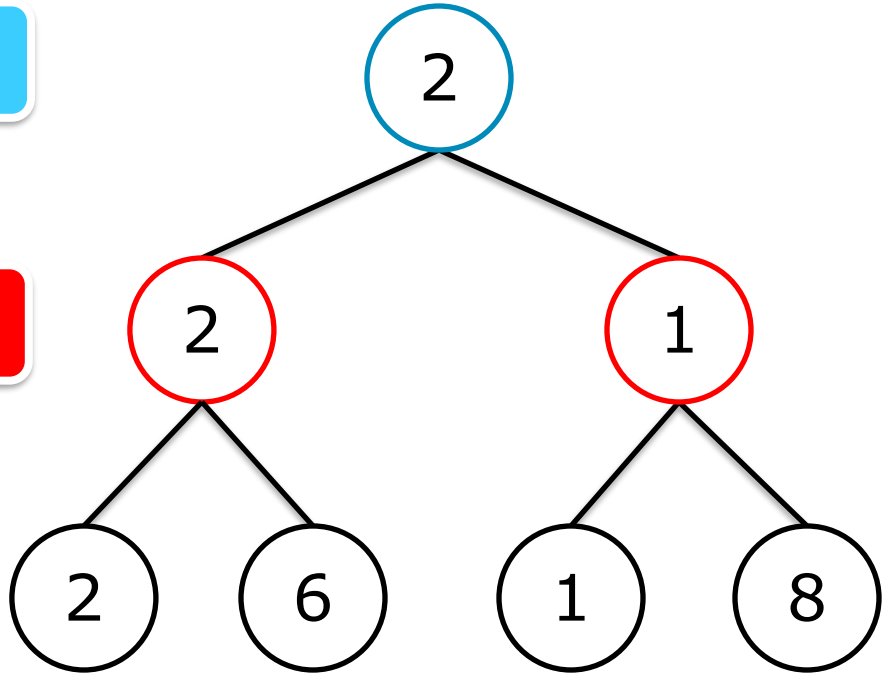
Min Player



# Competing Players

Max Player

Min Player



# Lookahead and Evaluate

```
1.  Function LookaheadAndEvaluate(state)
2.      maxVal = -infinity
3.      bestAction = null
4.      for (action : state.getLegalActions())
5.          child = state.doAction(action)
6.          val = eval(child)
7.          if (val > maxVal)
8.              maxVal = val
9.              bestAction = action
10.     return bestAction
```

# MaxValue (single depth)

```
1. Function MaxValue(s)
2.     v = -infinity
3.     for (c in children(s))
4.         v' = eval(c)
5.         if (v' > v) v = v'
6.     return v
```

# MaxValue (full tree)

```
1.  Function MaxValue(s)
2.      if terminal(s)
3.          return eval(s)
4.      v = -infinity
5.      for (c in children(s))
6.          v'= MinValue(c)
7.          if (v' > v) v = v'
8.      return v
```

# MaxValue and MinValue (full tree)

```
1.  Function MaxValue(s)
2.      if (terminal(s))
3.          return eval(s)
4.      v = -infinity
5.      for (c in children(s))
6.          v'= MinValue(c)
7.          if (v' > v) v = v'
8.      return v
```

```
1.  Function MinValue(s)
2.      if (terminal(s))
3.          return eval(s)
4.      v = +infinity
5.      for (c in children(s))
6.          v'= MaxValue(c)
7.          if (v' < v) v = v'
8.      return v
```

eval(s) returns score w.r.t. the maximizing player

# MaxValue and MinValue (depth limit)

```
1.  Function MaxValue(s, d)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      v = -infinity
5.      for (c in children(s))
6.          v'= MinValue(c, d+1)
7.          if (v' > v) v = v'
8.      return v
```

```
1.  Function MinValue(s, d)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      v = +infinity
5.      for (c in children(s))
6.          v'= MaxValue(c, d+1)
7.          if (v' < v) v = v'
8.      return v
```

eval(s) returns score w.r.t. the maximizing player

# Minimax Algorithm

```
1.  Function MiniMax(s, d, max)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      if (max)        // maximizing player
5.          v = -infinity
6.          for (c in children(s))
7.              v = max(v, MiniMax(c, d+1, false))
8.          return v
9.      else            // minimizing player
10.         v = +infinity
11.         for (c in children(s))
12.             v = min(v, MiniMax(c, d+1, true))
13.         return v
Initial Call: MiniMax(startState, 0, true)
```

# Negamax Algorithm

```
1.  Function NegaMax(s, d, player)
2.      if (terminal(s) d > maxD)
3.          return eval(s, player)
4.      v = -infinity
5.      for (c in children(s))
6.          v = max(v, -NegaMax(child, d+1, !player))
7.      return v
```

Relies on: max(a, b) = -min(-a, -b)

Initial Call: NegaMax(startState, 0, startState.player)

# Mini-Max Properties

- Complete and Optimal: Will find the optimal solution to a given max depth
- Each player plays a best response to the possible actions of the other player
- Mini-Max plays the Nash Equilibrium

# Nash Equilibrium (NE)

- For two-player, finite, zero-sum games, a Nash Equilibrium exists
- Recall that in a NE:
  - Each player is best responding to the other
  - Neither player can gain by deviating
  - Neither player has any regrets
- Playing a NE is a very strong strategy

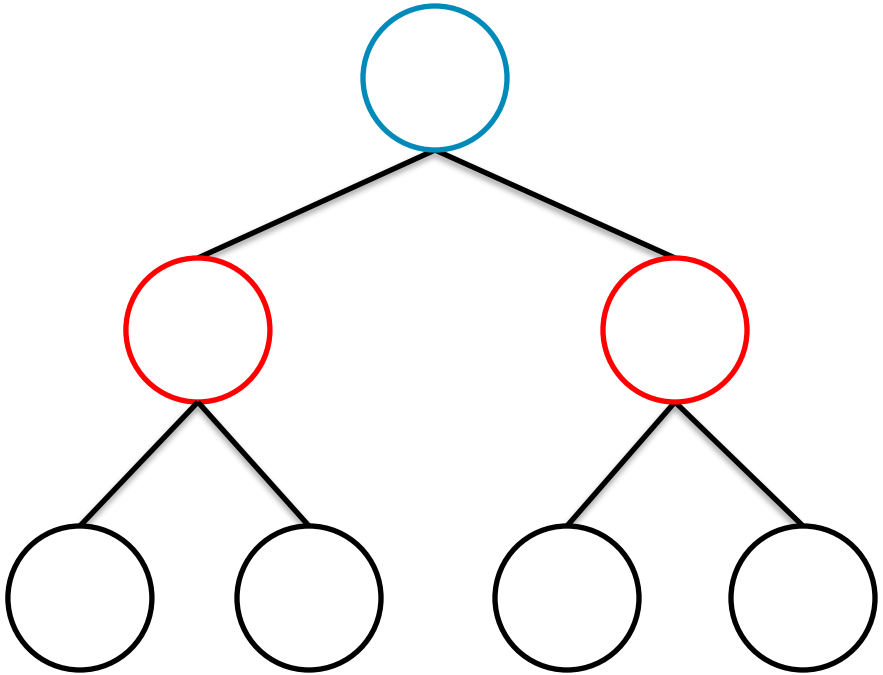
# Improving Minimax

# Alpha-Beta Pruning

# Improving MiniMax

Max

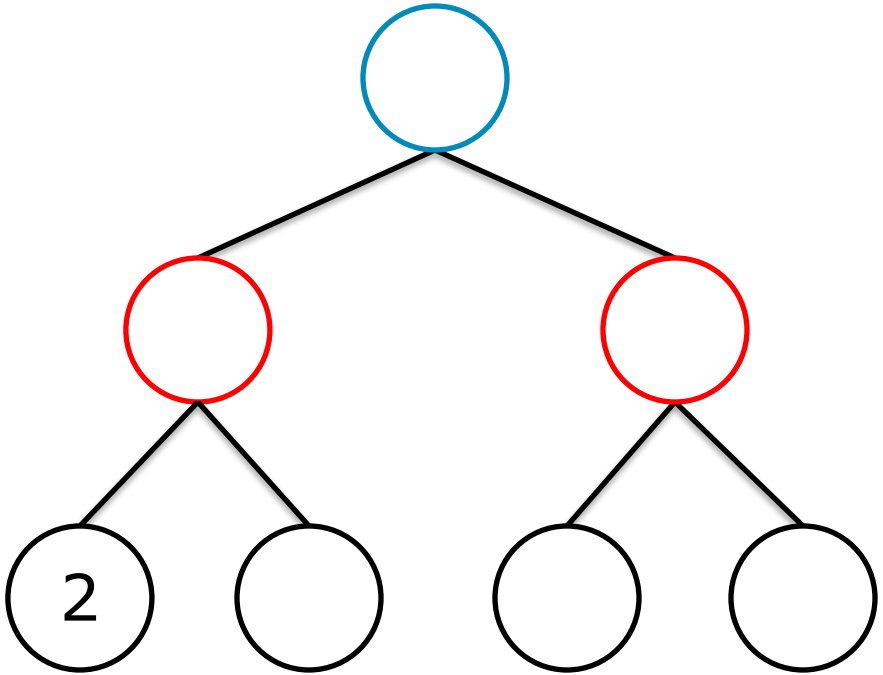
Min



# Improving MiniMax

Max

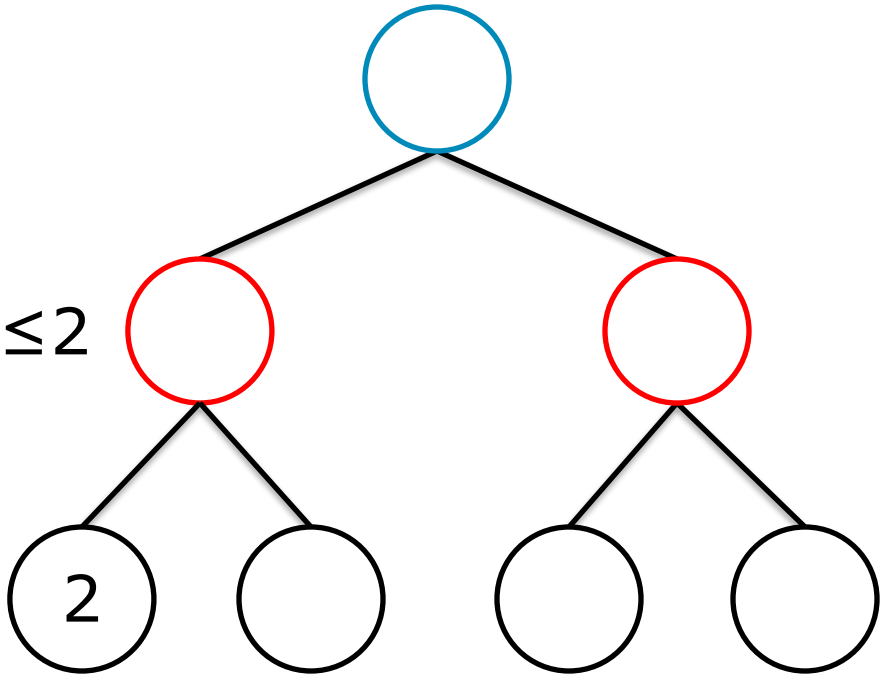
Min



# Improving MiniMax

Max

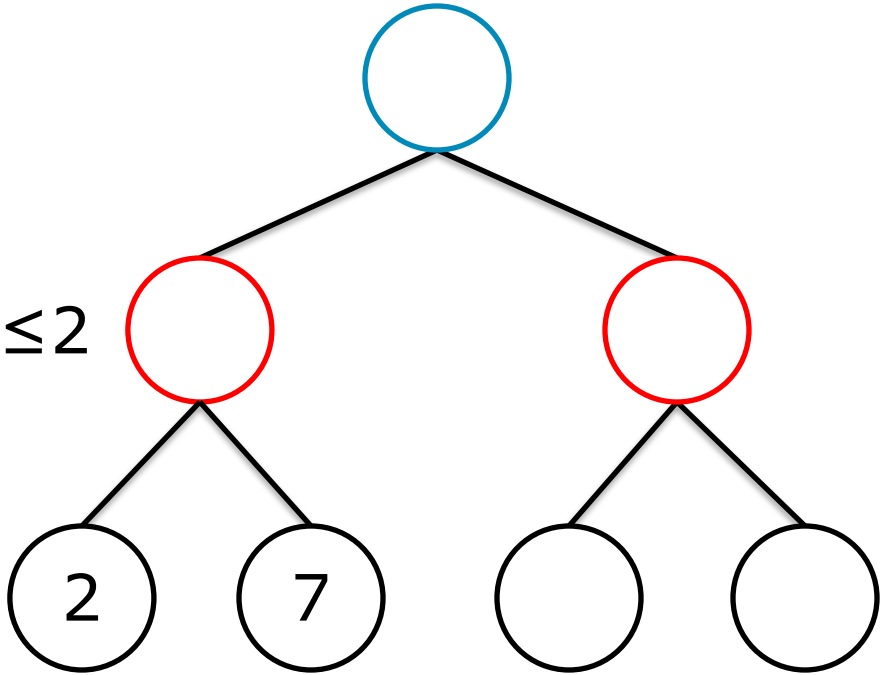
Min



# Improving MiniMax

Max

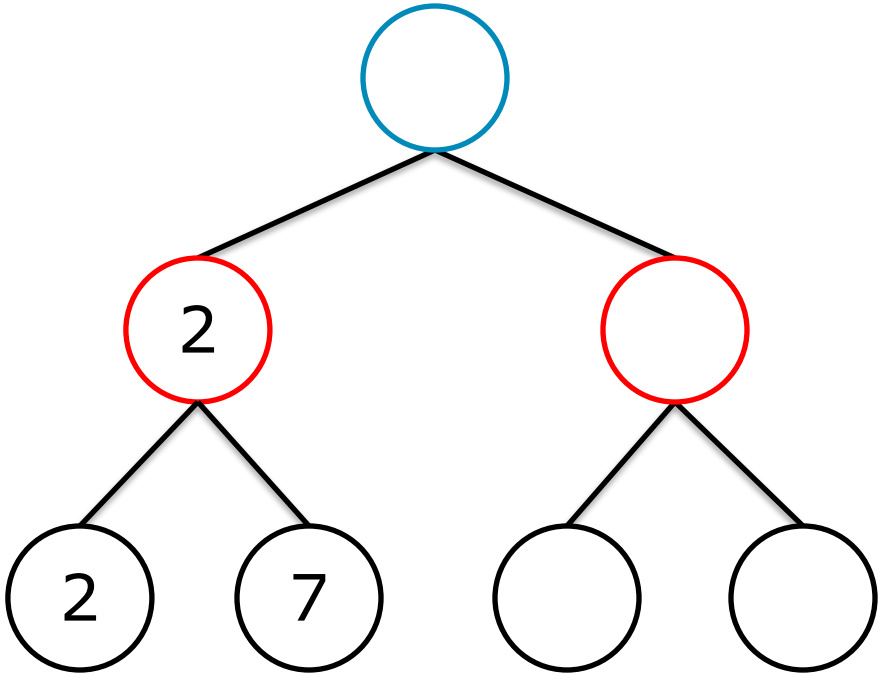
Min



# Improving MiniMax

Max

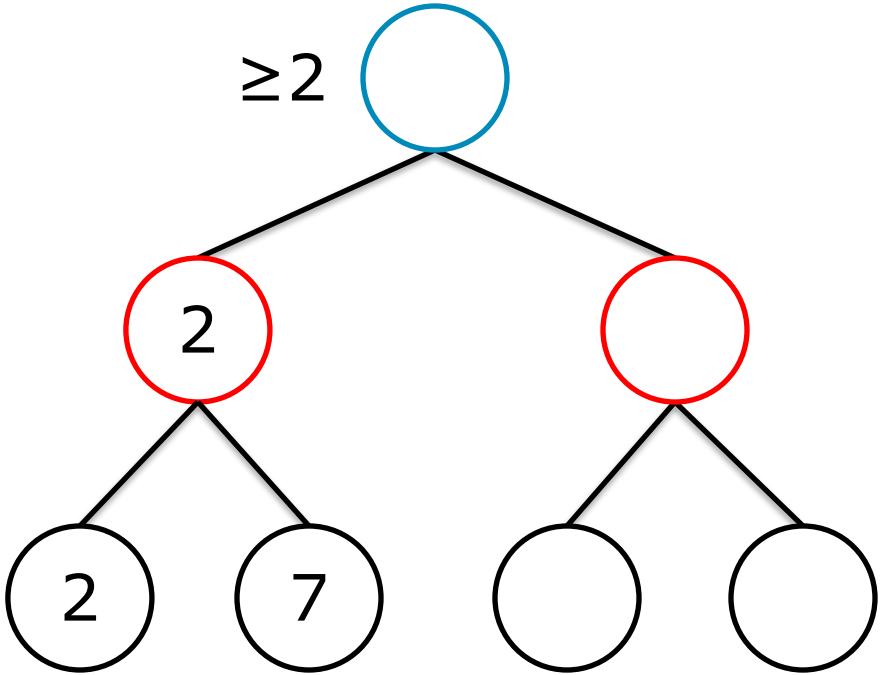
Min



# Improving MiniMax

Max

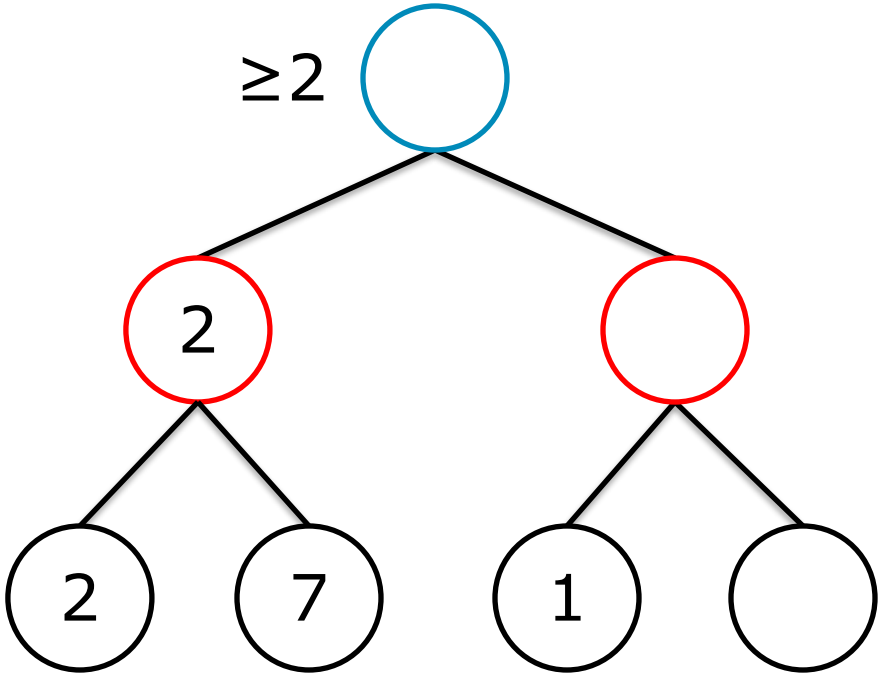
Min



# Improving MiniMax

Max

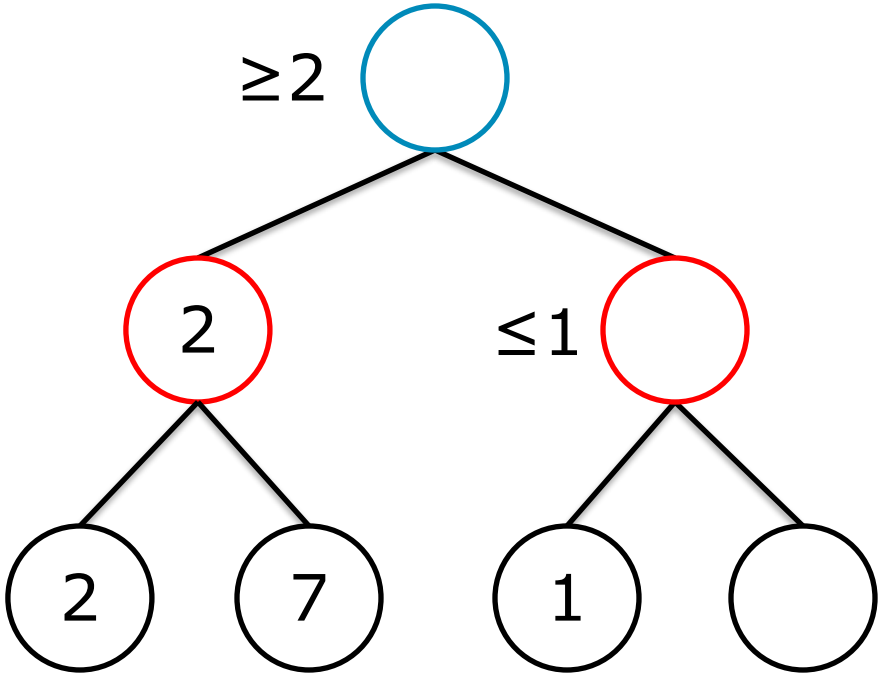
Min



# Improving MiniMax

Max

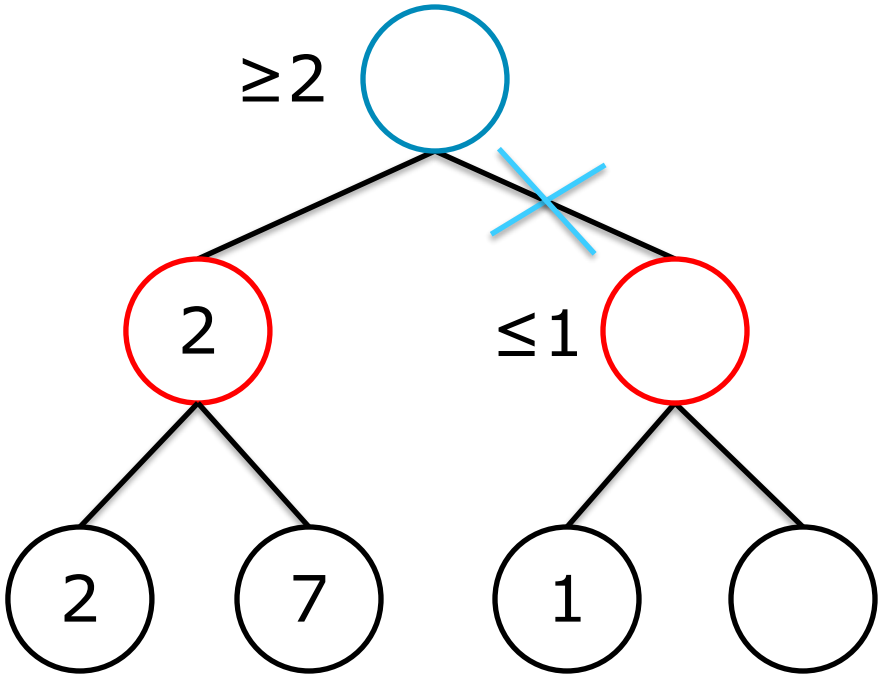
Min



# Improving MiniMax

Max

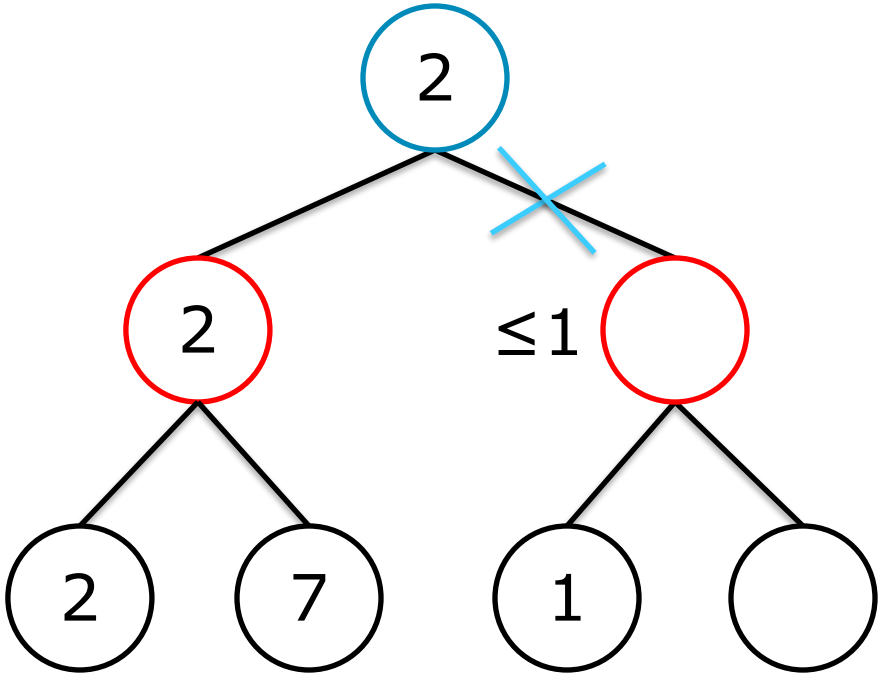
Min



# Improving MiniMax

Max

Min

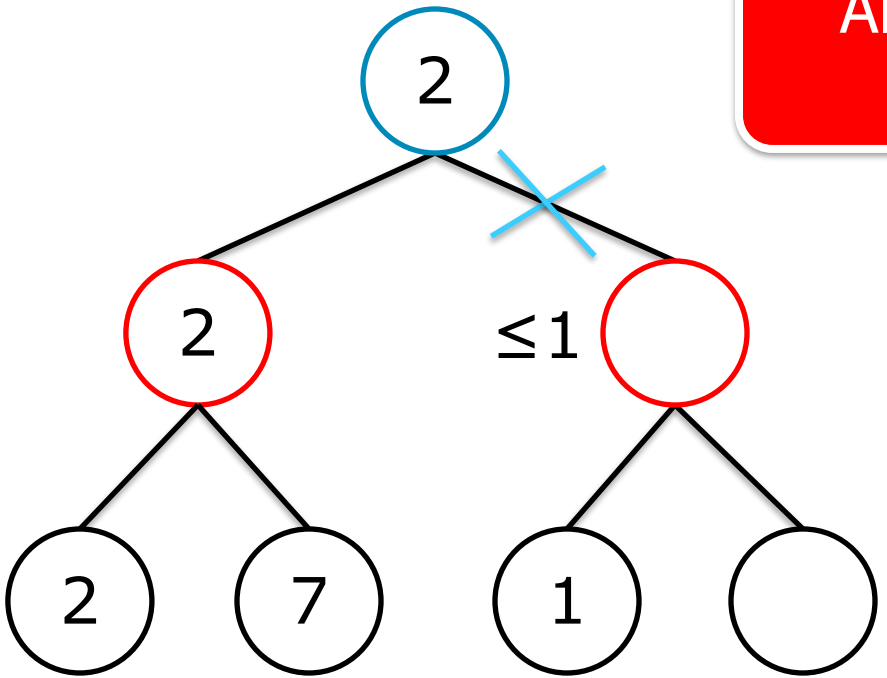


# Improving MiniMax

Max

Min

Alpha-Beta Pruning



# Alpha-Beta Pruning

- Alpha-Beta is not a different algorithm than MiniMax, it is an optimization
  - “Minimax with Alpha-Beta Pruning”
- Maintains all of the properties of minimax but is strictly better
- Cutting off branches of the search tree yields exponential savings

Before pruning

A well-shaped plant
after pruning

# Alpha-Beta Pruning

- Uses the MiniMax algorithm
- Introduce 2 new variables
- Alpha (α) – Best alternative for Max on this particular path through the tree
- Beta (β) – Best alternative for Min on this particular path through the tree
- α, β are an ever-narrowing window which 'good' paths pass through, and other paths are pruned

# MaxValue and MinValue (minimax)

```
1.  Function MaxValue(s, d)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      v = -infinity
5.      for (c in children(s))
6.          v'= MinValue(c, d+1)
7.          if (v' > v) v = v'
8.      return v
```

```
1.  Function MinValue(s, d)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      v = +infinity
5.      for (c in children(s))
6.          v'= MaxValue(c, d+1)
7.          if (v' < v) v = v'
8.      return v
```

eval(s) returns score w.r.t. the maximizing player

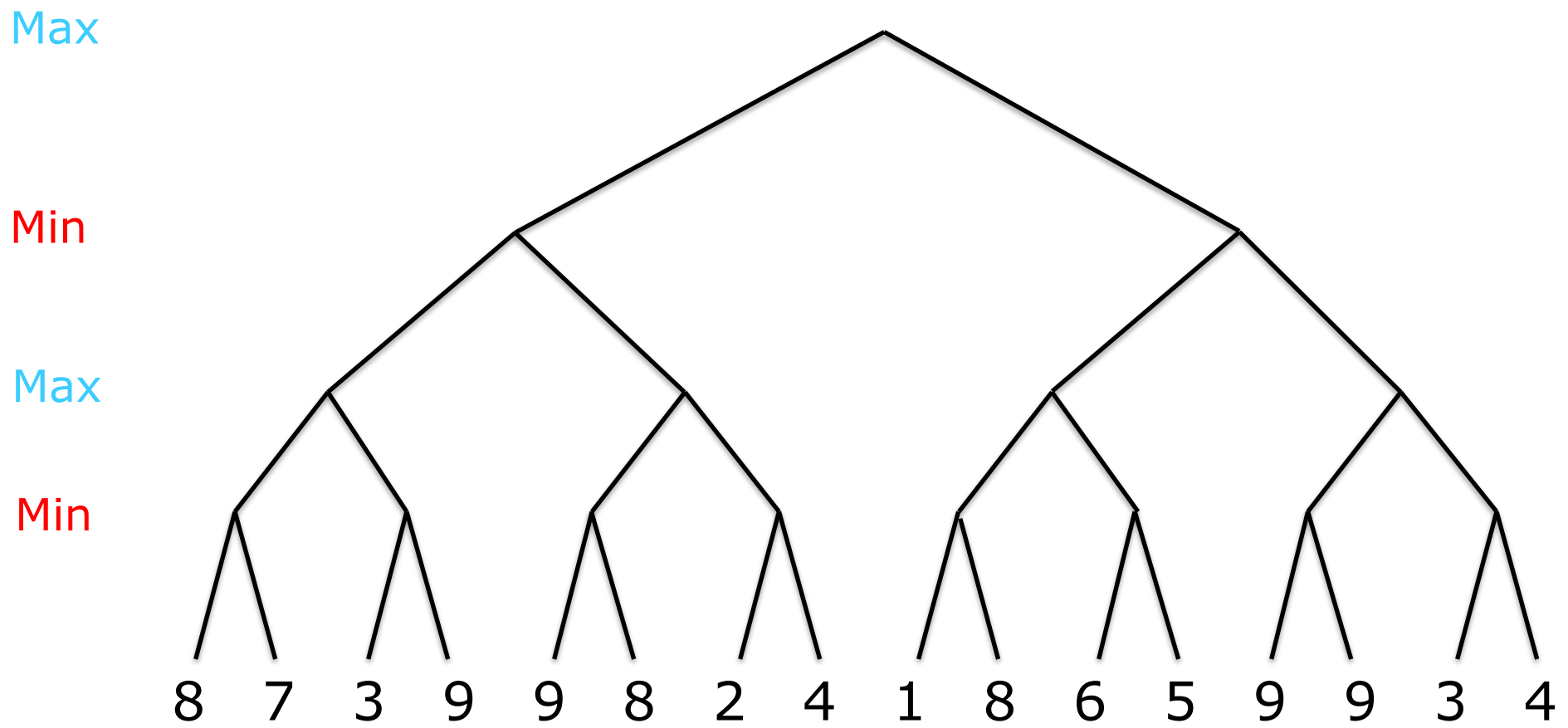
# MaxValue and MinValue (alpha beta)

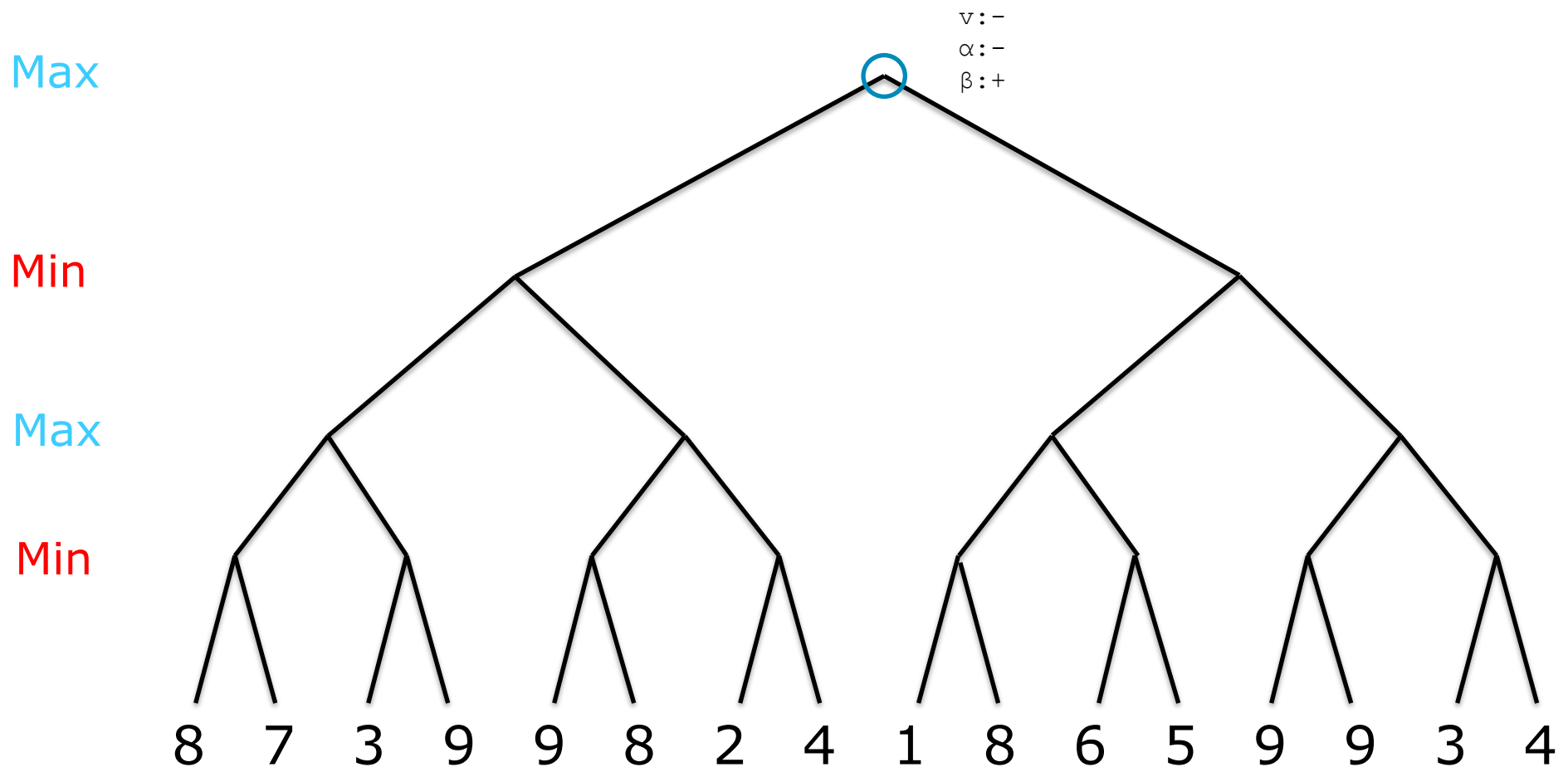
```
1.  Function MaxValue(s, α, β, d)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      v = -infinity
5.      for (c in children(s))
6.          v'= MinValue(c, α, β, d+1)
7.          if (v' > v) v = v'
8.          if (v' ≥ β) return v
9.          if (v' > α) α = v'
10.     return v
```

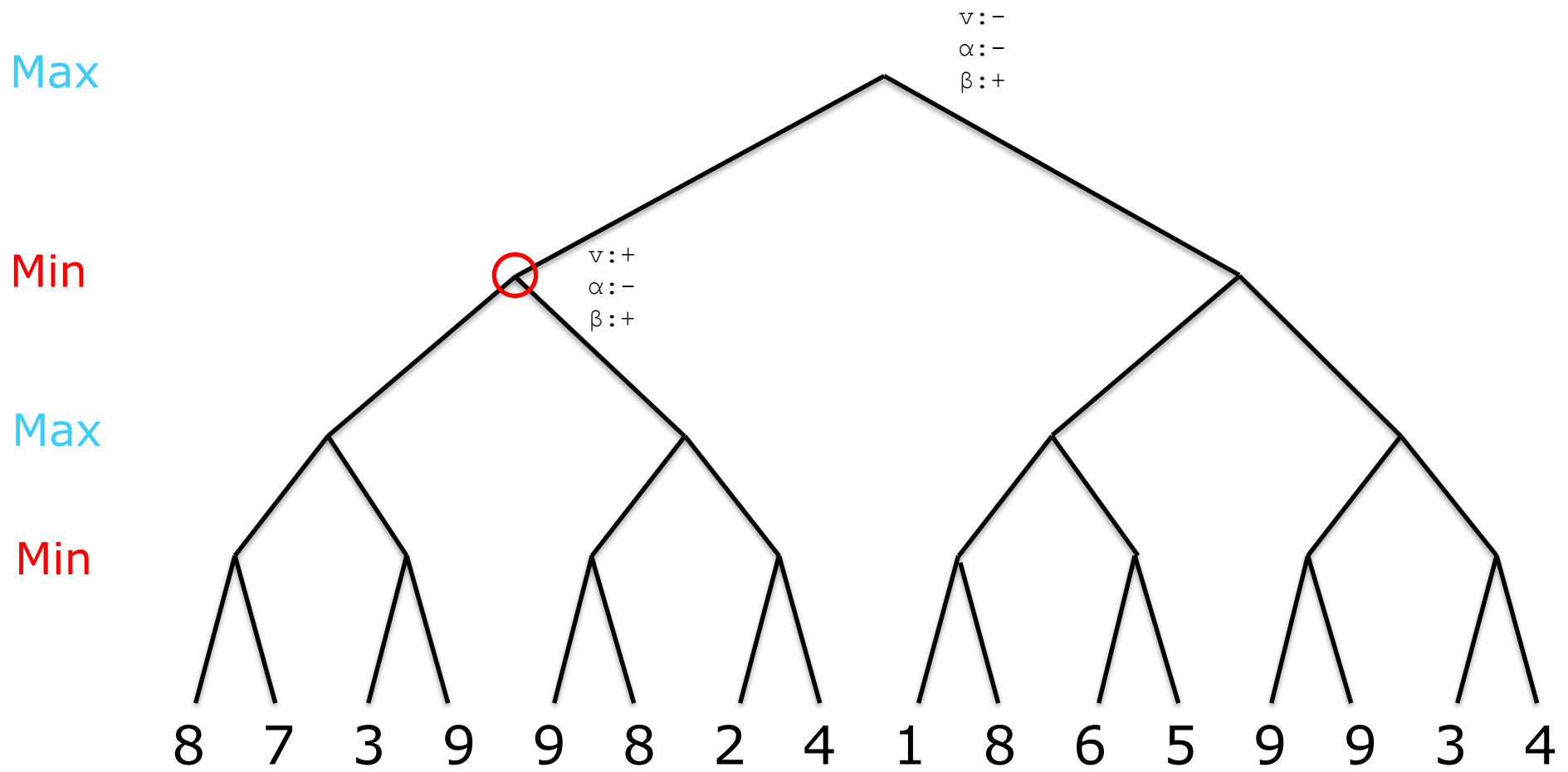
```
1.  Function MinValue(s, α, β, d)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      v = +infinity
5.      for (c in children(s))
6.          v'= MaxValue(c, α, β, d+1)
7.          if (v' < v) v = v'
8.          if (v' ≤ α) return v
9.          if (v' < β) β = v'
10.     return v
```

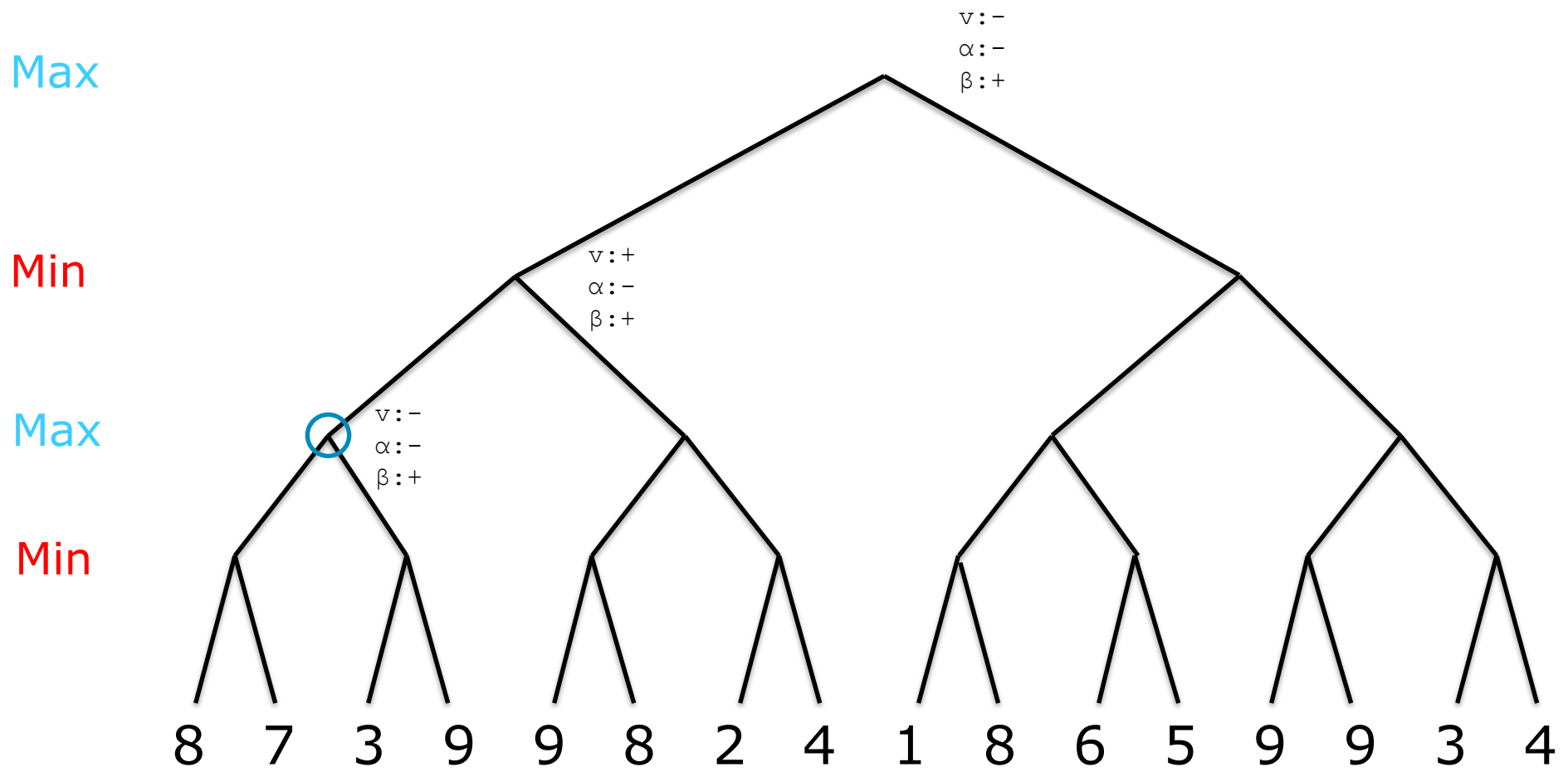
Initial Call: MaxValue(startState, -infinity, +infinity, 0)

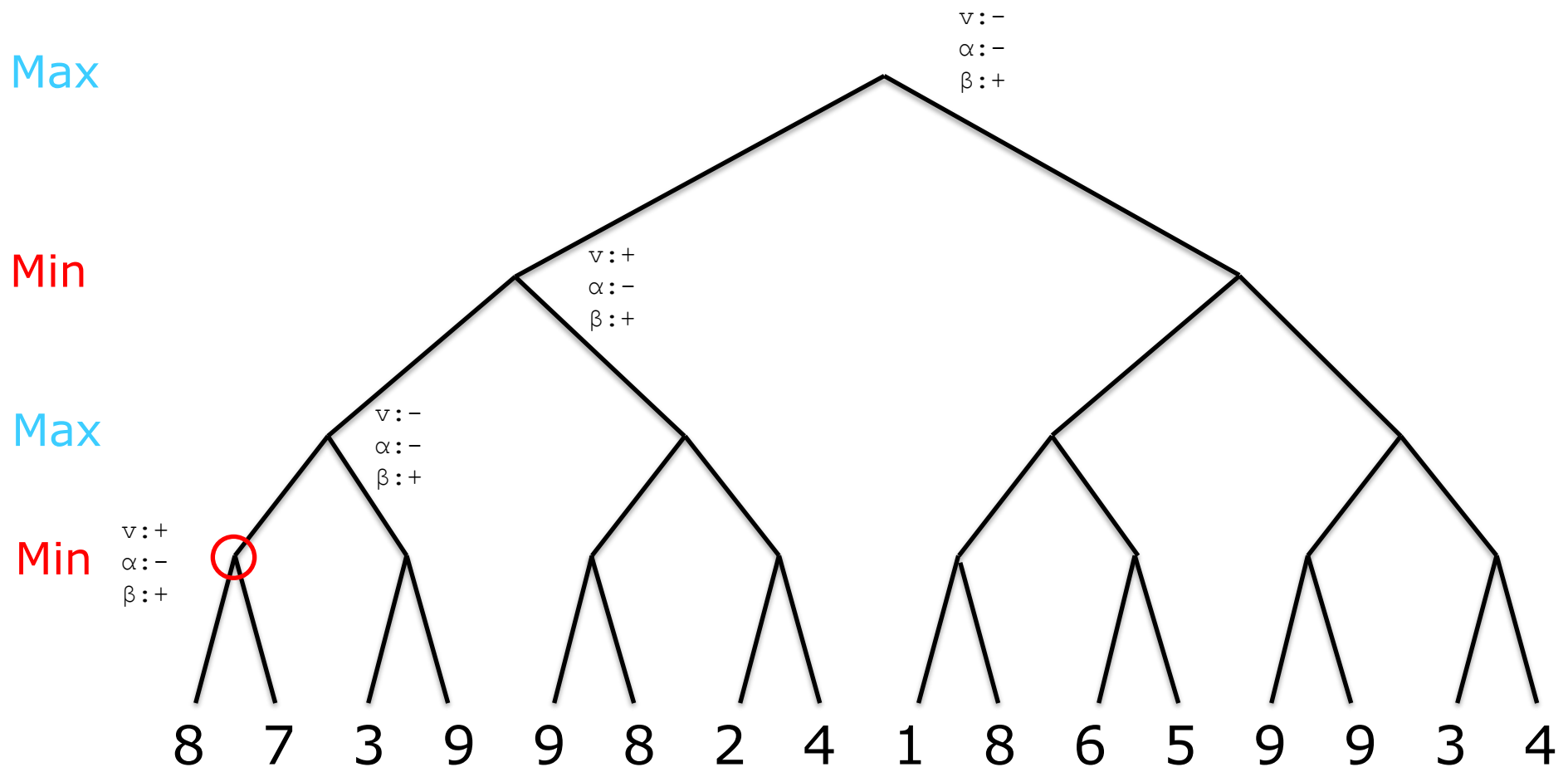
MaxValue(startState, -infinity, +infinity, 0)

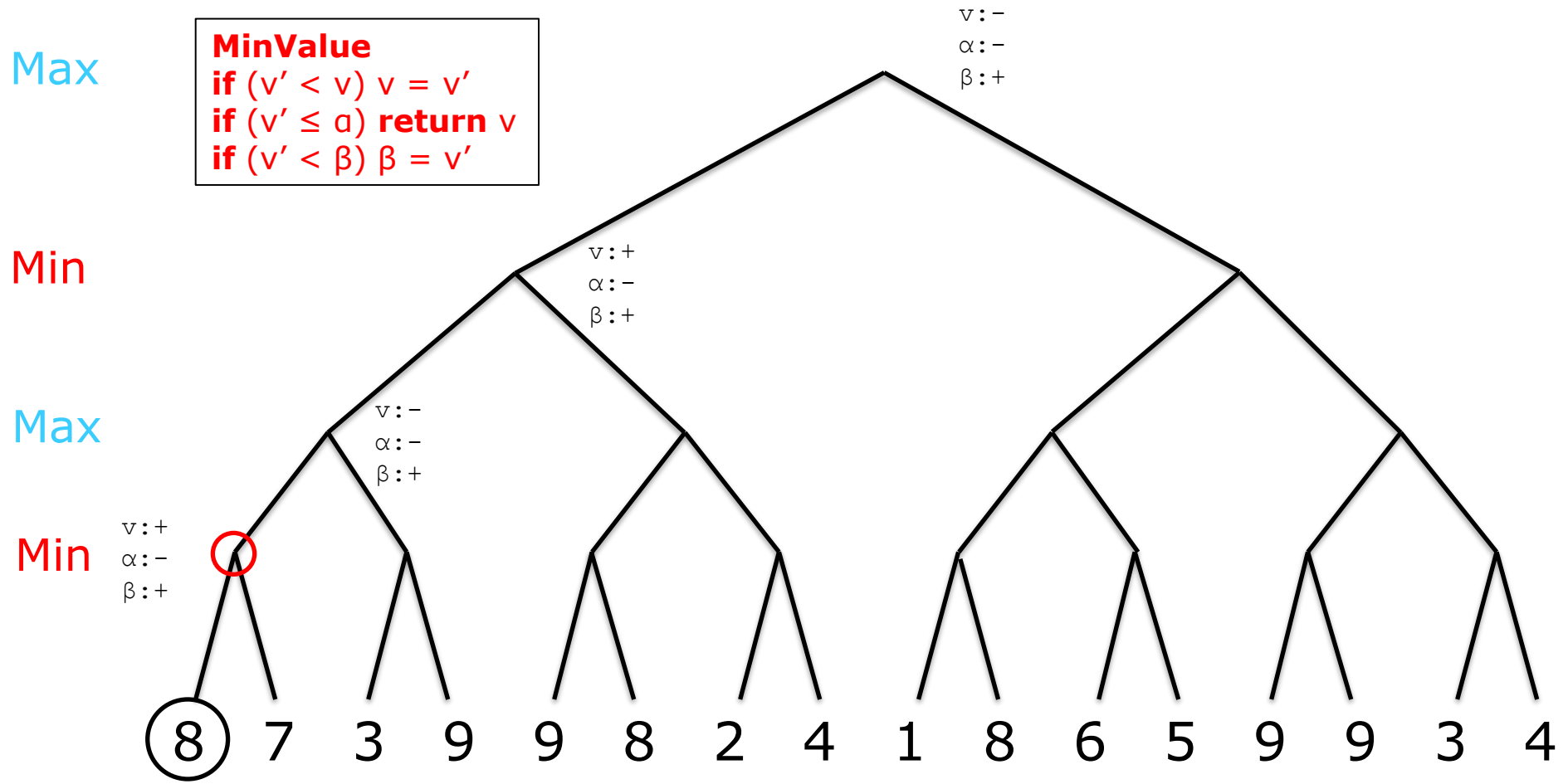


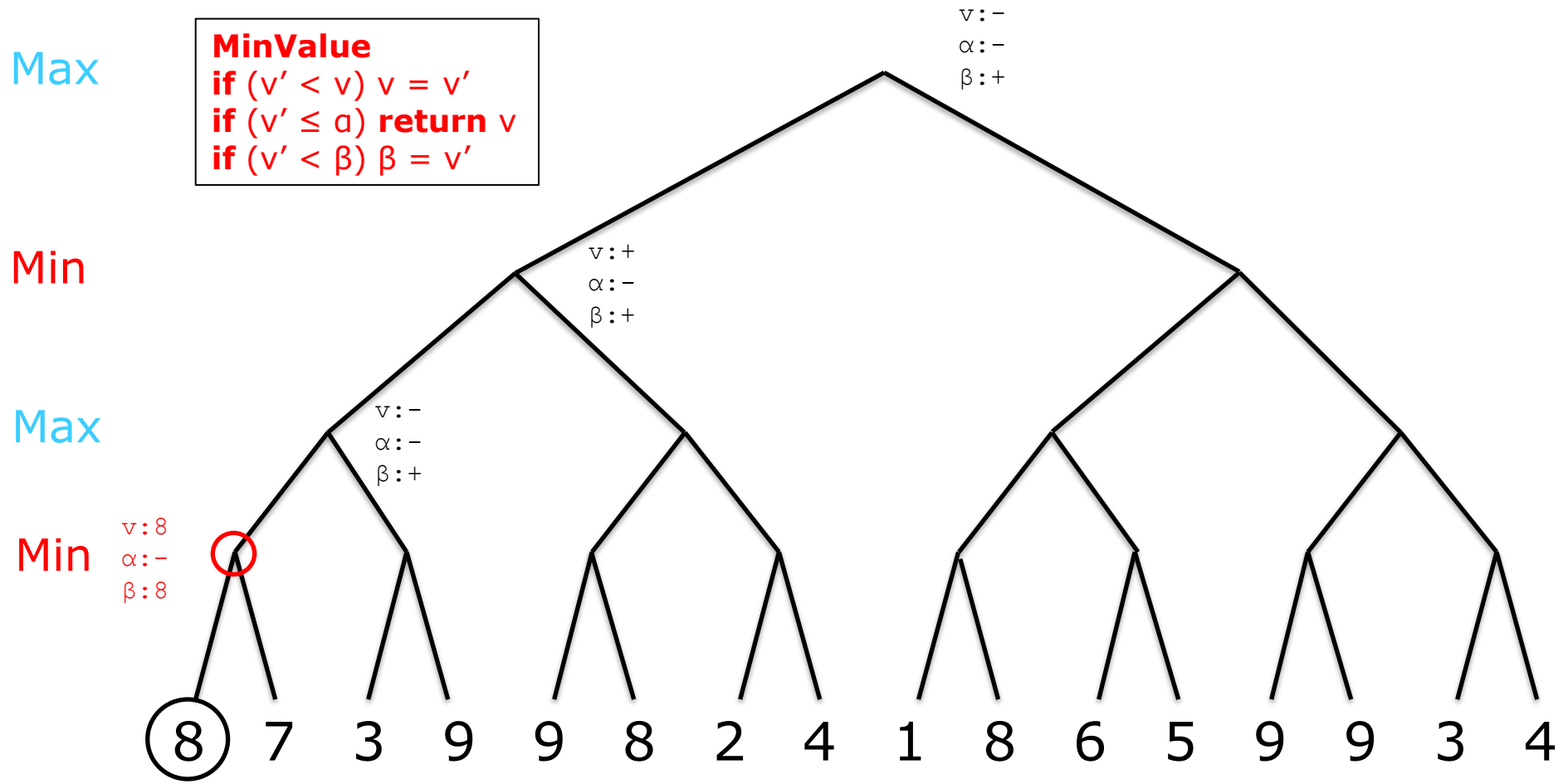

v : −
α : −
β : +
Max
Min
Max
Min
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

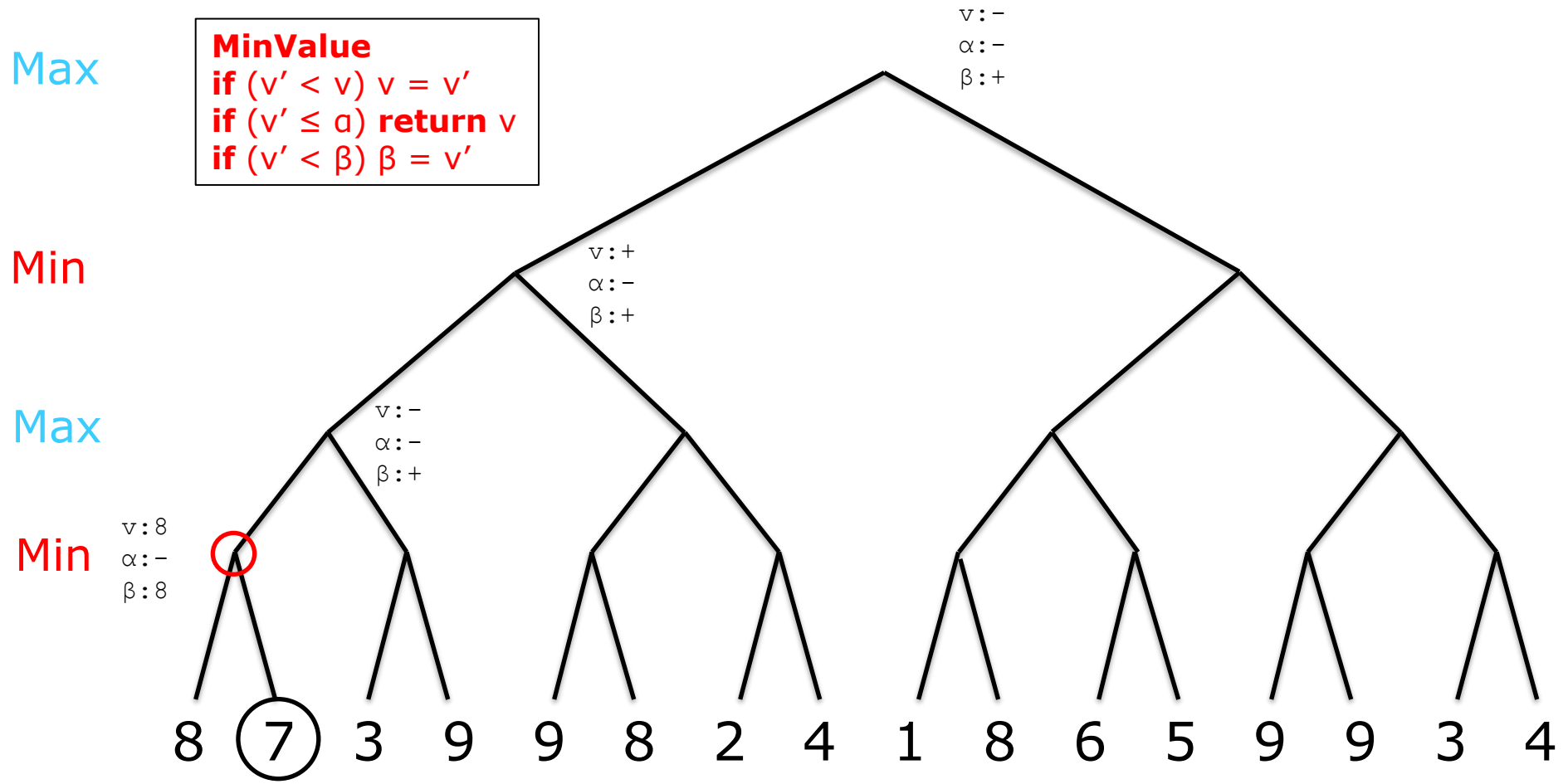

Max
Min
Max
Min
v : −
α : −
β : +
v : +
α : −
β : +
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

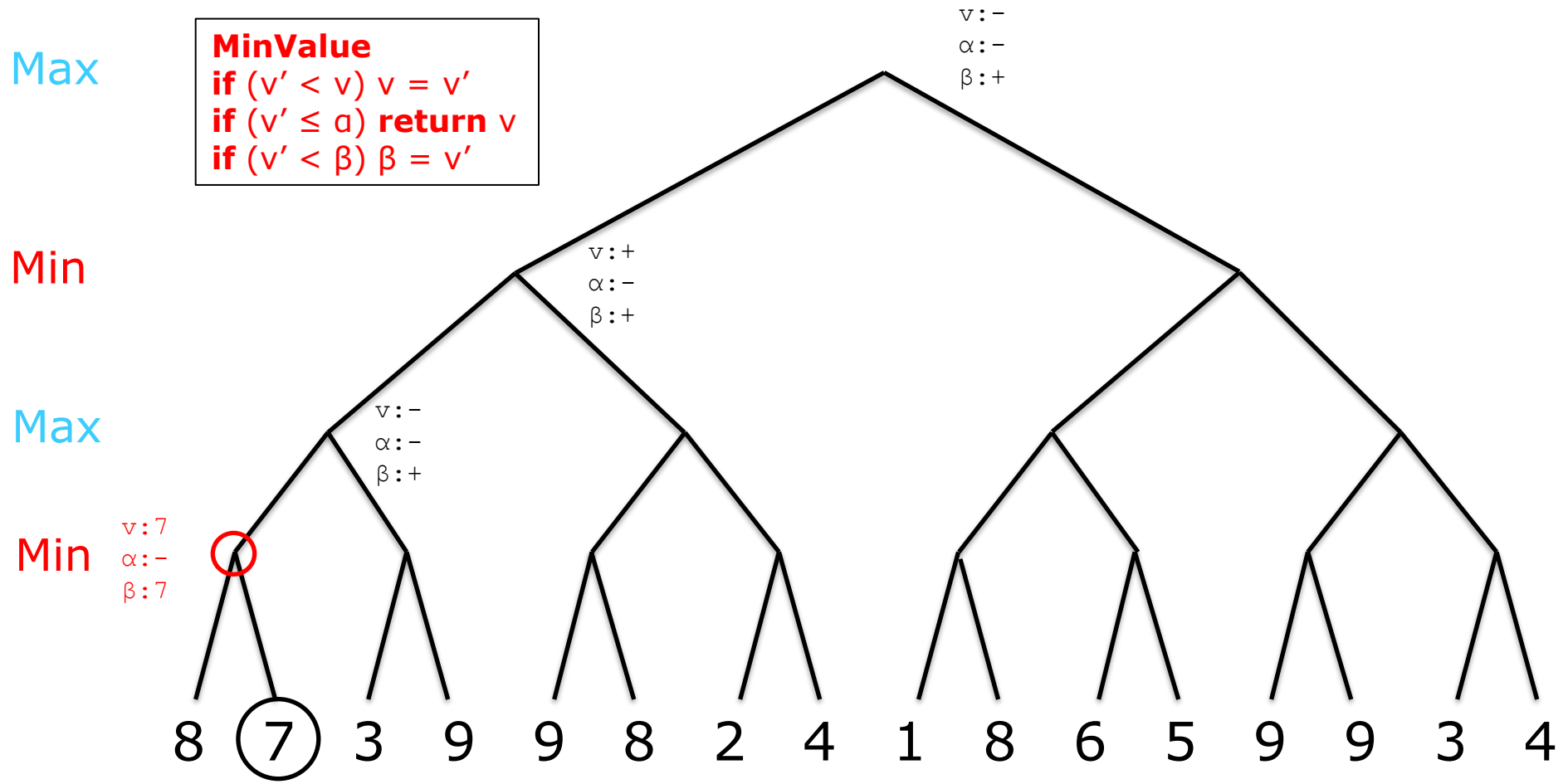

Max
Min
Max
Min
v : −
α : −
β : +
v : +
α : −
β : +
v : −
α : −
β : +
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

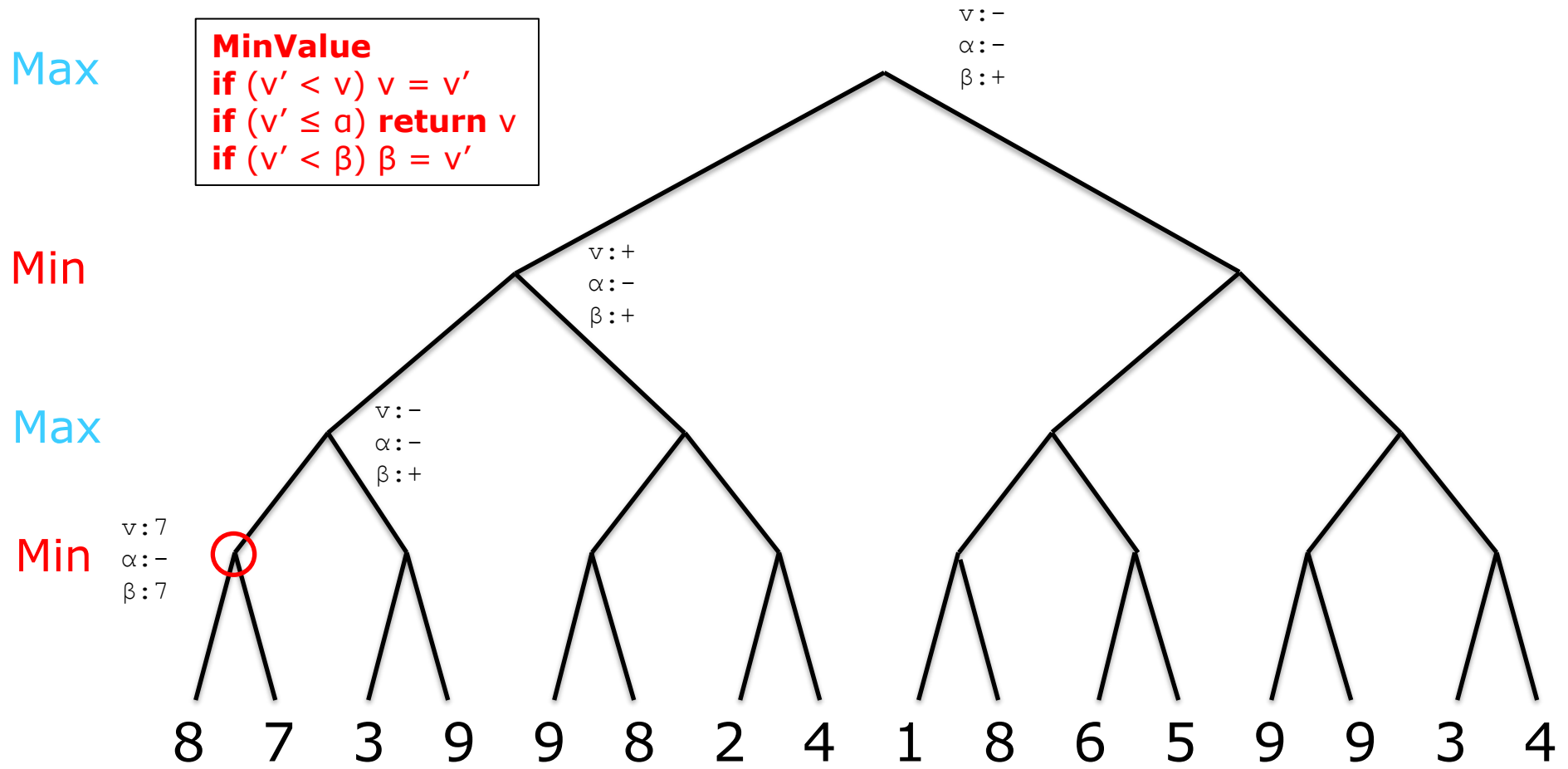

Max
Min
Max
Min
v : −
α : −
β : +
v : +
α : −
β : +
v : −
α : −
β : +
v : +
α : −
β : +
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

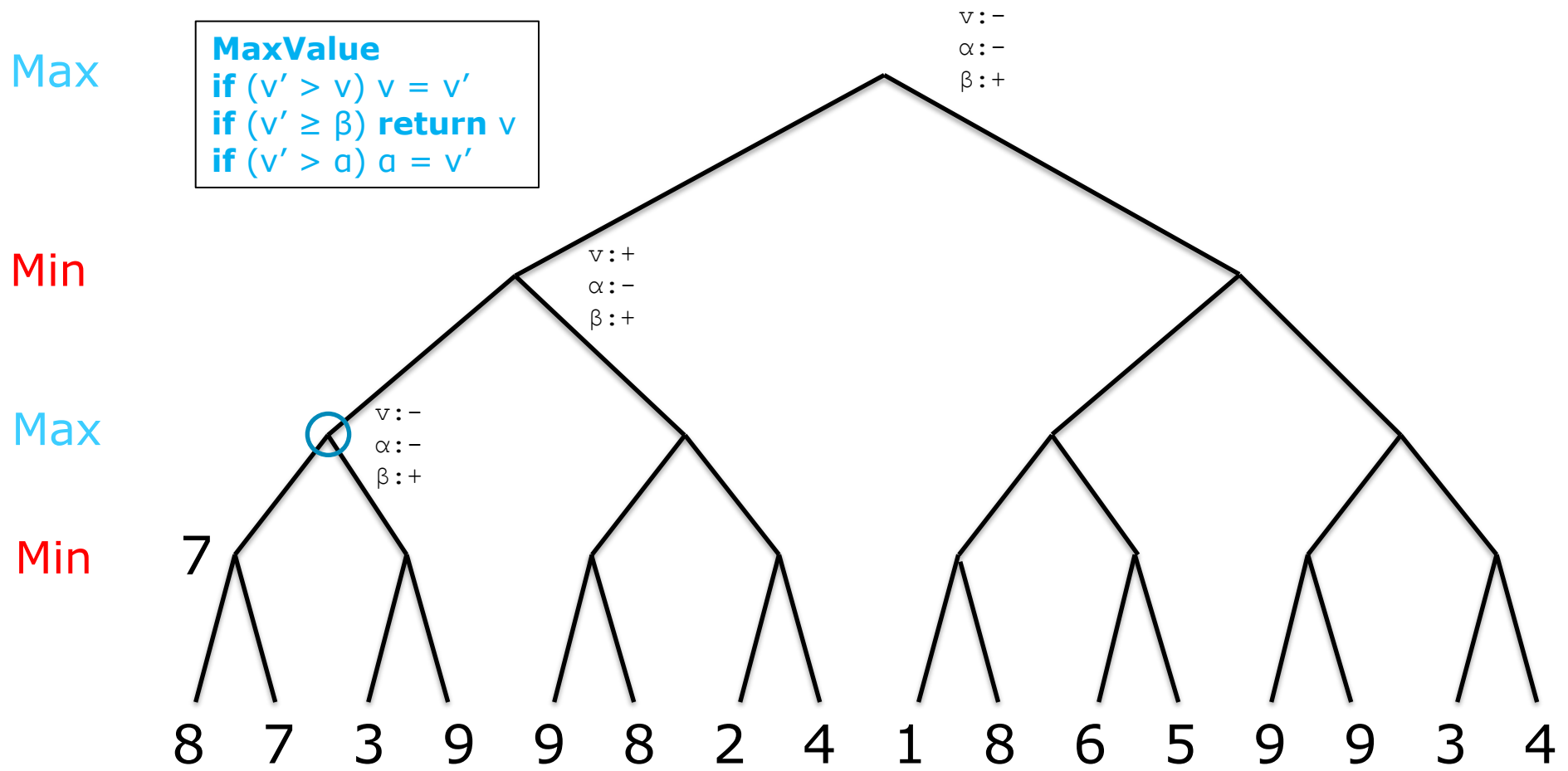

MinValue
if (v′ < v) v = v′
if (v′ ≤ α) return v
if (v′ < β) β = v′
Max
Min
Max
Min
v : −
α : −
β : +
v : +
α : −
β : +
v : −
α : −
β : +
v : +
α : −
β : +
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

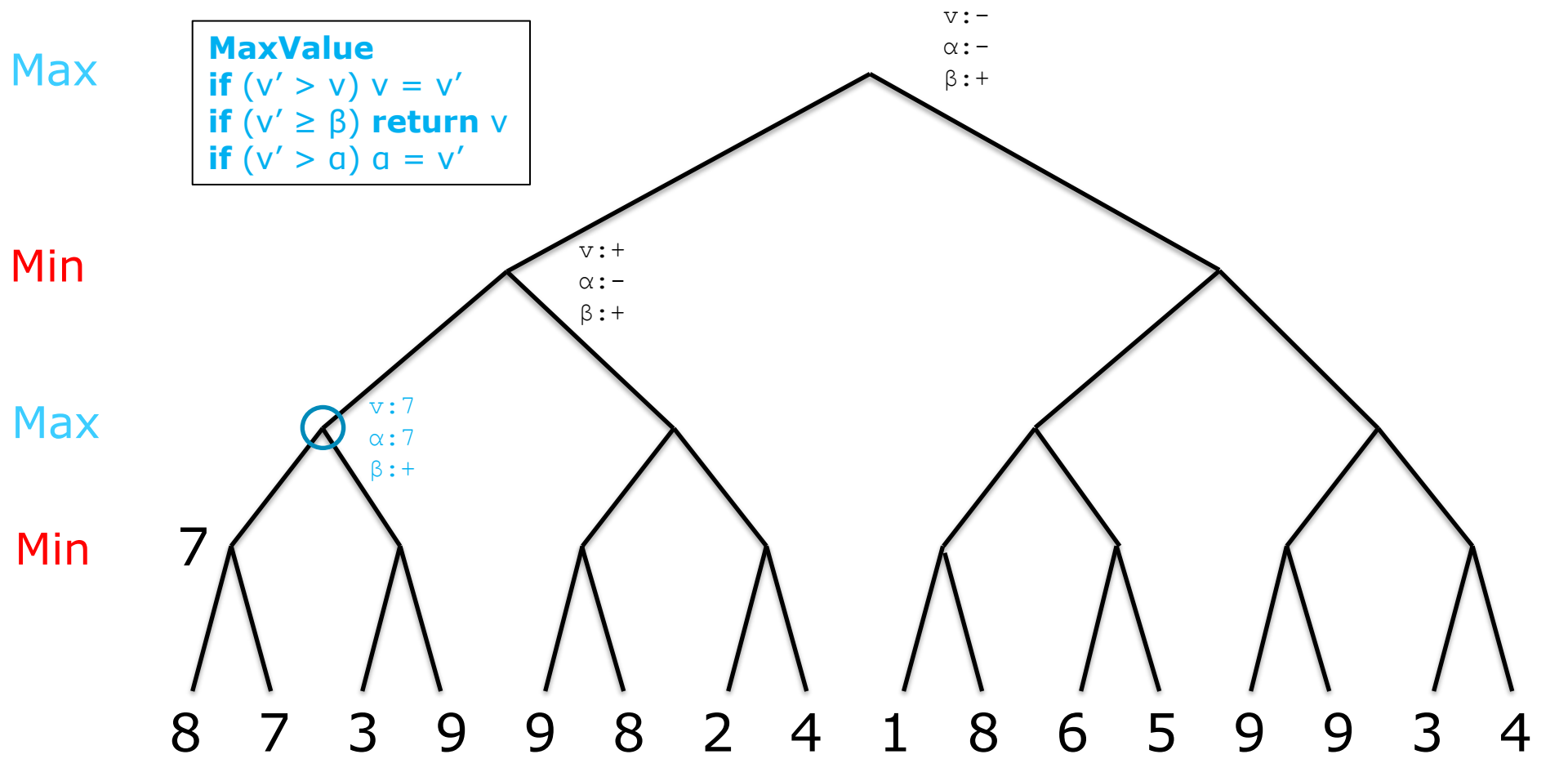

MinValue
if (v′ < v) v = v′
if (v′ ≤ α) return v
if (v′ < β) β = v′
Max
Min
Max
Min
v : −
α : −
β : +
v : +
α : −
β : +
v : −
α : −
β : +
v : 8
α : −
β : 8
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

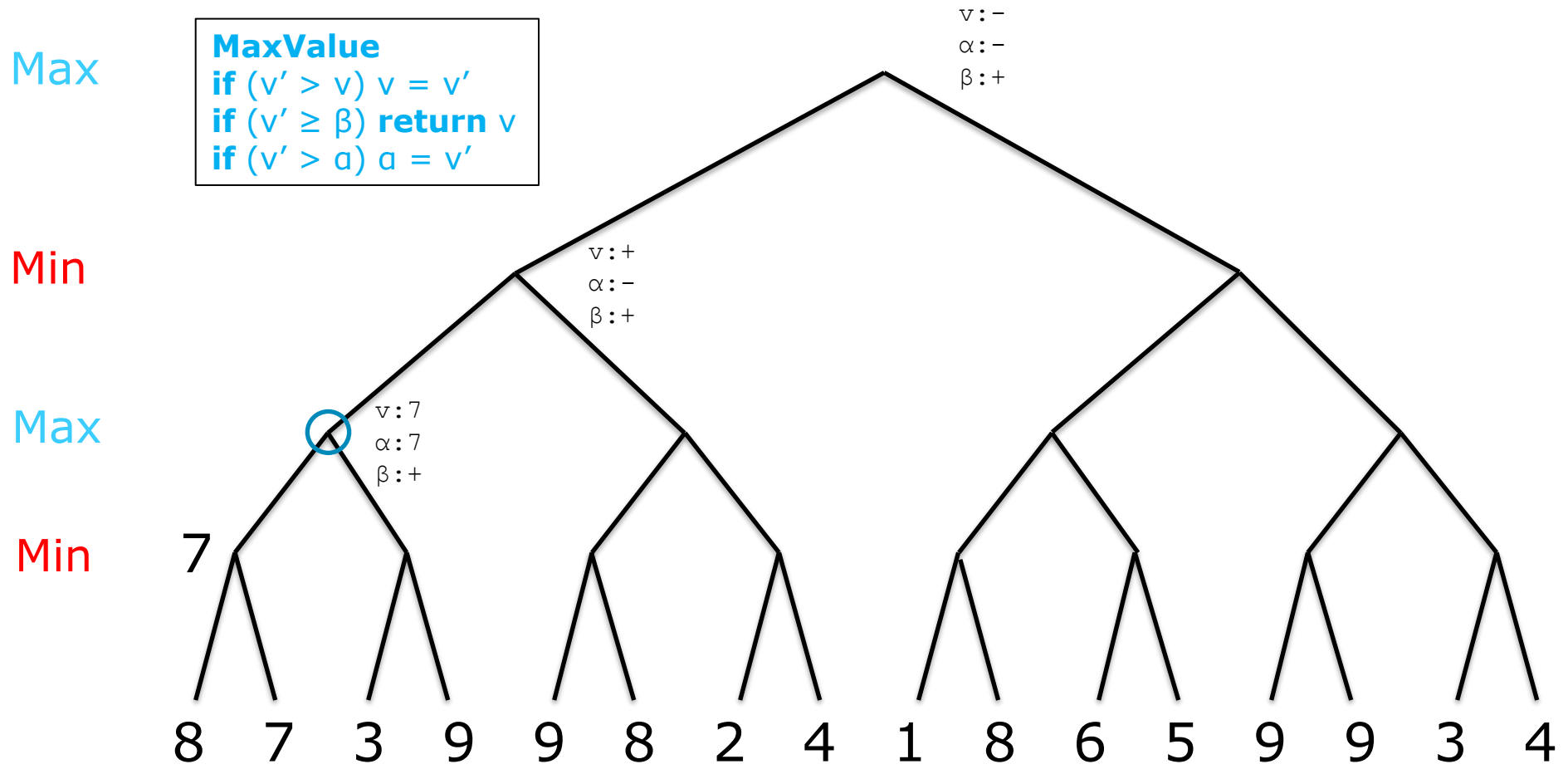

MinValue
if (v′ < v) v = v′
if (v′ ≤ α) return v
if (v′ < β) β = v′
Max
Min
Max
Min
v : −
α : −
β : +
v : +
α : −
β : +
v : −
α : −
β : +
v : 8
α : −
β : 8
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

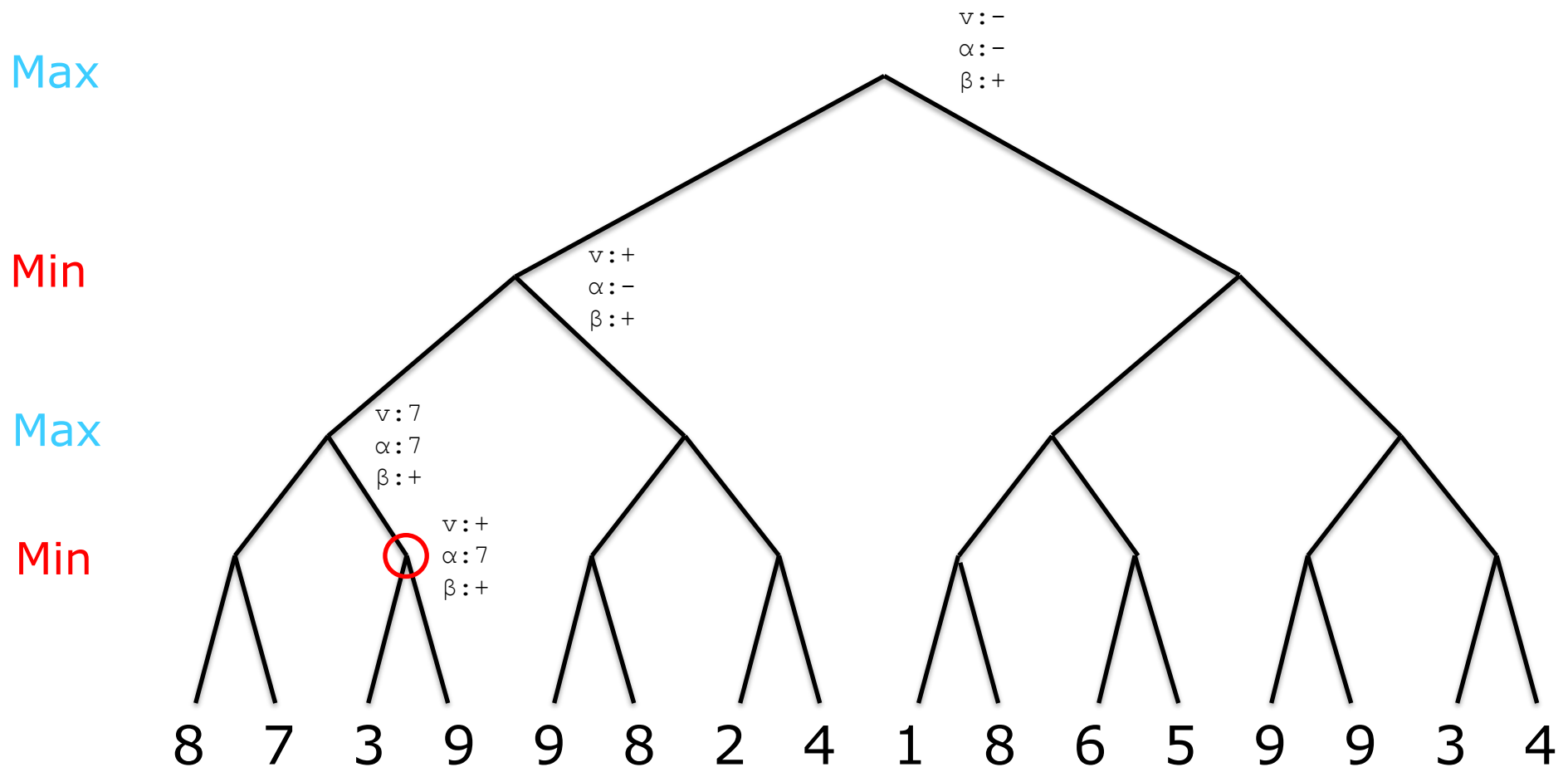

Max
MinValue
if (v′ < v) v = v′
if (v′ ≤ α) return v
if (v′ < β) β = v′
Min
Max
Min
v : −
α : −
β : +
v : +
α : −
β : +
v : −
α : −
β : +
v : 7
α : −
β : 7
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

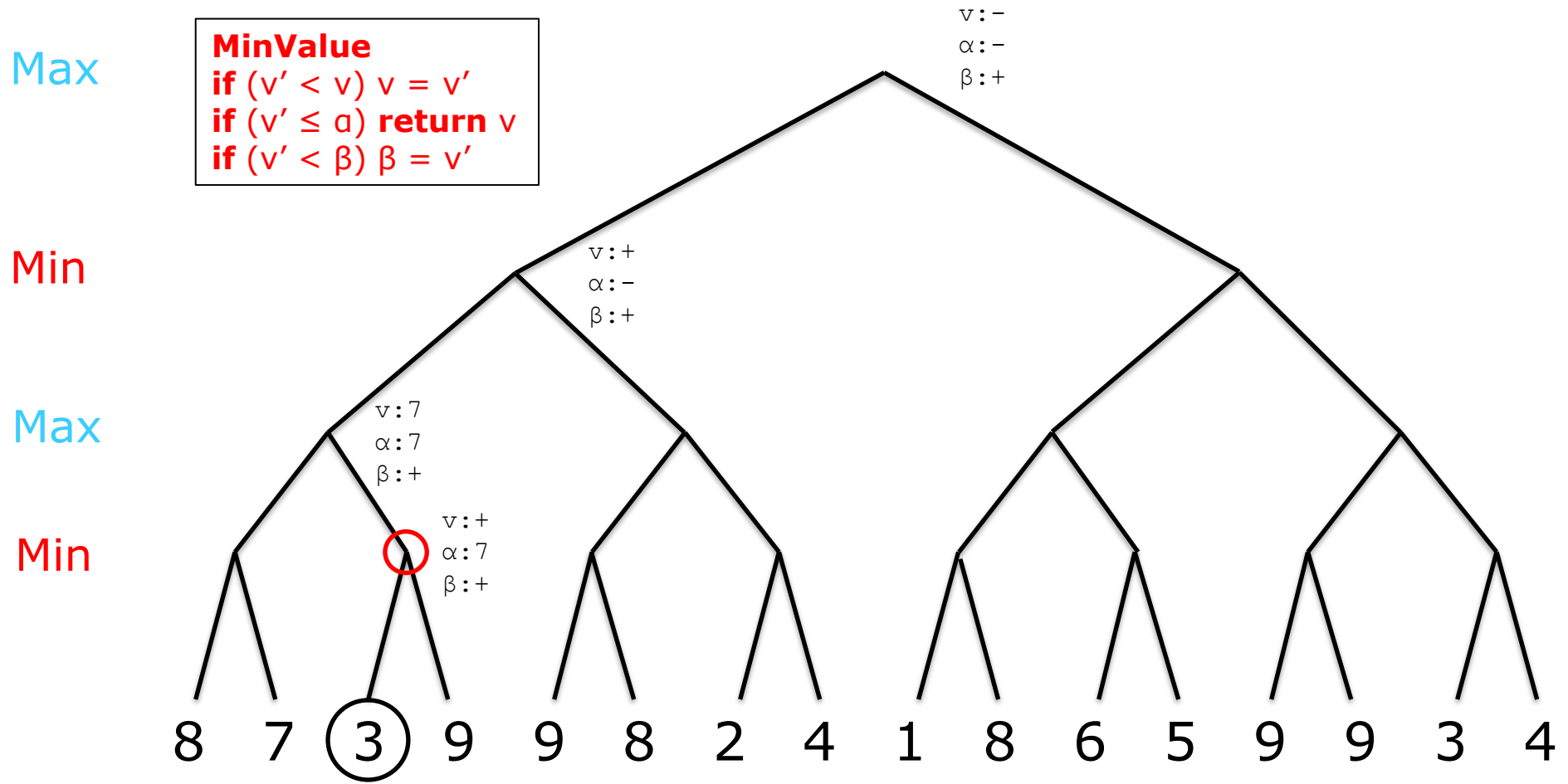

MinValue
if (v′ < v) v = v′
if (v′ ≤ α) return v
if (v′ < β) β = v′
Max
Min
Max
Min
v : −
α : −
β : +
v : +
α : −
β : +
v : −
α : −
β : +
v : 7
α : −
β : 7
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

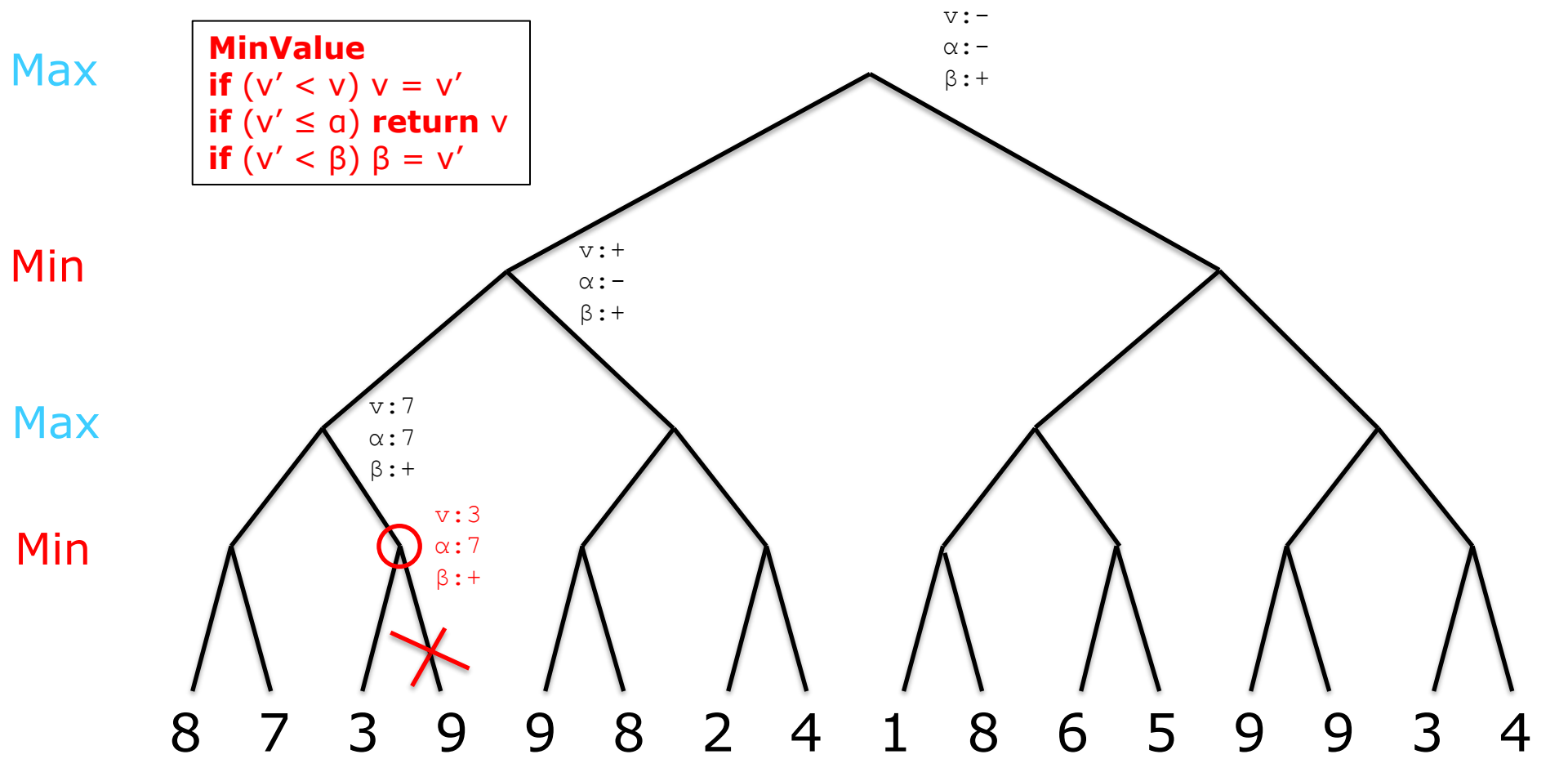

MaxValue
if (v′ > v) v = v′
if (v′ ≥ β) return v
if (v′ > α) α = v′
Max
Min
Max
Min
v : −
α : −
β : +
v : +
α : −
β : +
v : −
α : −
β : +
7
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

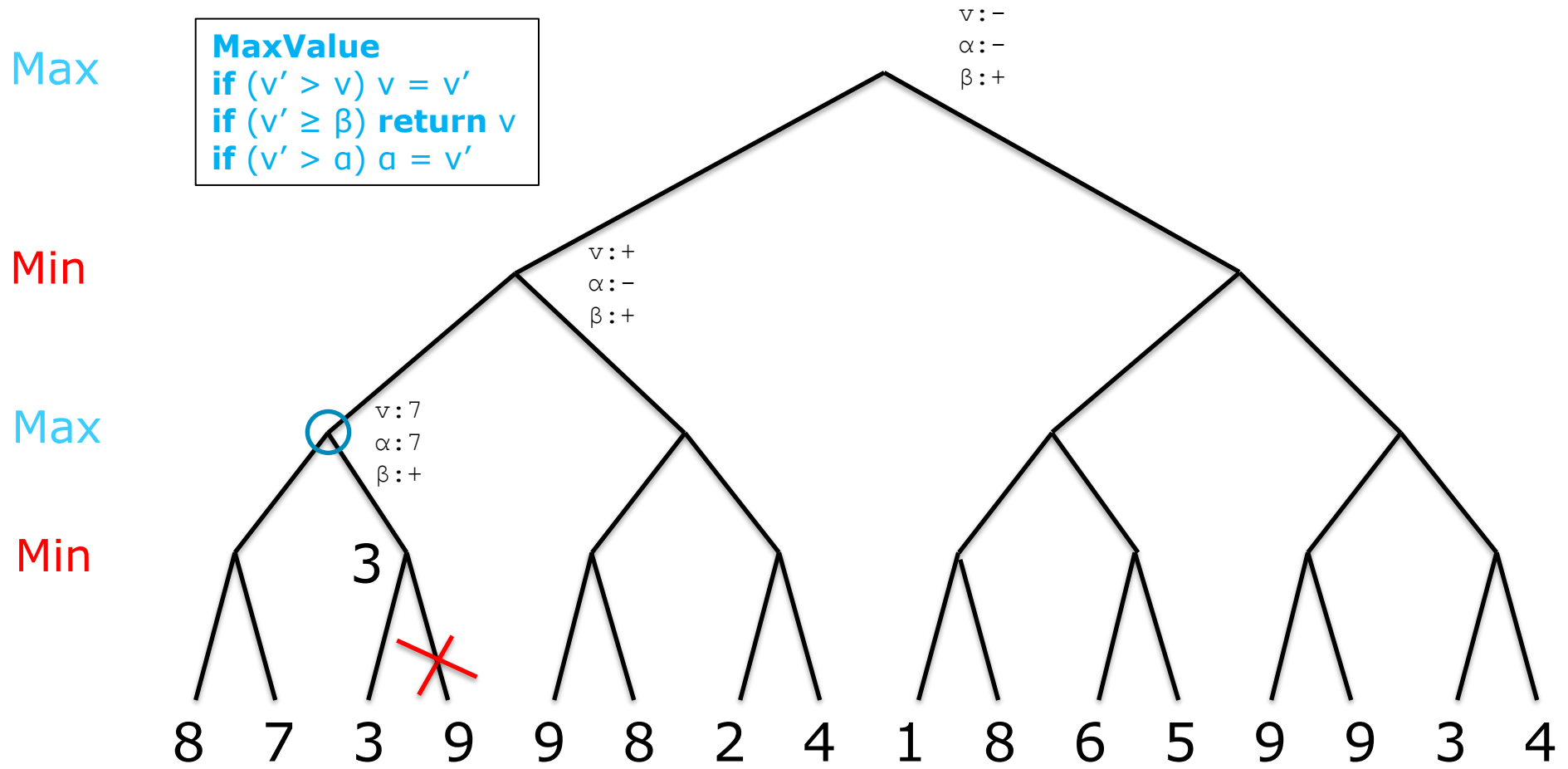

MaxValue
if (v′ > v) v = v′
if (v′ ≥ β) return v
if (v′ > α) α = v′
Max
Min
Max
Min
v : −
α : −
β : +
v : +
α : −
β : +
v : 7
α : 7
β : +
7
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

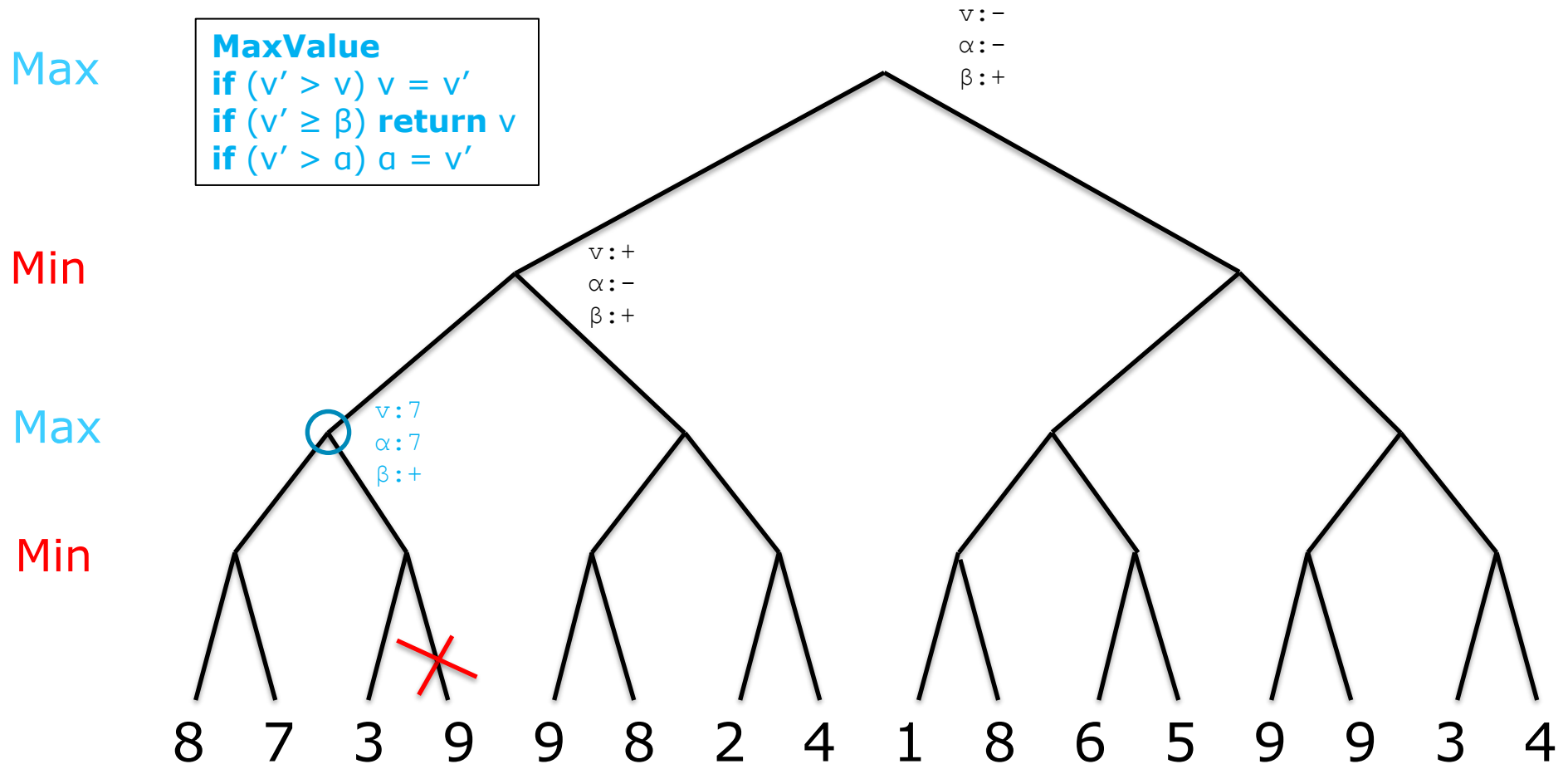

MaxValue
if (v′ > v) v = v′
if (v′ ≥ β) return v
if (v′ > α) α = v′
Max
Min
Max
Min
v : −
α : −
β : +
v : +
α : −
β : +
v : 7
α : 7
β : +
7
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

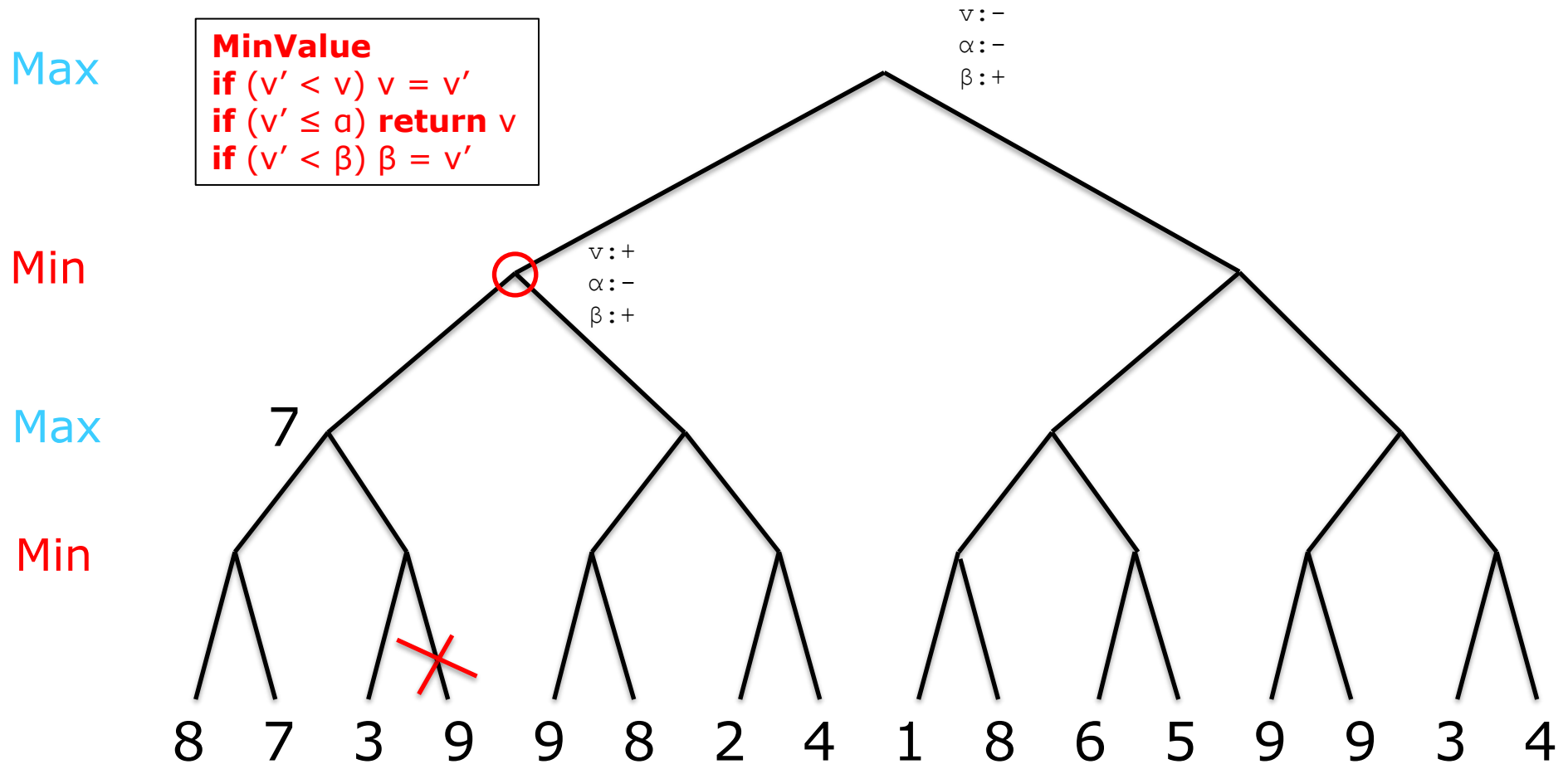

Max
Min
Max
Min
v : −
α : −
β : +
v : +
α : −
β : +
v : 7
α : 7
β : +
v : +
α : 7
β : +
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

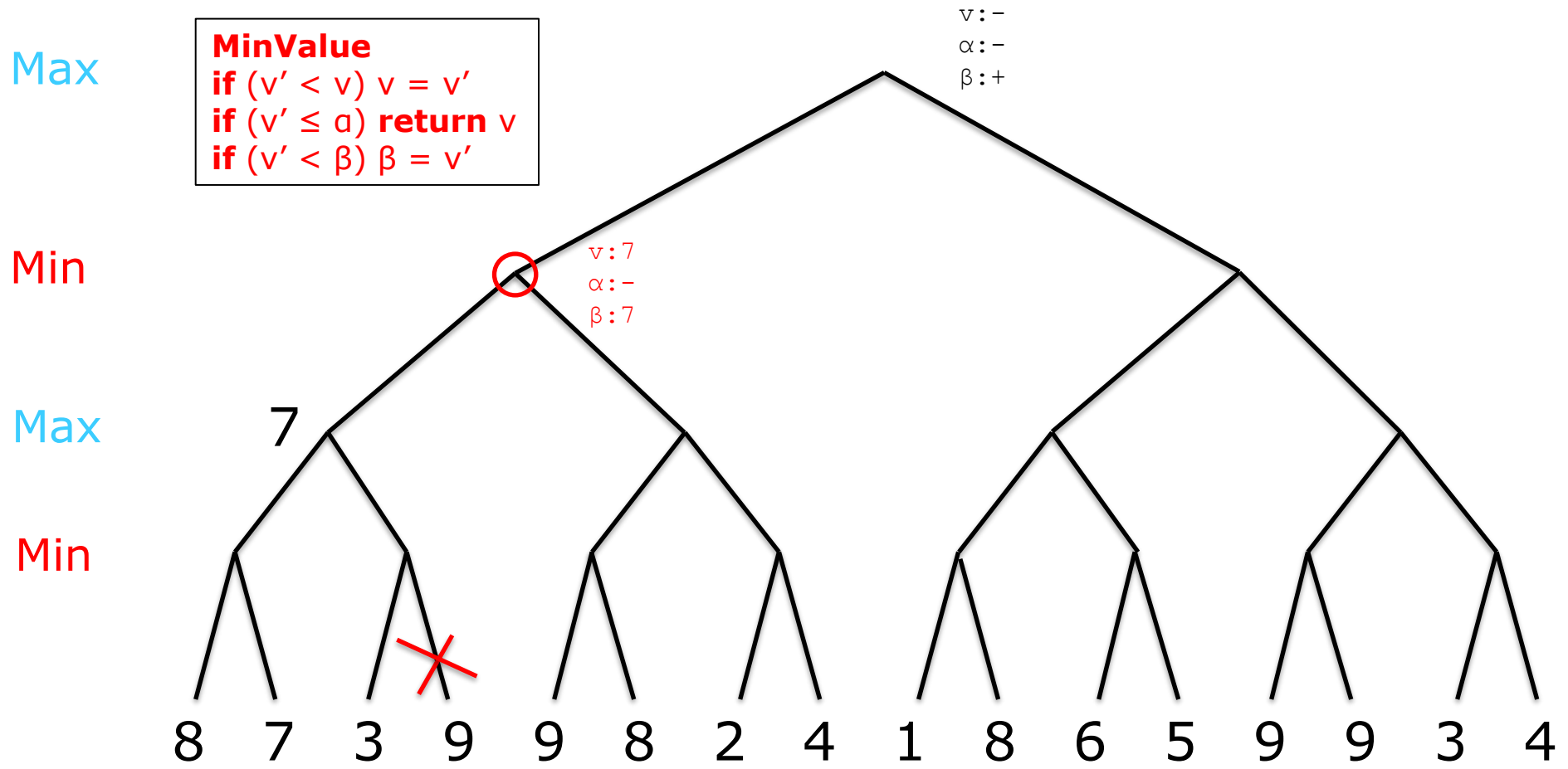

Max
Min
Max
Min
**MinValue**
**if** (v′ < v) v = v′
**if** (v′ ≤ α) **return** v
**if** (v′ < β) β = v′
v : −
α : −
β : +
v : +
α : −
β : +
v : 7
α : 7
β : +
v : +
α : 7
β : +
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

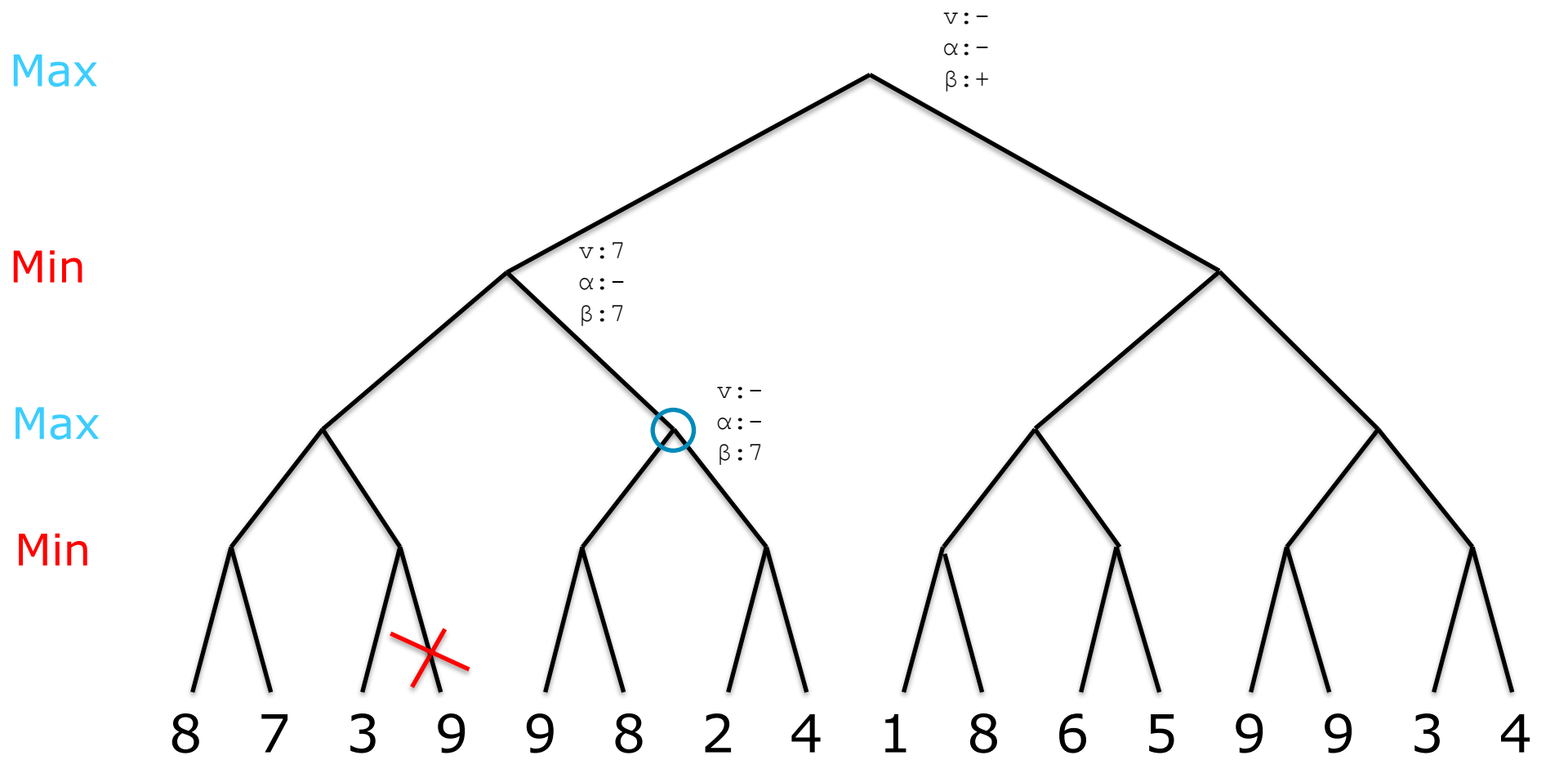

Max
Min
Max
Min
MinValue
if (v′ < v) v = v′
if (v′ ≤ α) return v
if (v′ < β) β = v′
v : −
α : −
β : +
v : +
α : −
β : +
v : 7
α : 7
β : +
v : 3
α : 7
β : +
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

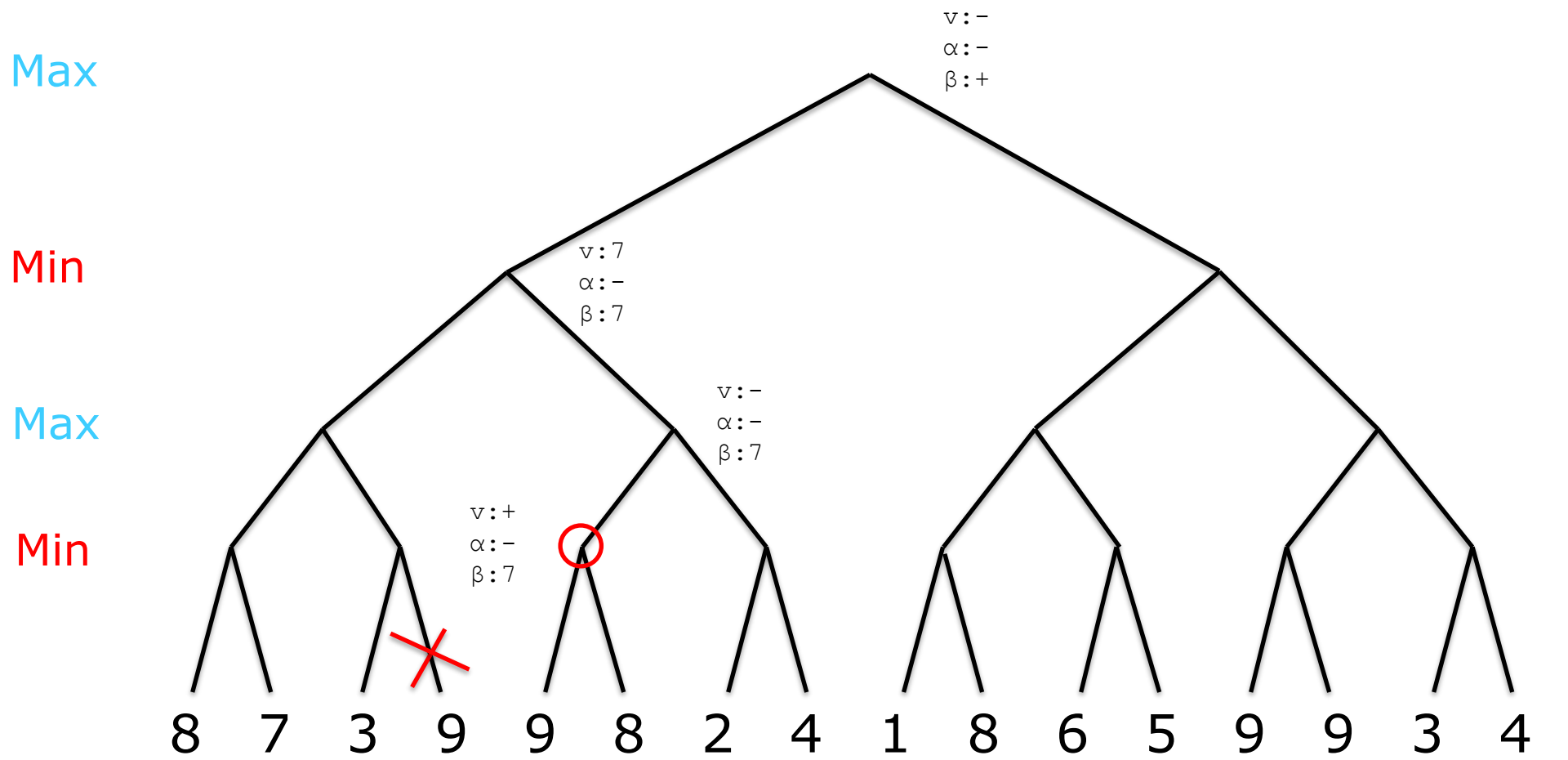

Max
MaxValue
if (v′ > v) v = v′
if (v′ ≥ β) return v
if (v′ > α) α = v′
Min
Max
Min
v : −
α : −
β : +
v : +
α : −
β : +
v : 7
α : 7
β : +
3
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4


MaxValue
if (v′ > v) v = v′
if (v′ ≥ β) return v
if (v′ > α) α = v′
Max
Min
Max
Min
v : −
α : −
β : +
v : +
α : −
β : +
v : 7
α : 7
β : +
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4


MinValue
if (v′ < v) v = v′
if (v′ ≤ α) return v
if (v′ < β) β = v′
Max
Min
Max
Min
v : −
α : −
β : +
v : +
α : −
β : +
7
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4


MinValue
if (v′ < v) v = v′
if (v′ ≤ α) return v
if (v′ < β) β = v′
Max
Min
Max
Min
v : −
α : −
β : +
v : 7
α : −
β : 7
7
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4


Max
Min
Max
Min
v : –
α : –
β : +
v : 7
α : –
β : 7
v : –
α : –
β : 7
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4


Max
Min
Max
Min
v : −
α : −
β : +
v : 7
α : −
β : 7
v : −
α : −
β : 7
v : +
α : −
β : 7
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

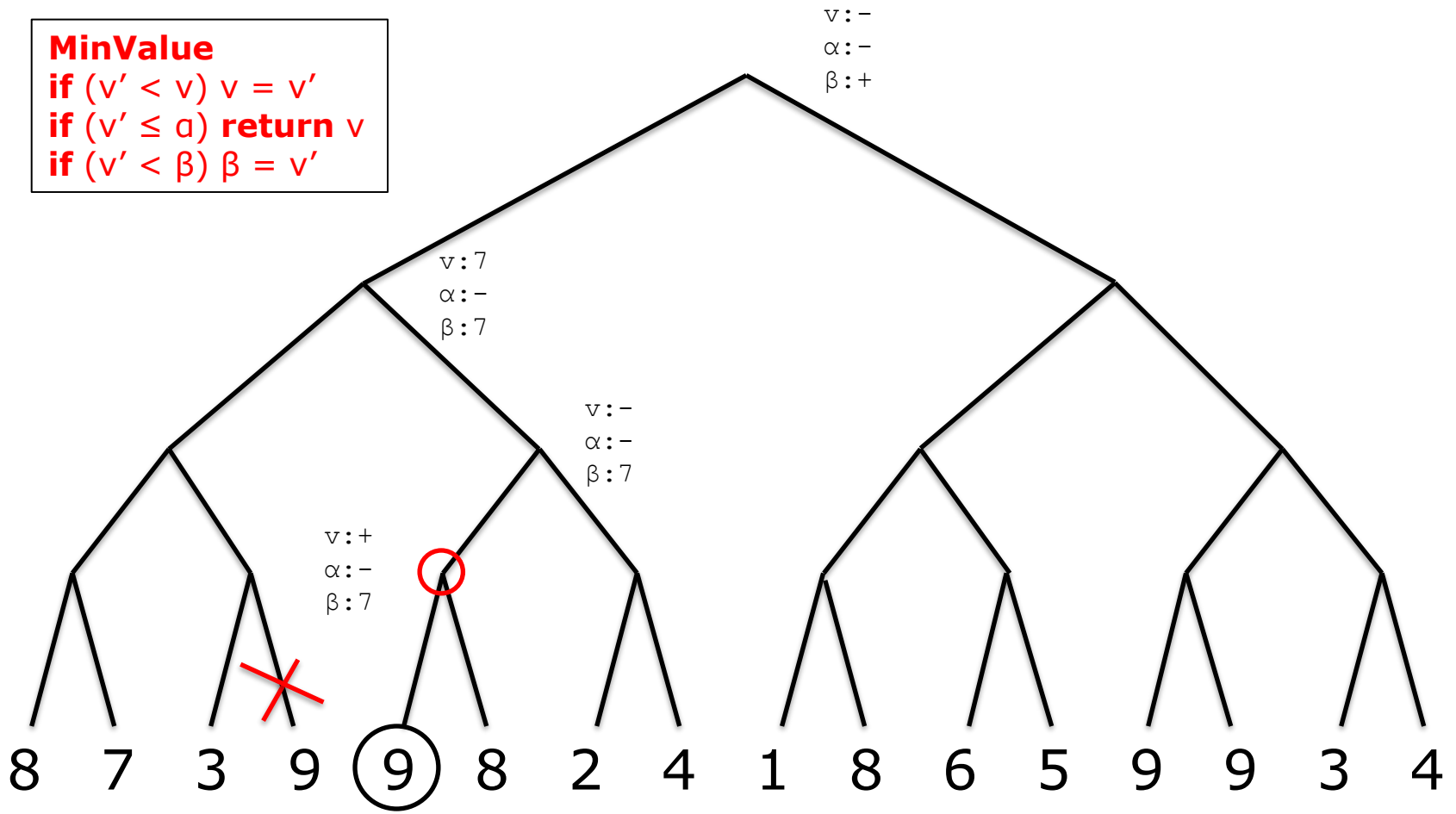
Max

Min

Max

Min

**MinValue**
**if** (v′ < v) v = v′
**if** (v′ ≤ α) **return** v
**if** (v′ < β) β = v′



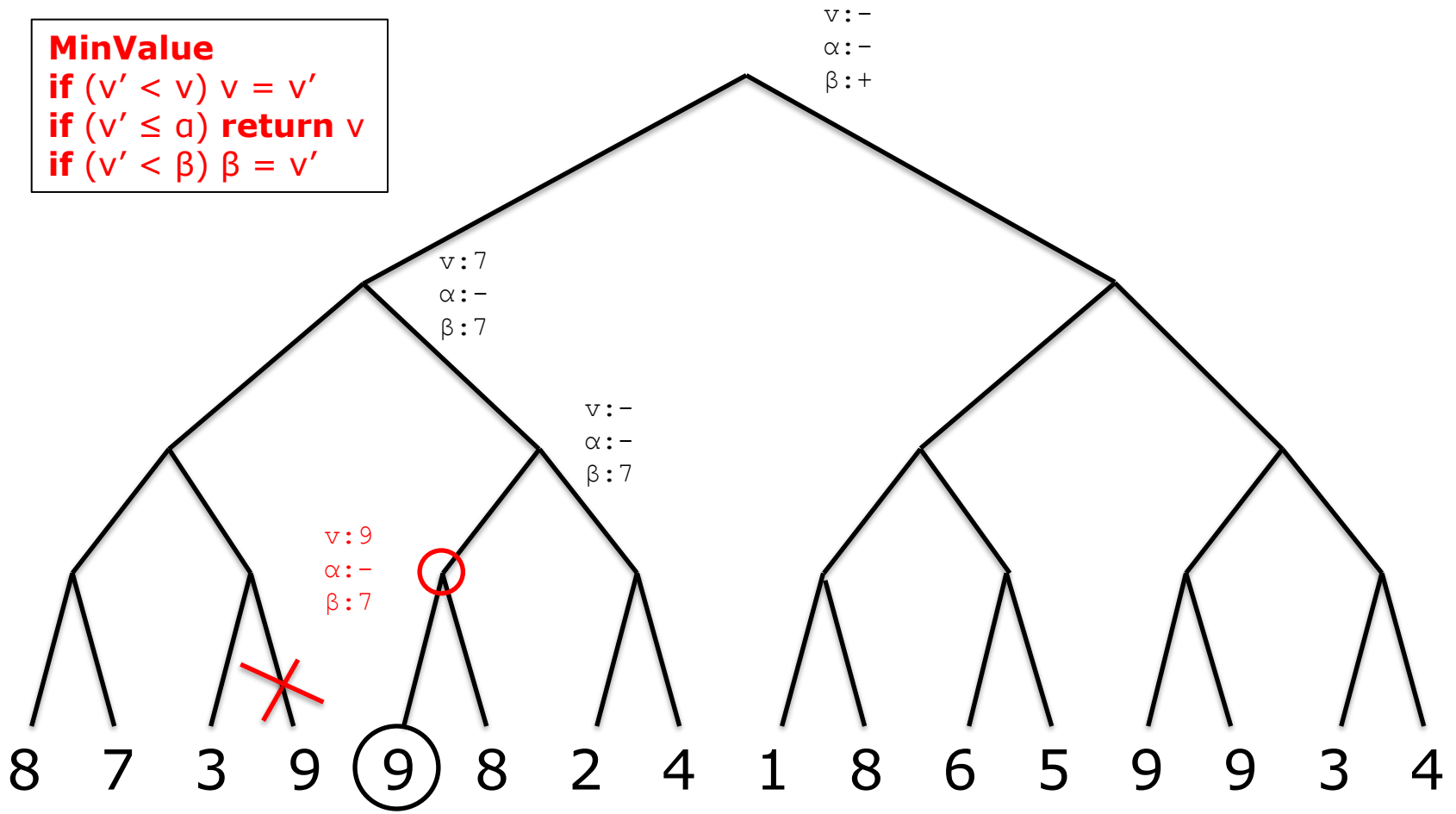
Max

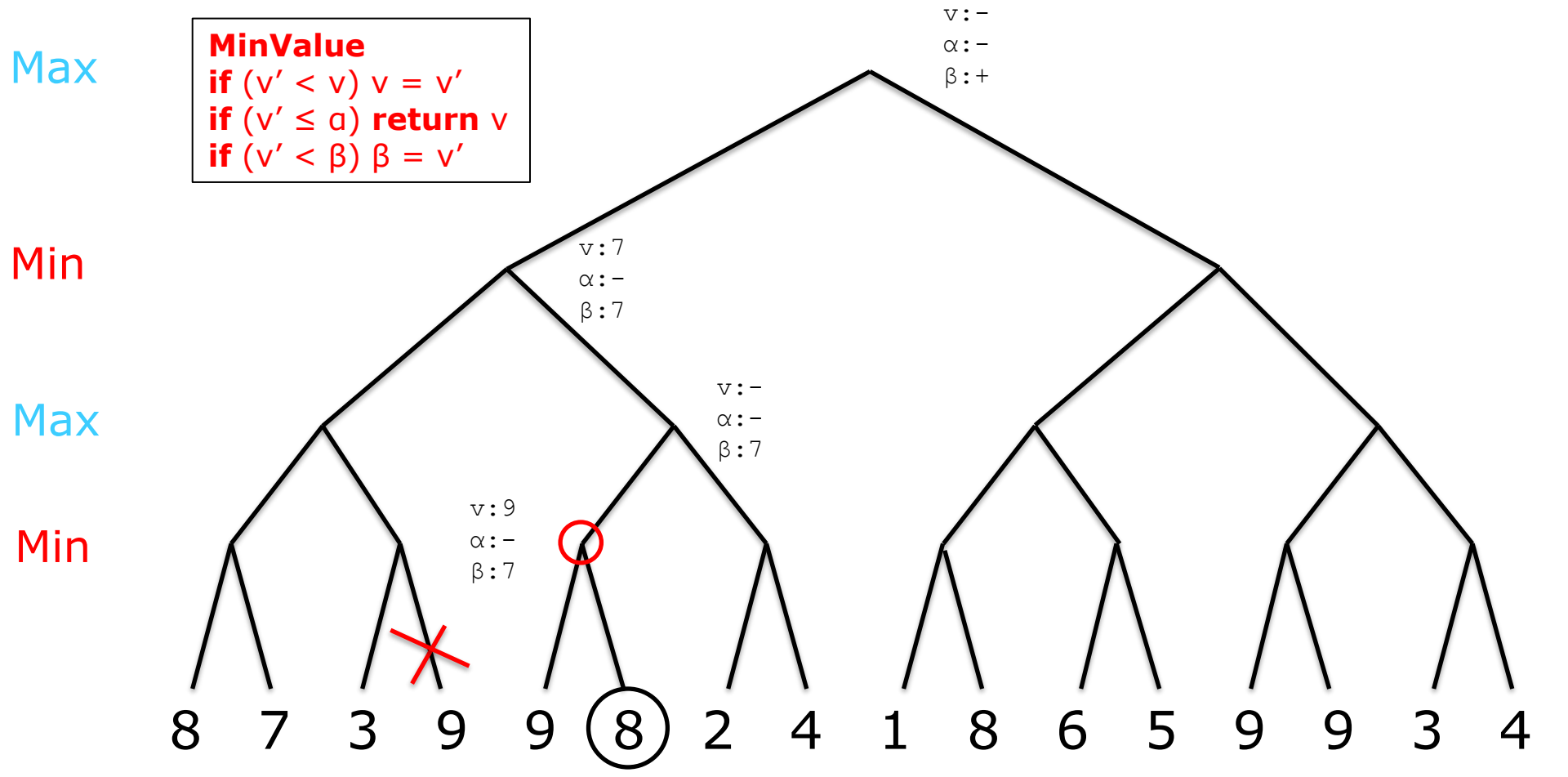
Min

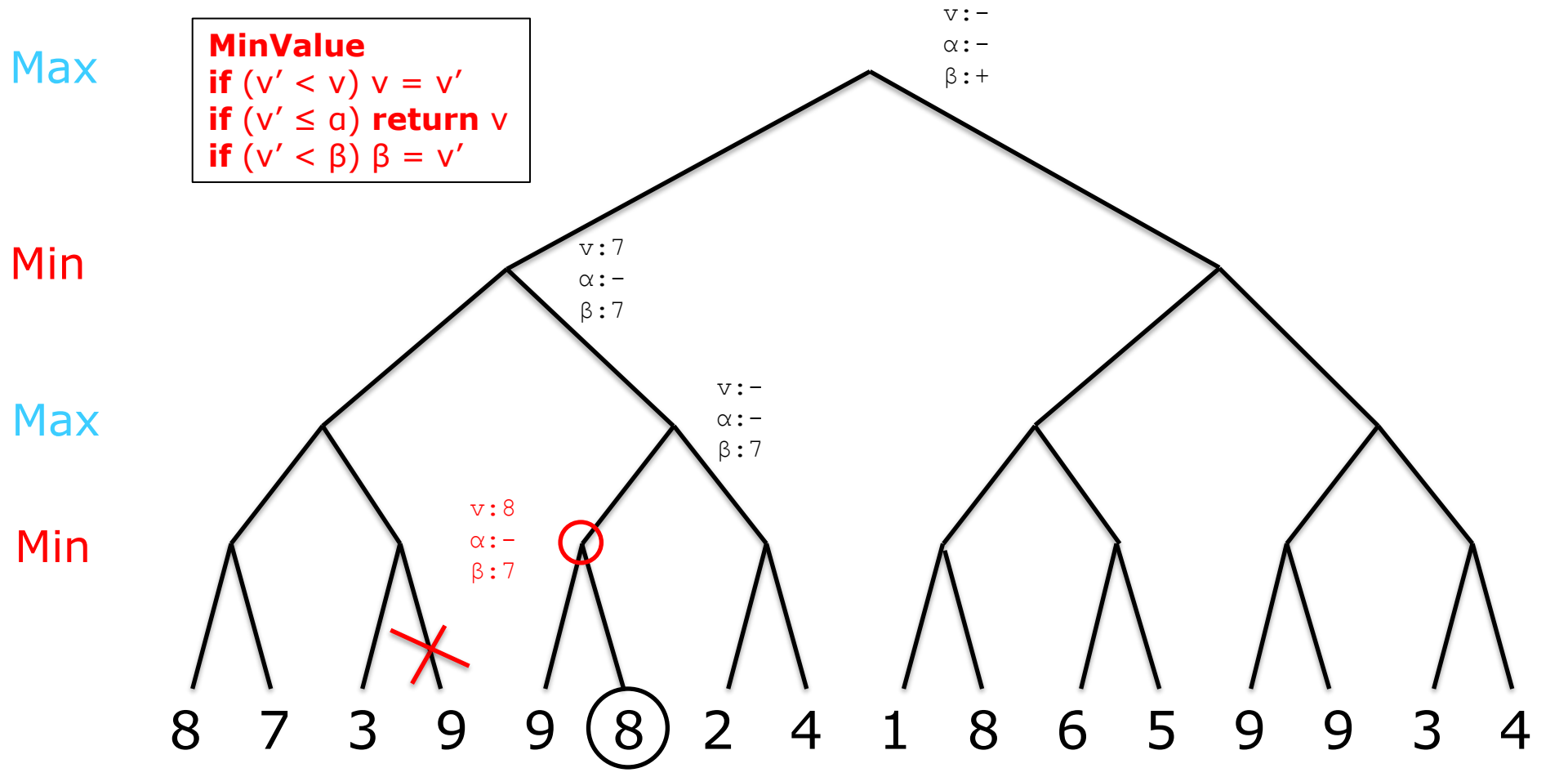
Max

Min

**MinValue**
**if** (v′ < v) v = v′
**if** (v′ ≤ α) **return** v
**if** (v′ < β) β = v′




MinValue
if (v′ < v) v = v′
if (v′ ≤ α) return v
if (v′ < β) β = v′
Max
Min
Max
Min
v : −
α : −
β : +
v : 7
α : −
β : 7
v : −
α : −
β : 7
v : 9
α : −
β : 7
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4


Max
MinValue
if (v′ < v) v = v′
if (v′ ≤ α) return v
if (v′ < β) β = v′
Min
Max
Min
v : −
α : −
β : +
v : 7
α : −
β : 7
v : −
α : −
β : 7
v : 8
α : −
β : 7
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

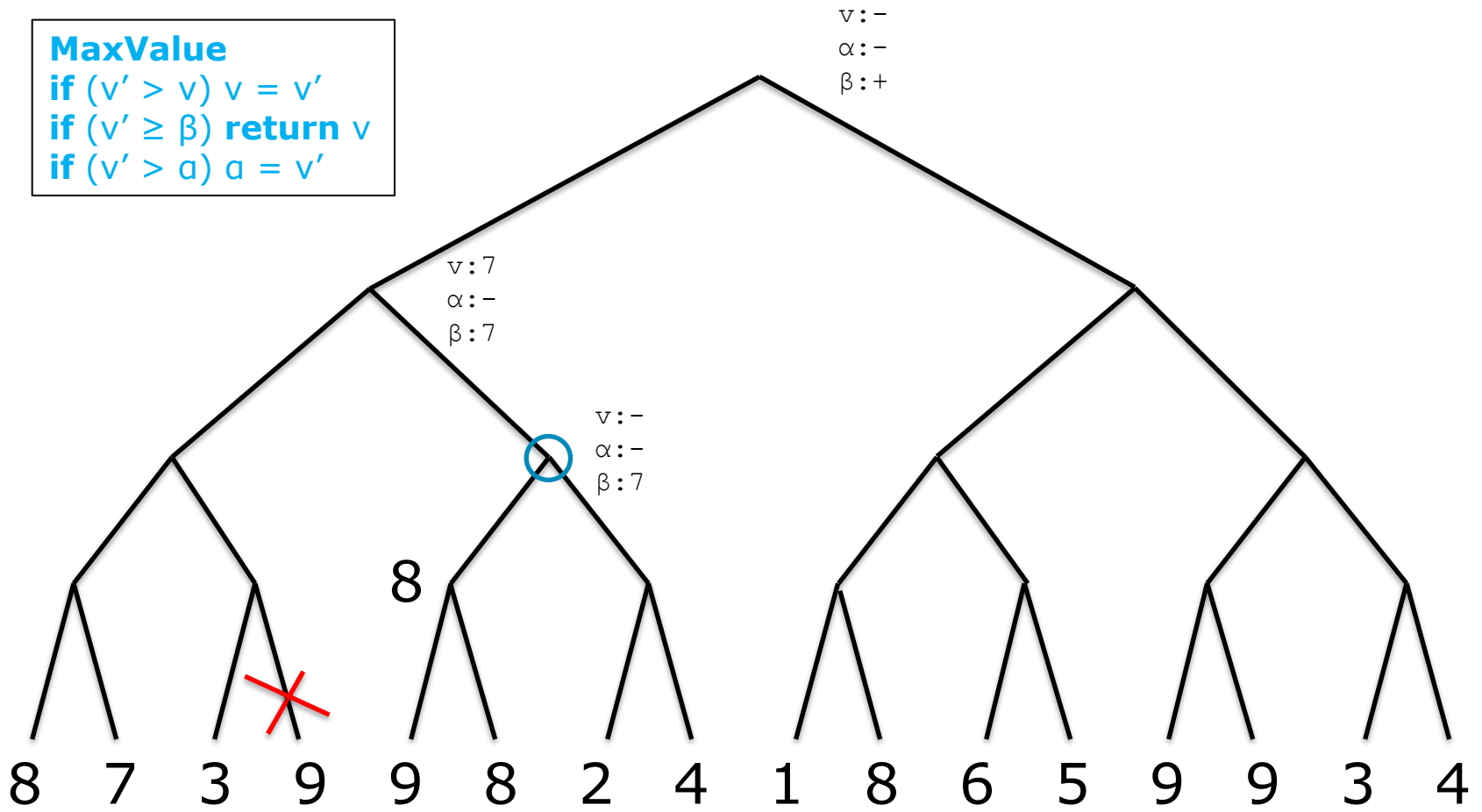
Max

**MaxValue**
**if** (v′ > v) v = v′
**if** (v′ ≥ β) **return** v
**if** (v′ > α) α = v′

Min

Max

Min



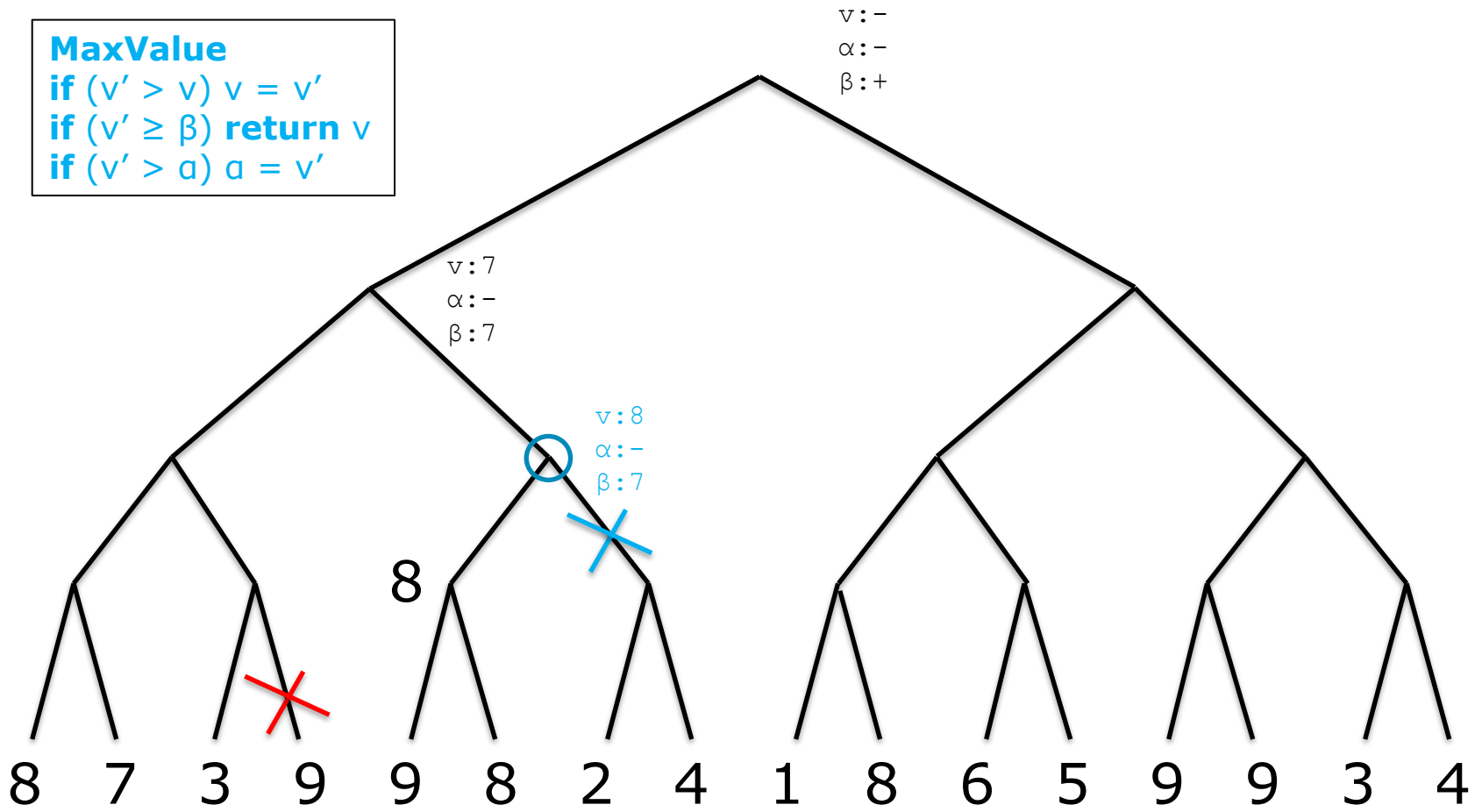
Max

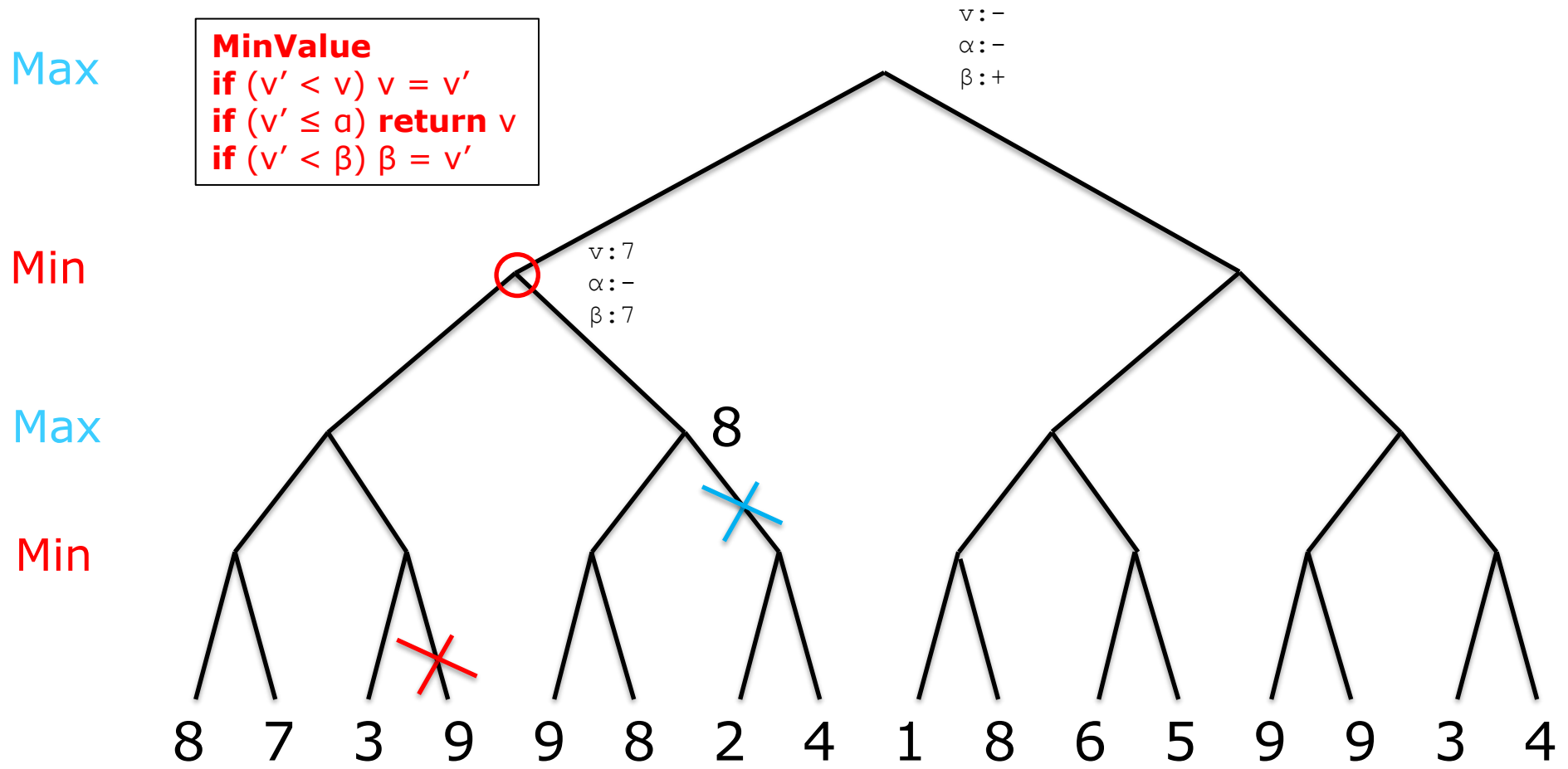
Min

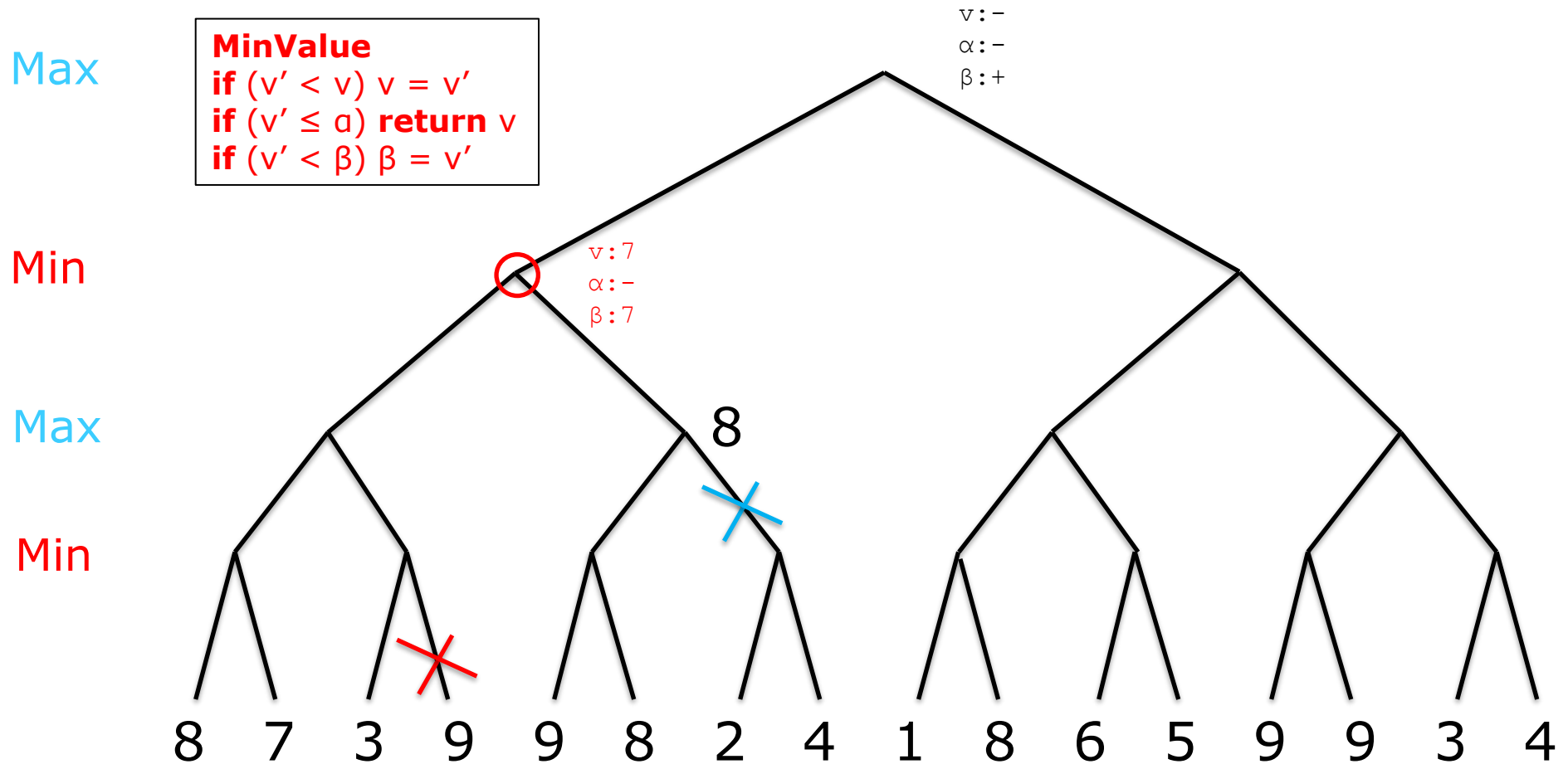
Max

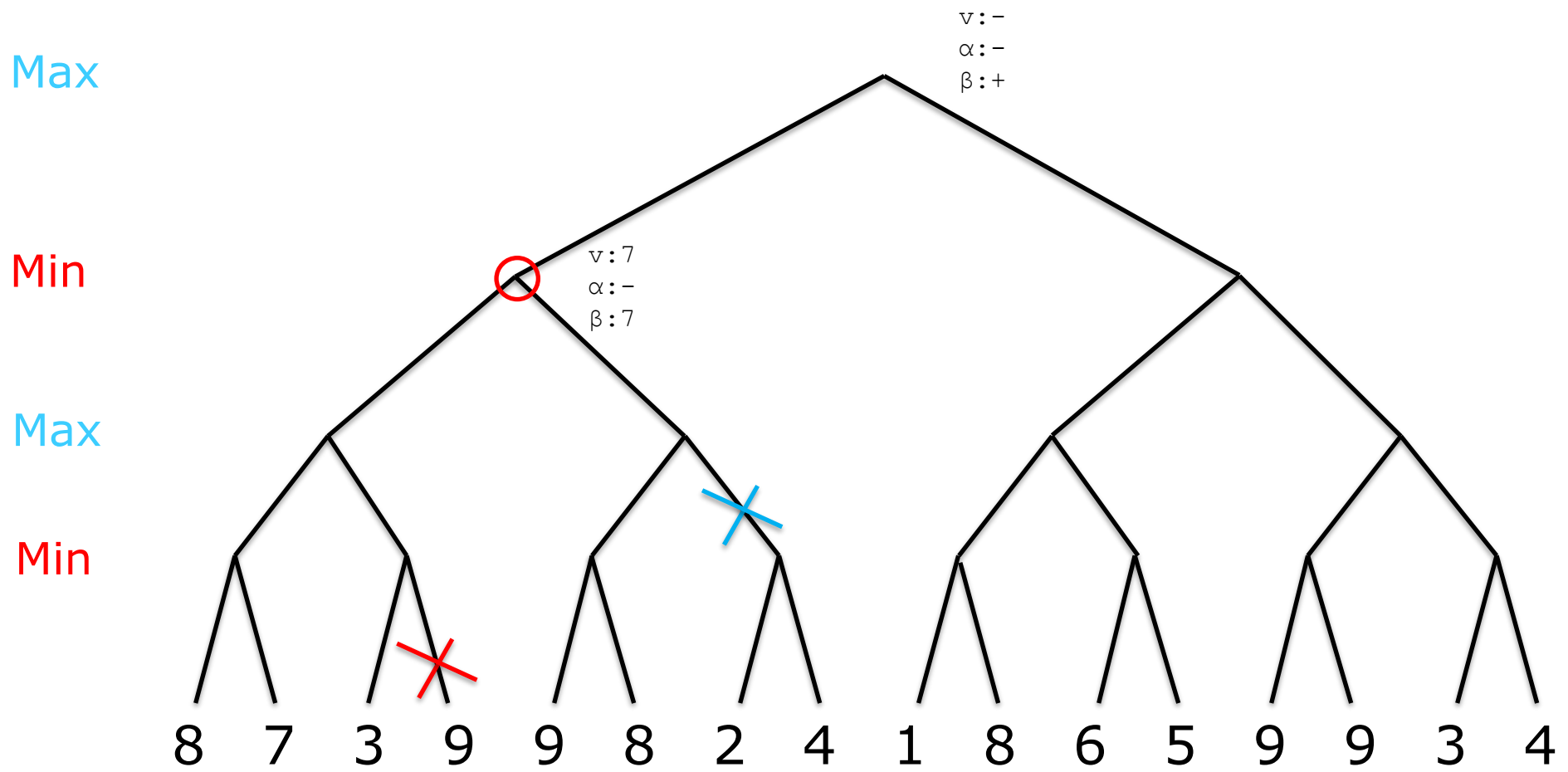
Min

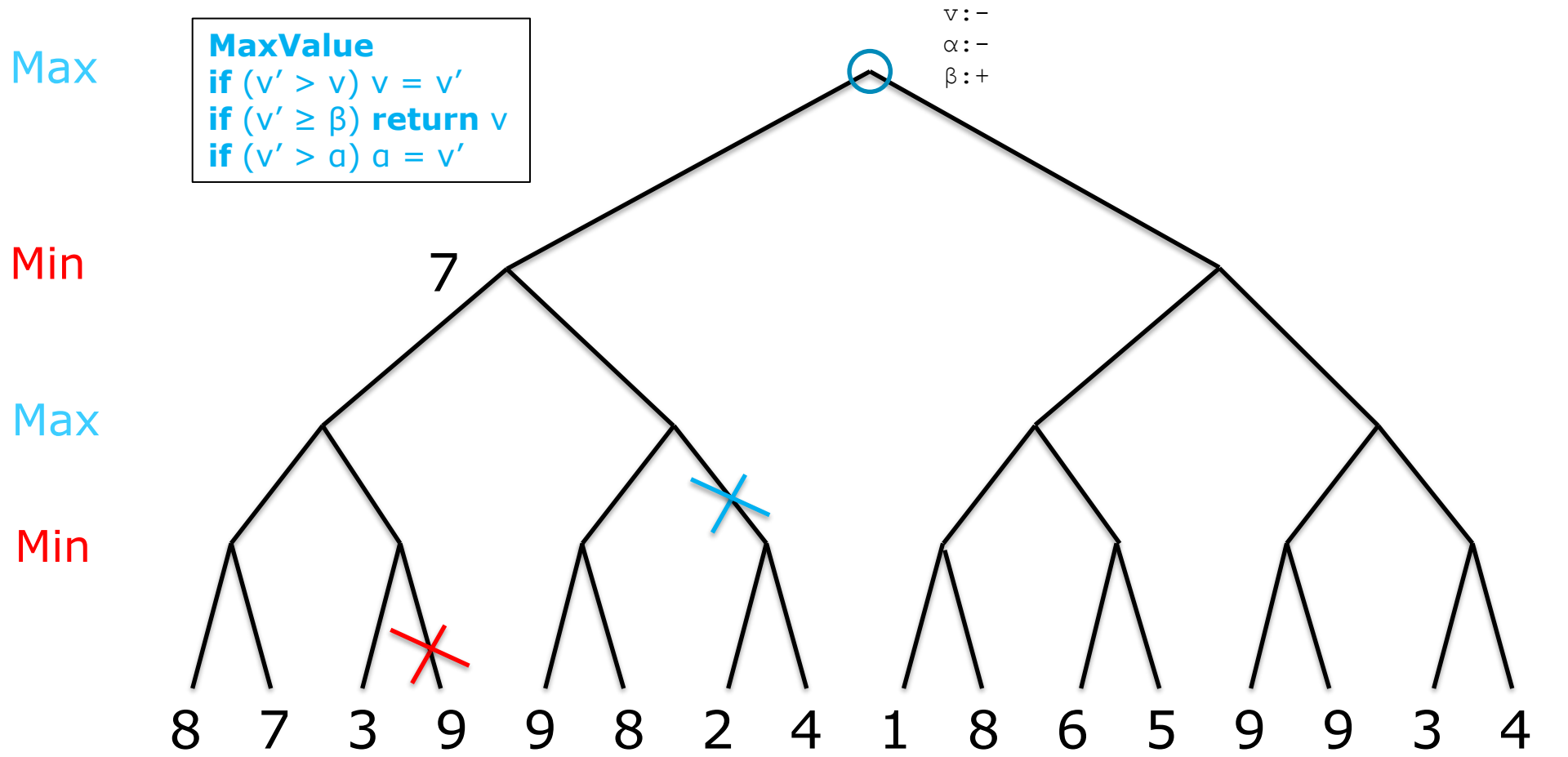
**MaxValue**
**if** (v′ > v) v = v′
**if** (v′ ≥ β) **return** v
**if** (v′ > α) α = v′

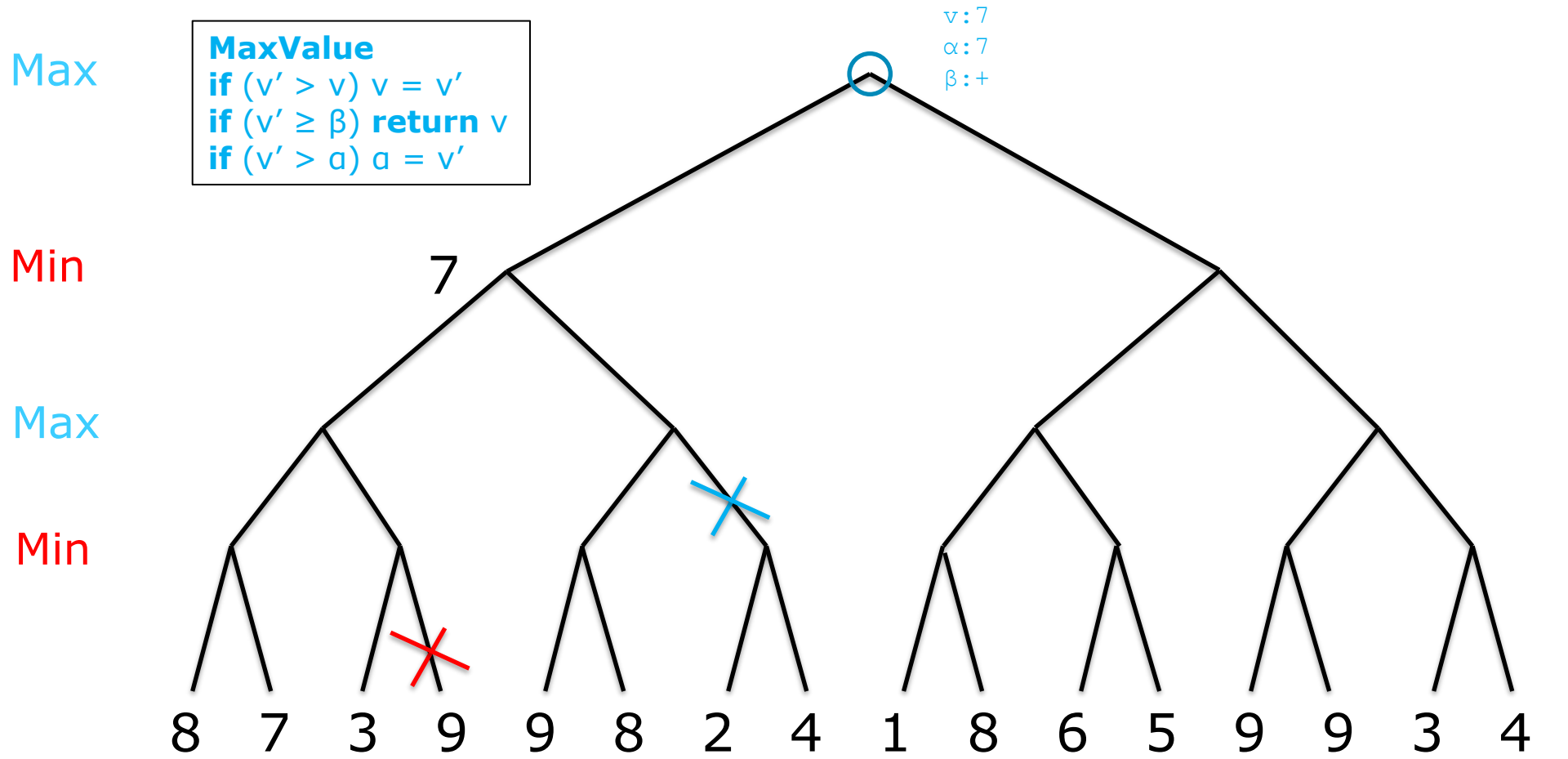


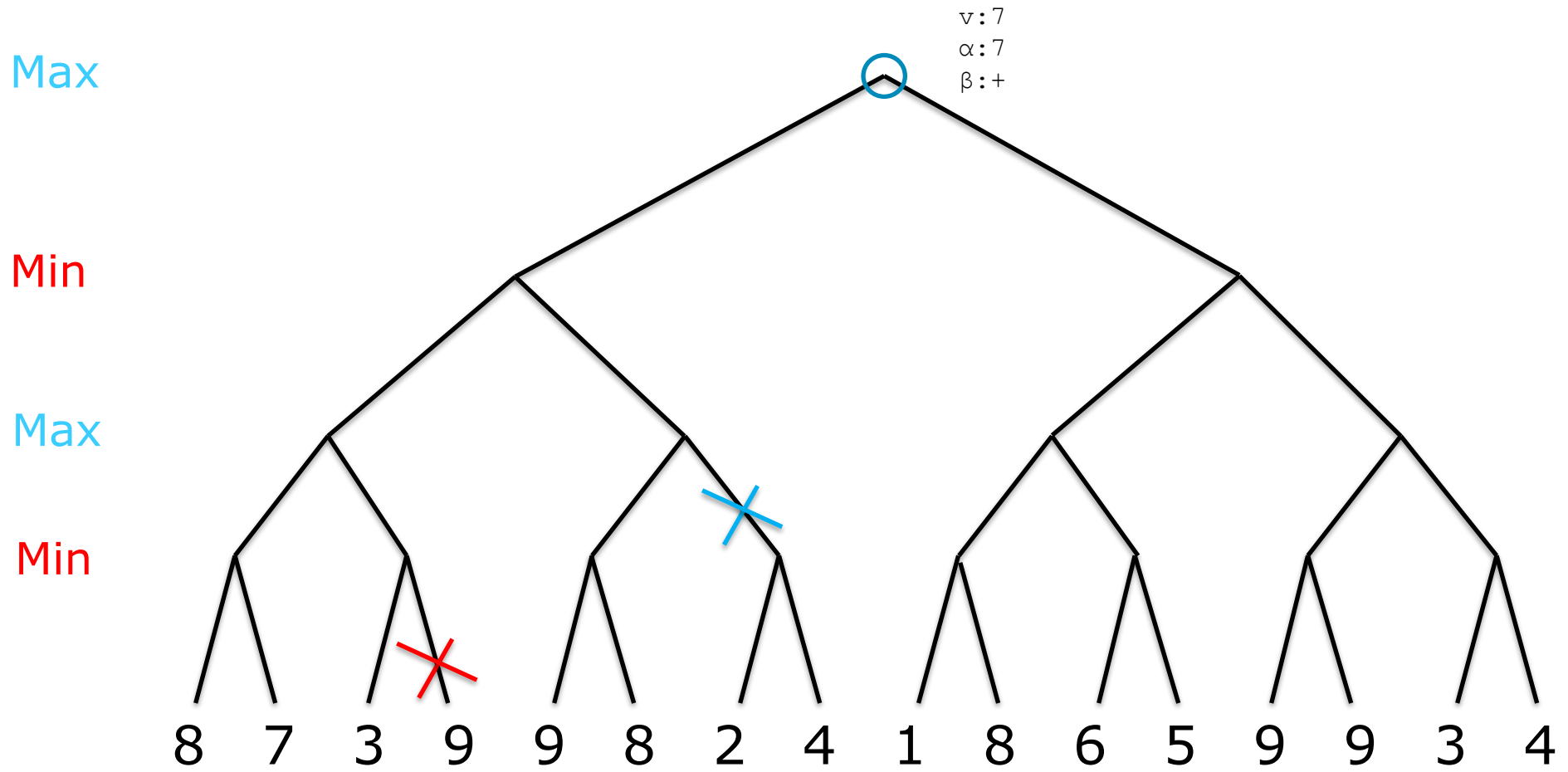

MinValue
if (v′ < v) v = v′
if (v′ ≤ α) return v
if (v′ < β) β = v′
Max
Min
Max
Min
v : −
α : −
β : +
v : 7
α : −
β : 7
8
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

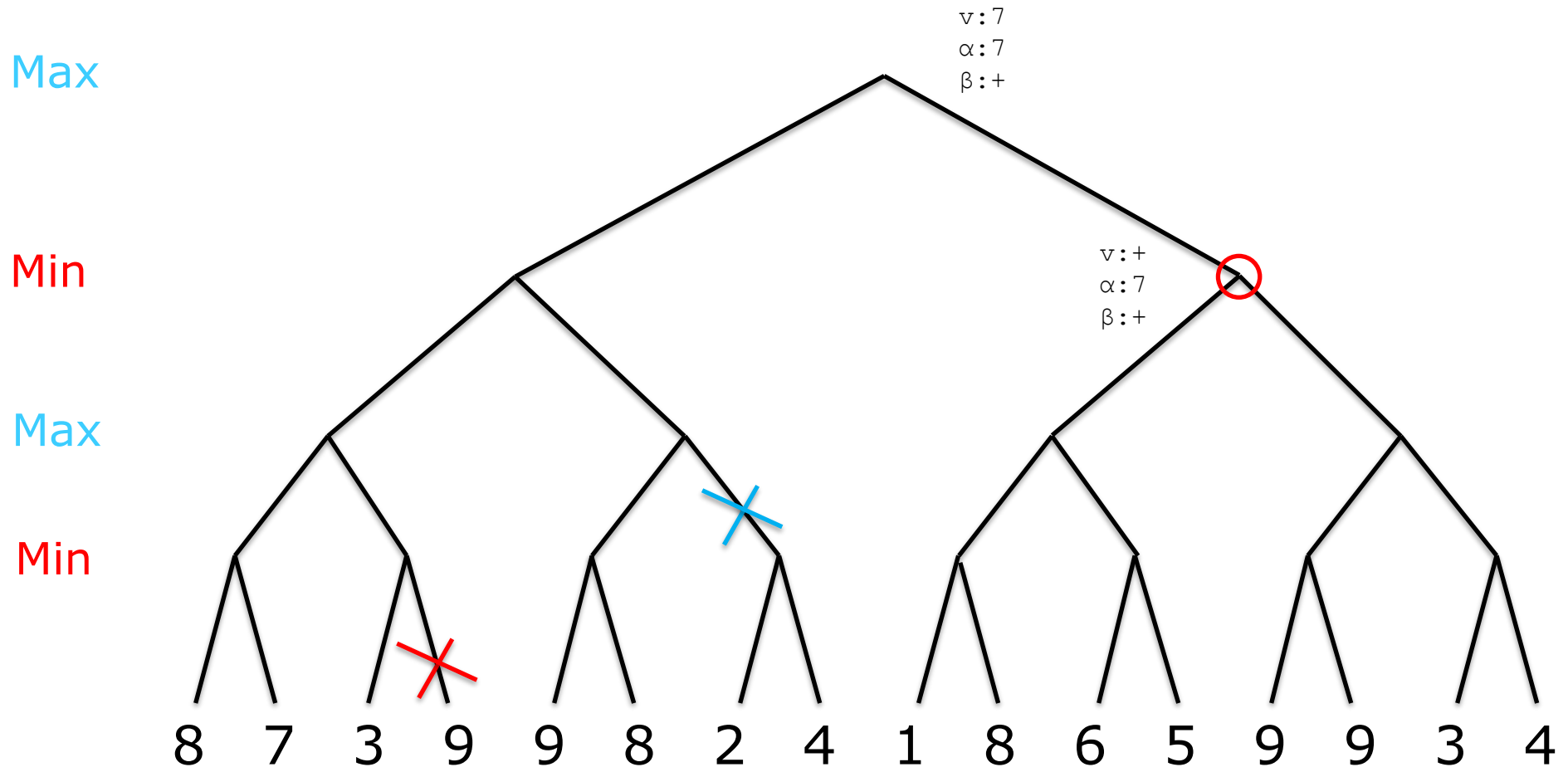

MinValue
if (v′ < v) v = v′
if (v′ ≤ α) return v
if (v′ < β) β = v′
Max
Min
Max
Min
v : −
α : −
β : +
v : 7
α : −
β : 7
8
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

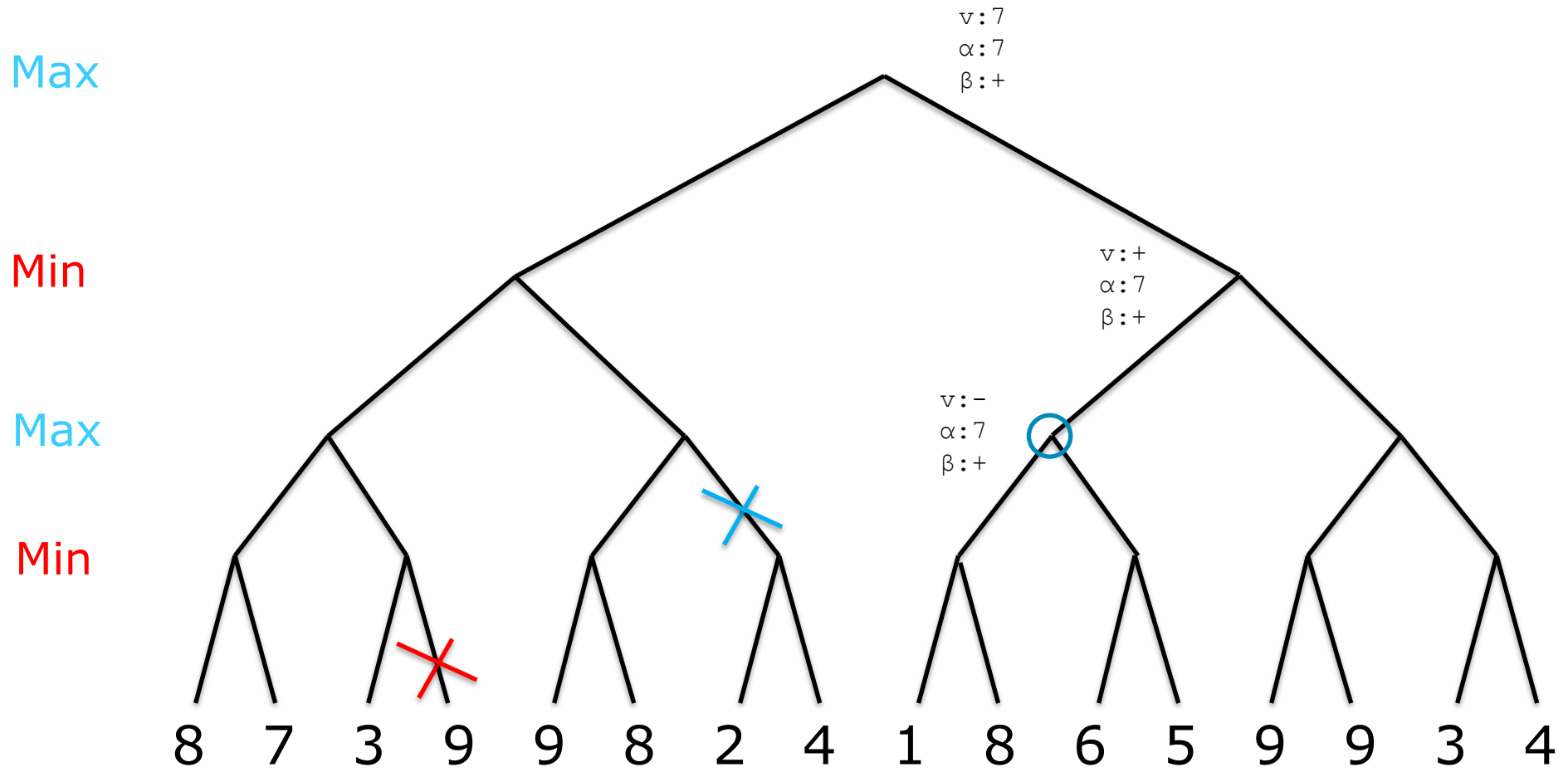

Max
Min
Max
Min
v : –
α : –
β : +
v : 7
α : –
β : 7
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

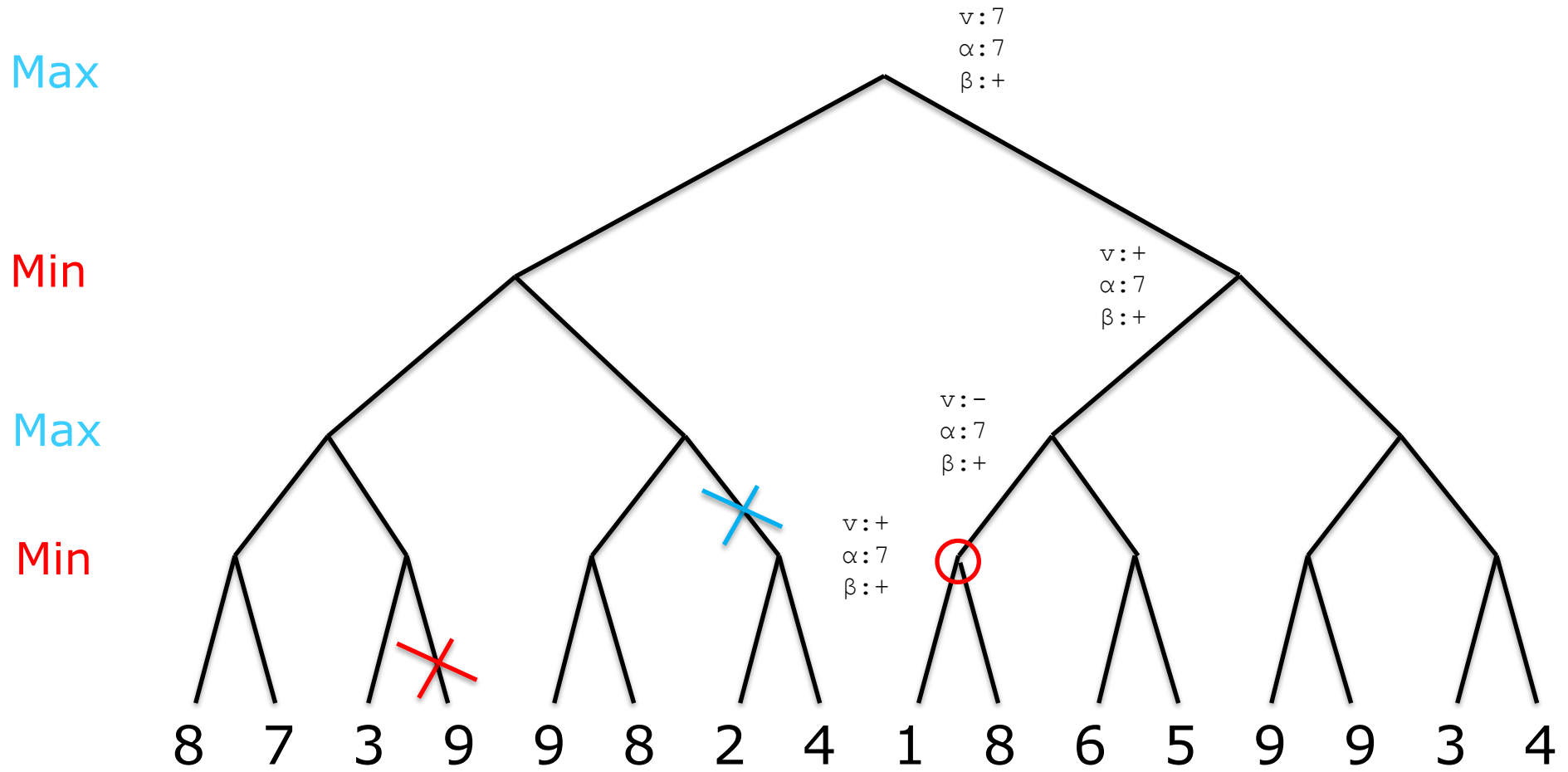

Max
MaxValue
if (v′ > v) v = v′
if (v′ ≥ β) return v
if (v′ > α) α = v′
v : −
α : −
β : +
Min
7
Max
Min
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

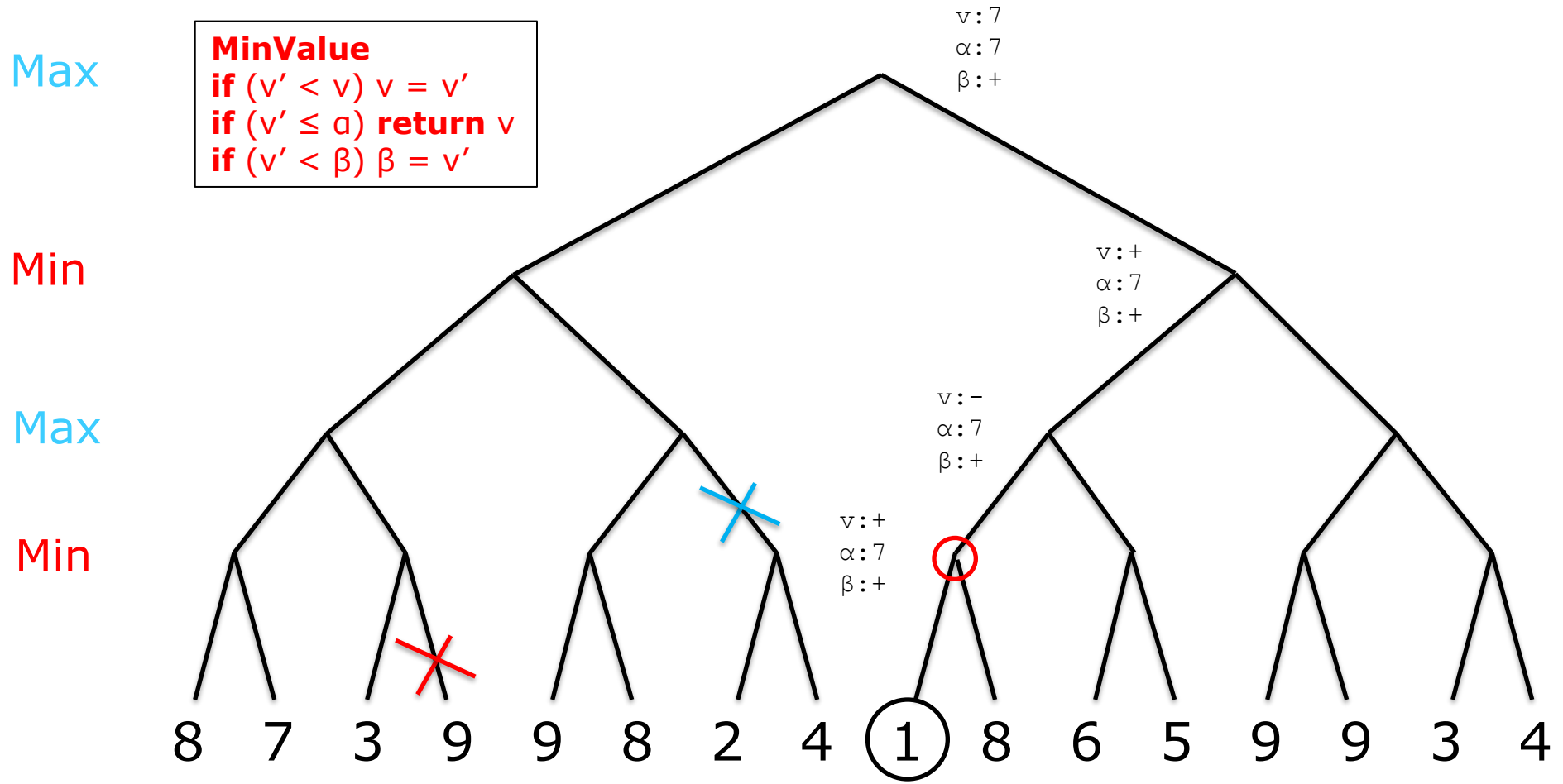

MaxValue
if (v′ > v) v = v′
if (v′ ≥ β) return v
if (v′ > α) α = v′
v : 7
α : 7
β : +
Max
Min
Max
Min
7
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

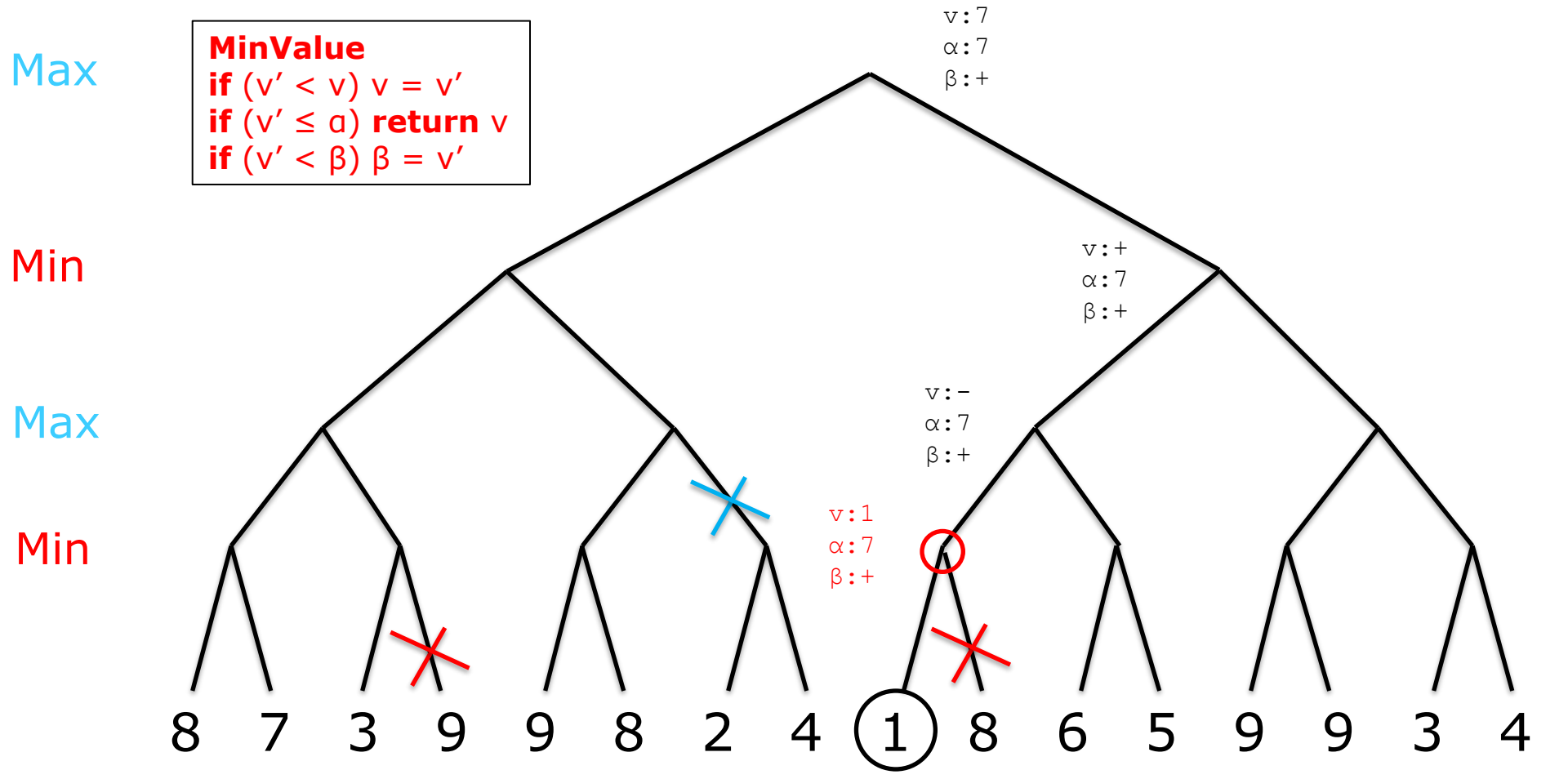

v : 7
α : 7
β : +
Max
Min
Max
Min
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

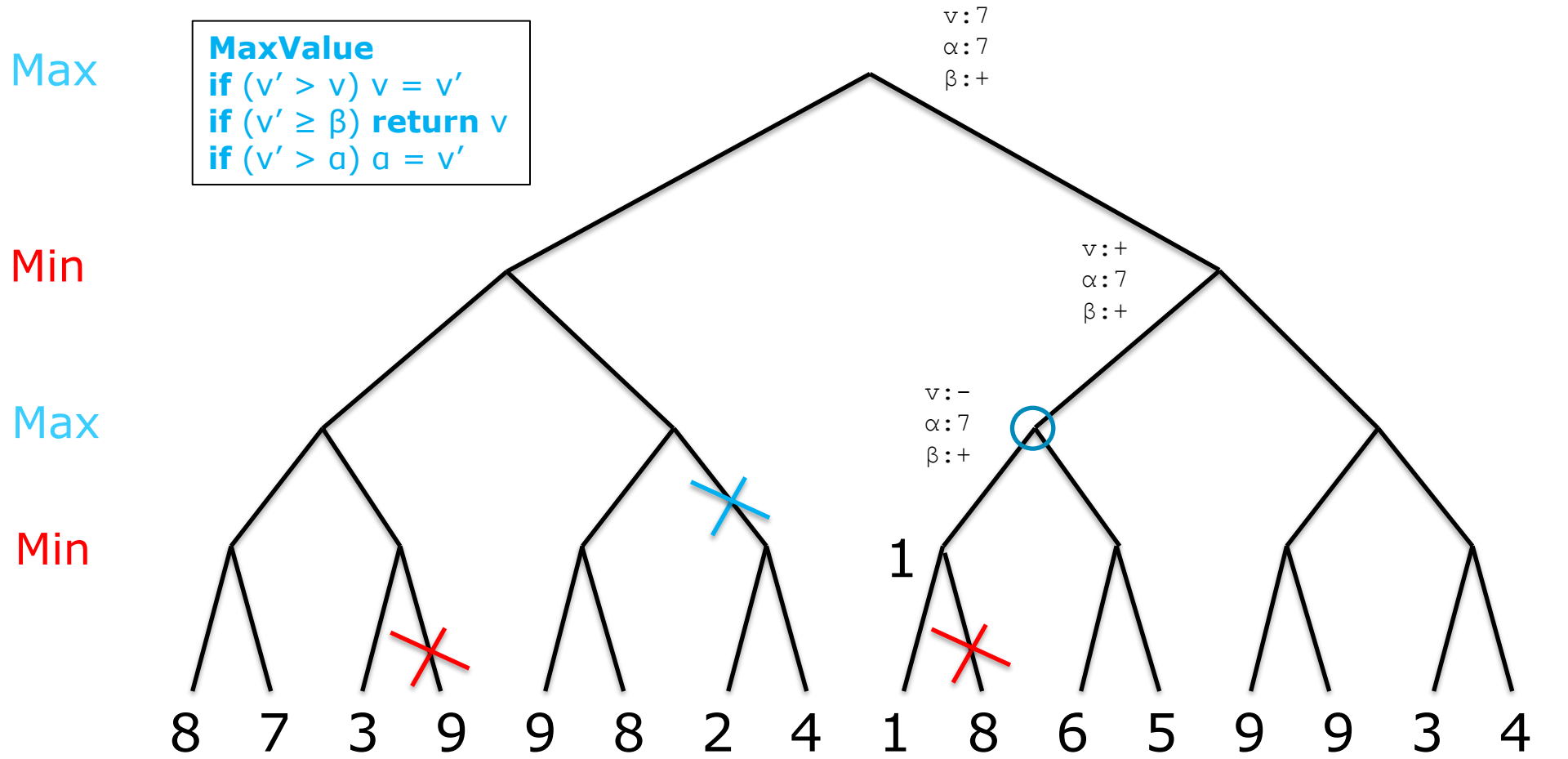

Max
Min
Max
Min
v : 7
α : 7
β : +
v : +
α : 7
β : +
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

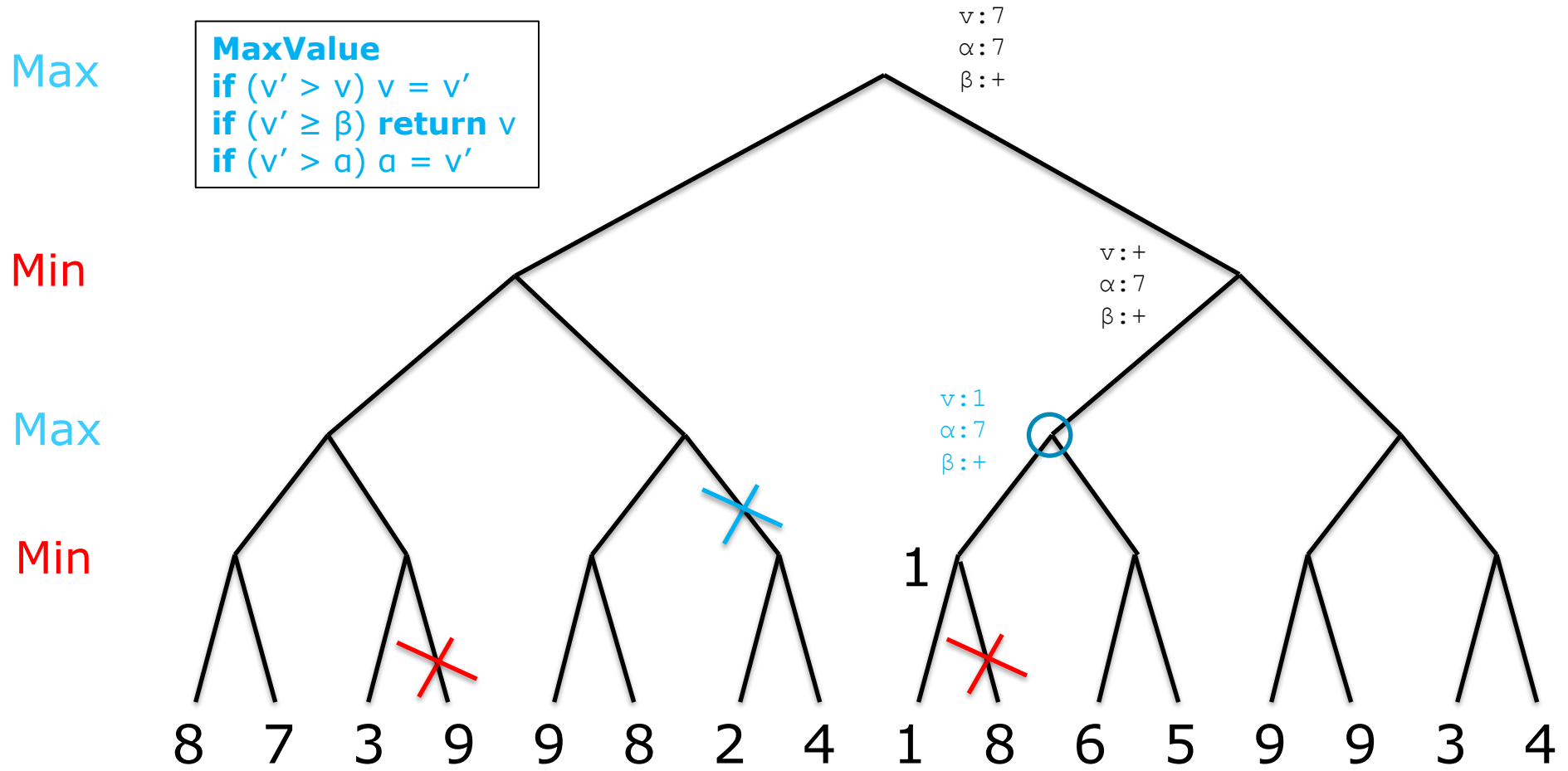

Max
Min
Max
Min
v : 7
α : 7
β : +
v : +
α : 7
β : +
v : −
α : 7
β : +
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

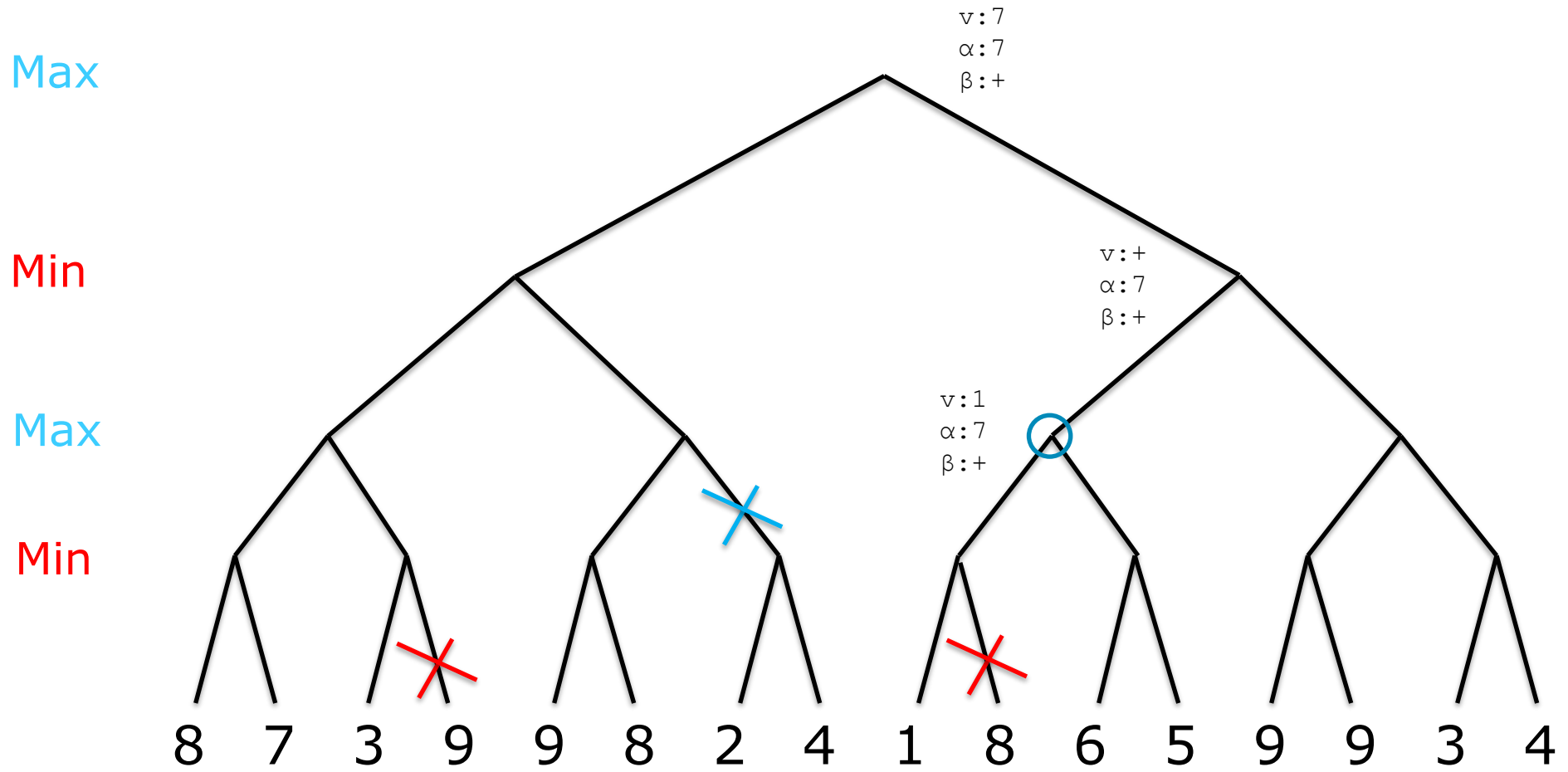

Max
Min
Max
Min
v : 7
α : 7
β : +
v : +
α : 7
β : +
v : −
α : 7
β : +
v : +
α : 7
β : +
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

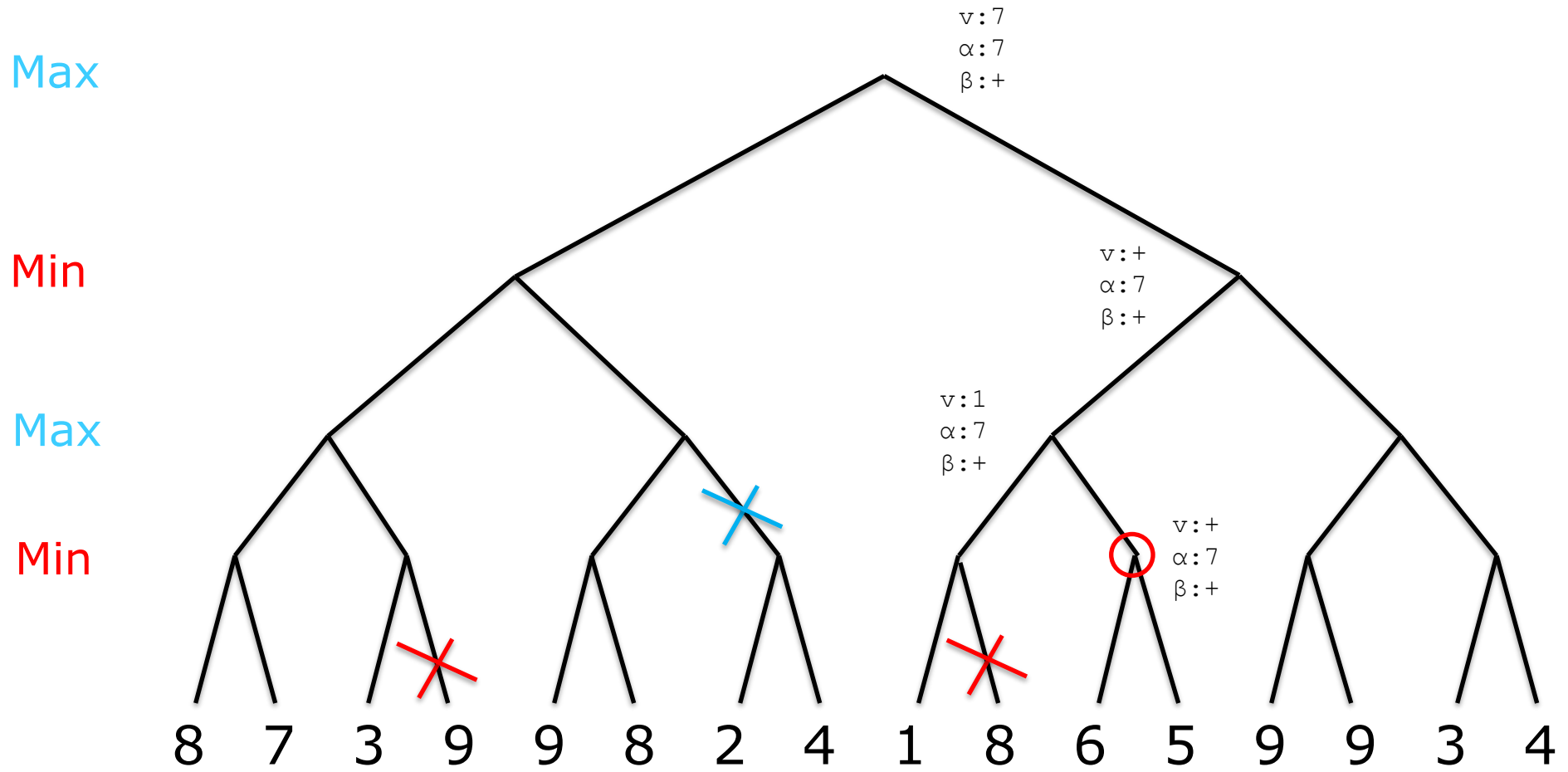

MinValue
if (v′ < v) v = v′
if (v′ ≤ α) return v
if (v′ < β) β = v′
Max
Min
Max
Min
v : 7
α : 7
β : +
v : +
α : 7
β : +
v : −
α : 7
β : +
v : +
α : 7
β : +
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

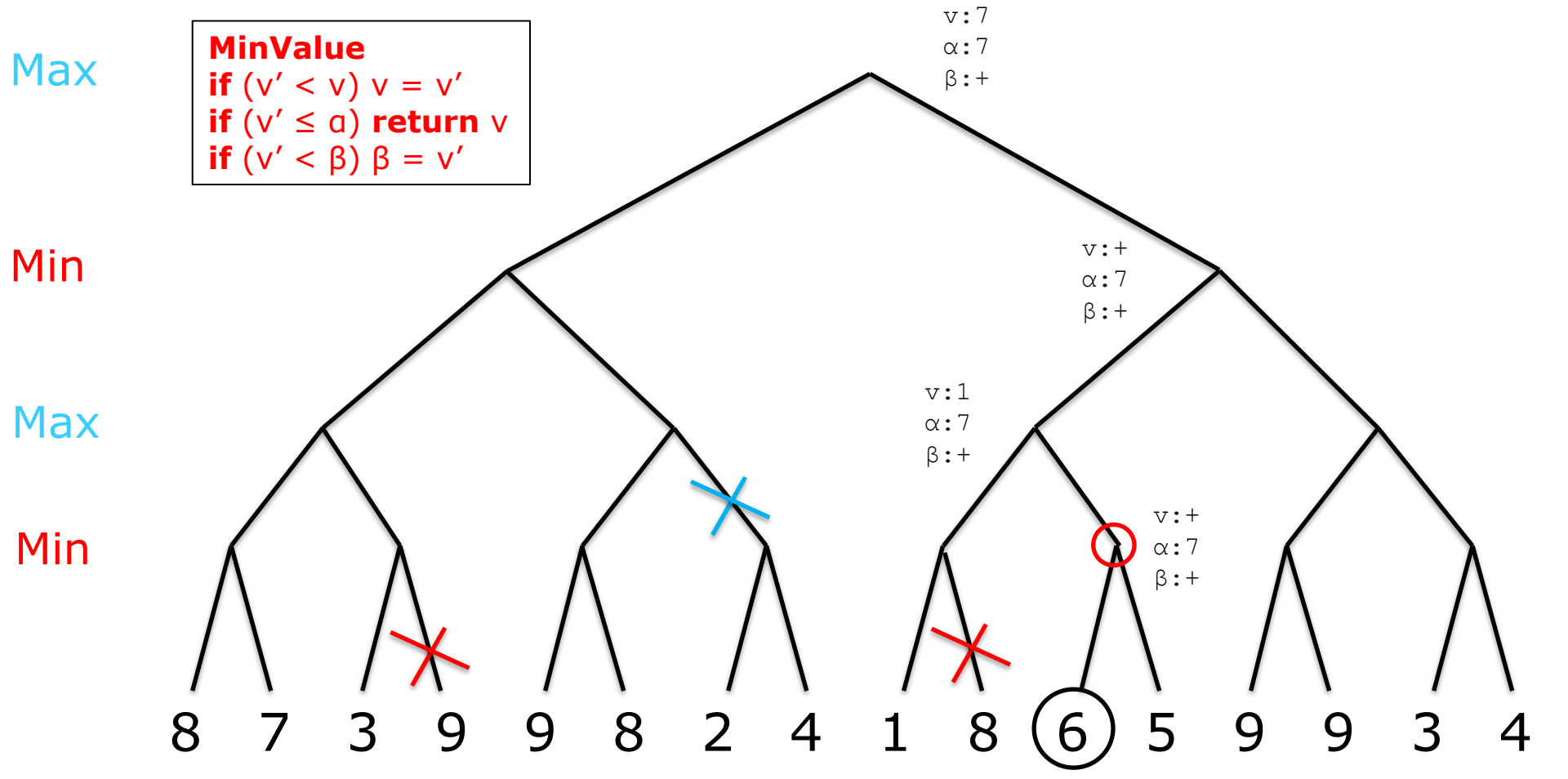

MinValue
if (v′ < v) v = v′
if (v′ ≤ α) return v
if (v′ < β) β = v′
Max
Min
Max
Min
v : 7
α : 7
β : +
v : +
α : 7
β : +
v : −
α : 7
β : +
v : 1
α : 7
β : +
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

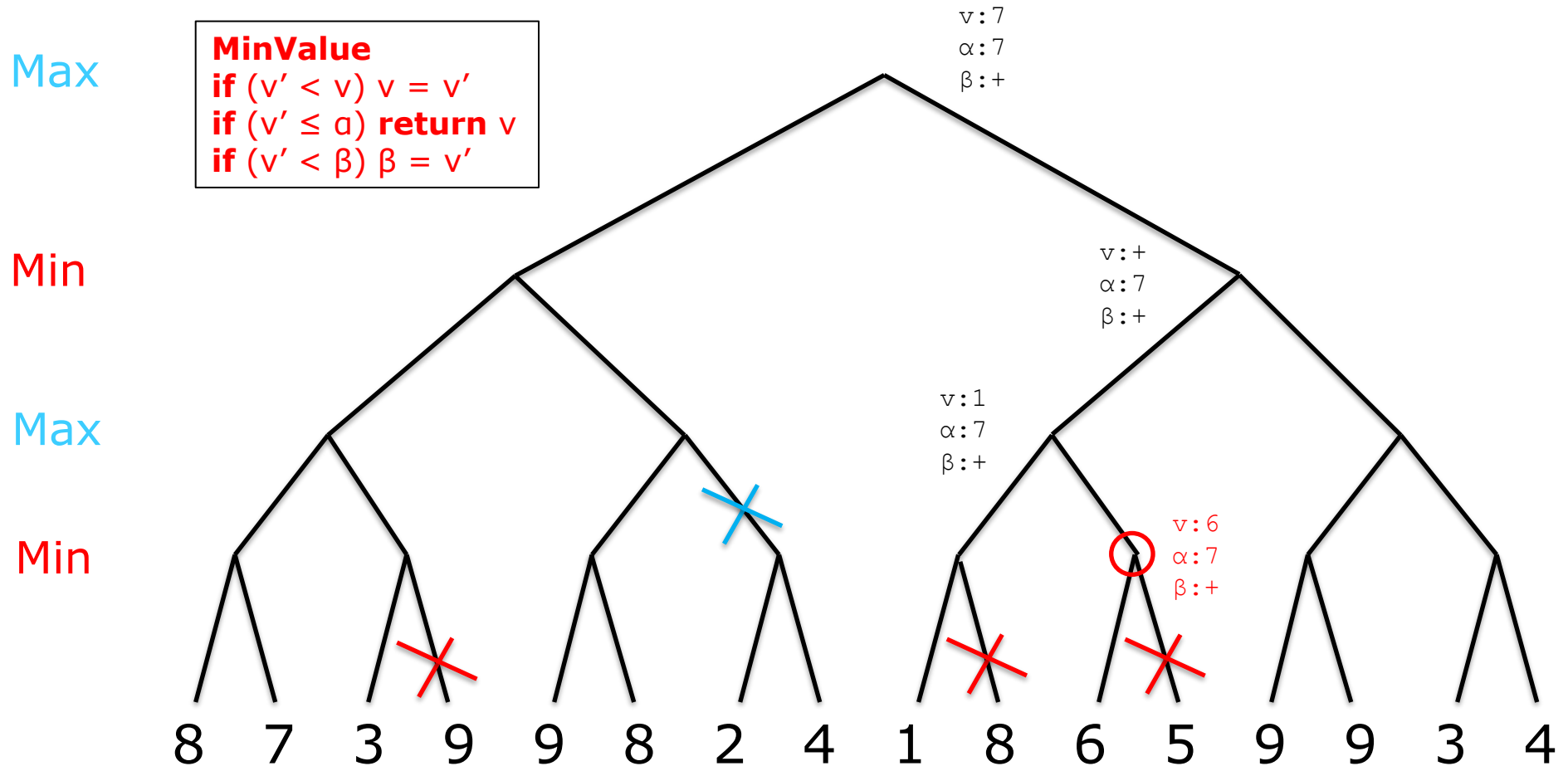

MaxValue
if (v′ > v) v = v′
if (v′ ≥ β) return v
if (v′ > α) α = v′
Max
Min
Max
Min
v : 7
α : 7
β : +
v : +
α : 7
β : +
v : −
α : 7
β : +
1
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

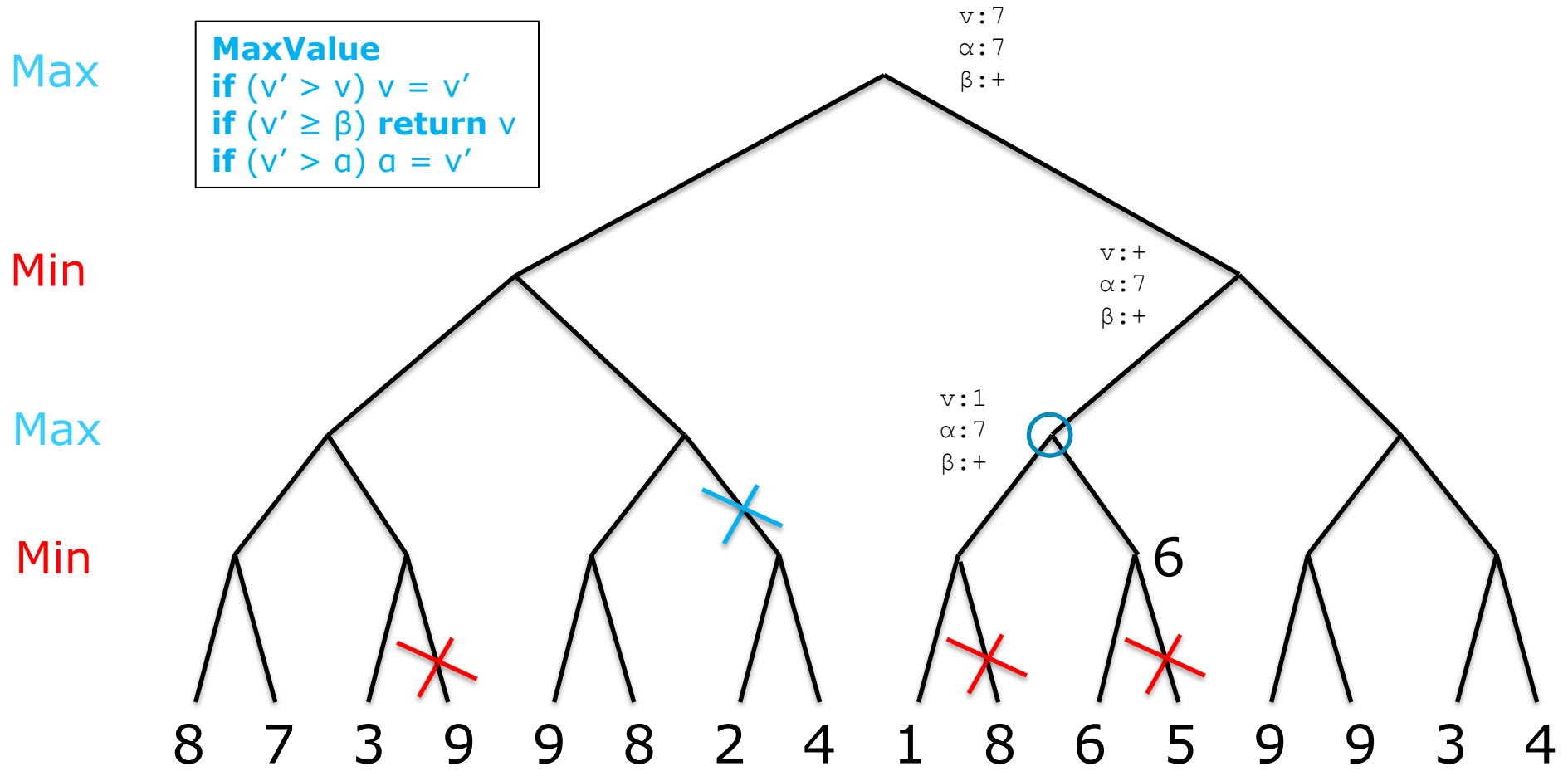

Max
Min
Max
Min
MaxValue
if (v′ > v) v = v′
if (v′ ≥ β) return v
if (v′ > α) α = v′
v : 7
α : 7
β : +
v : +
α : 7
β : +
v : 1
α : 7
β : +
1
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

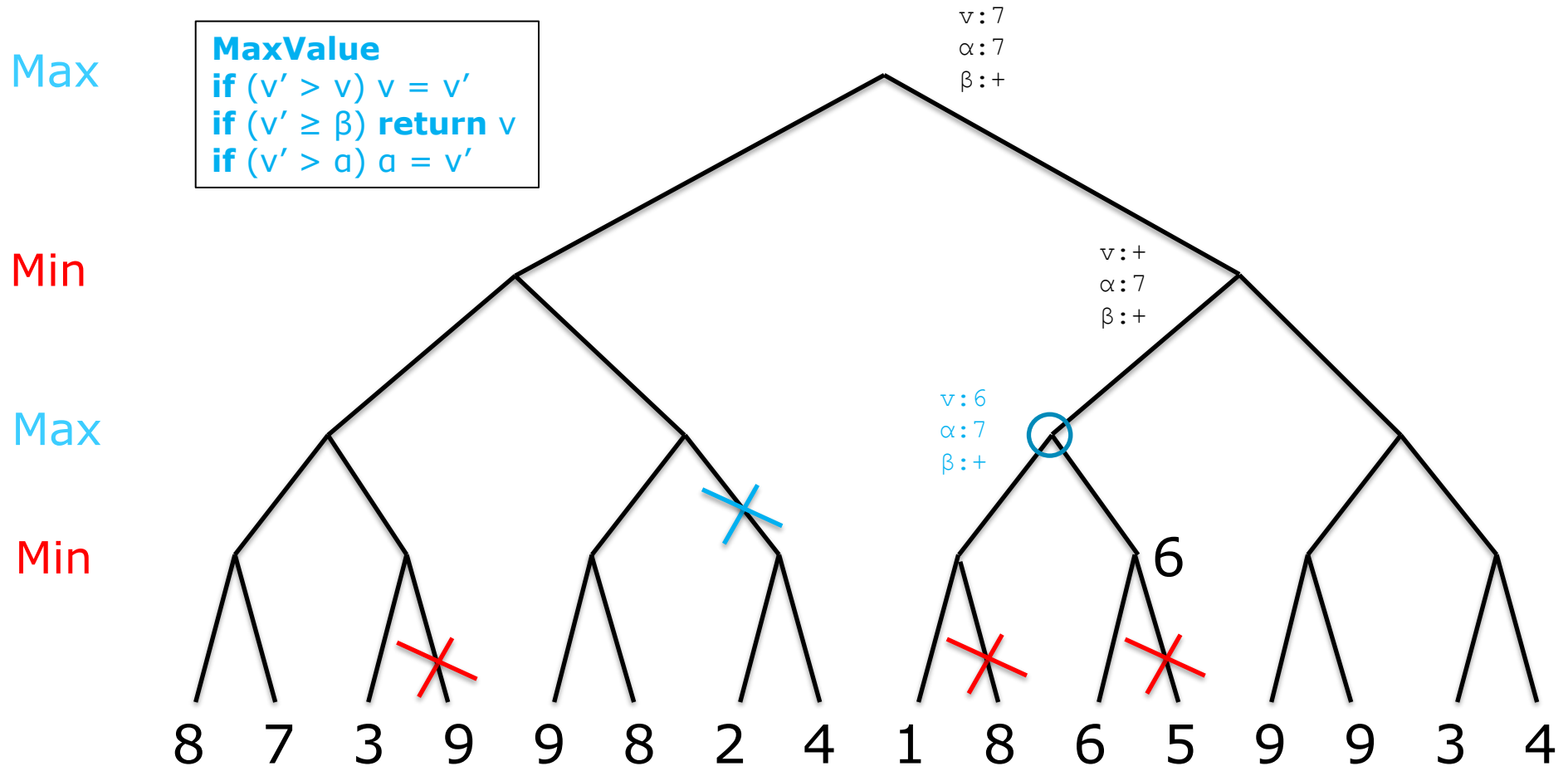

Max
Min
Max
Min
v:7
α:7
β:+
v:+
α:7
β:+
v:1
α:7
β:+
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

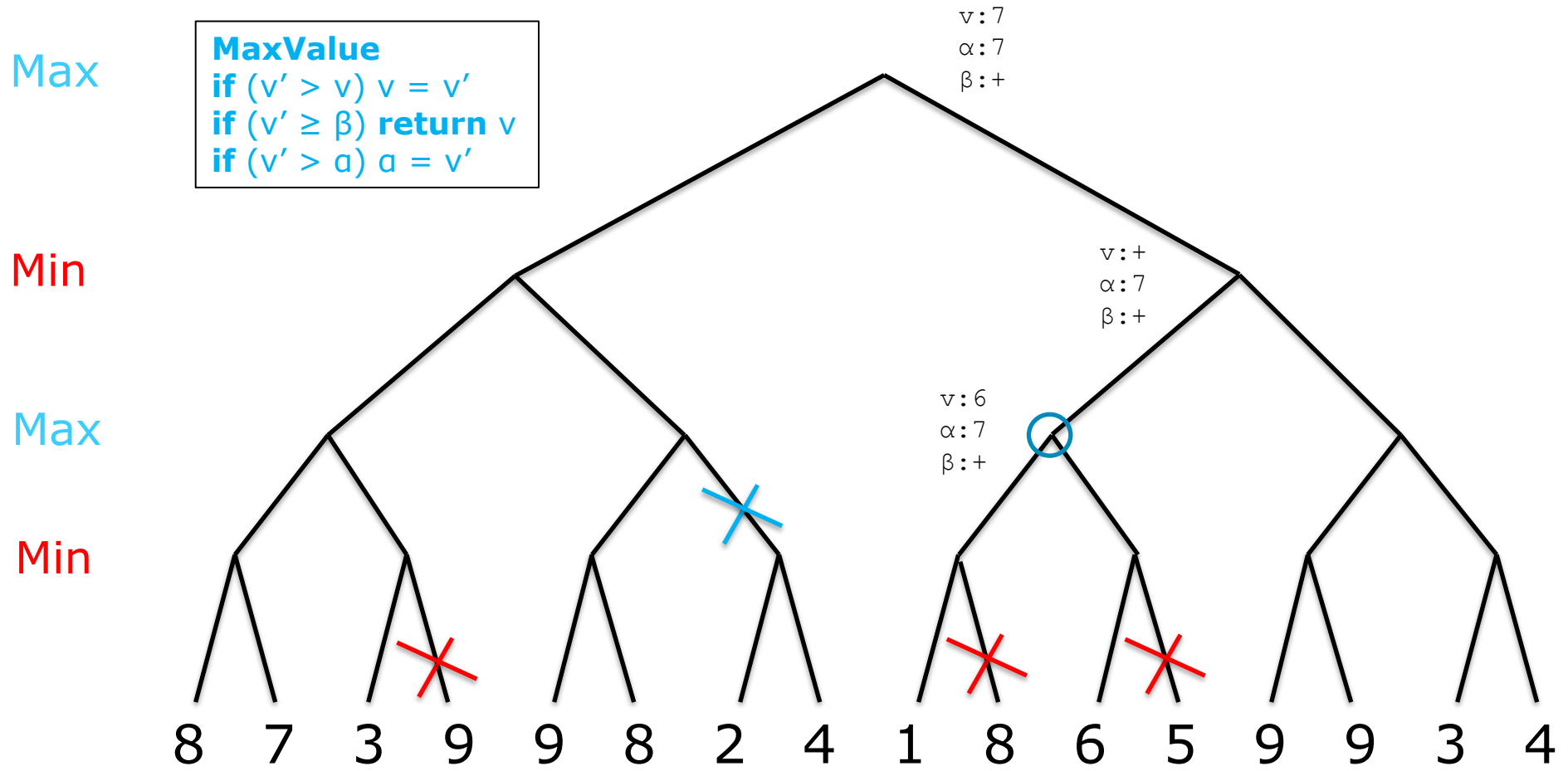

Max
Min
Max
Min
v : 7
α : 7
β : +
v : +
α : 7
β : +
v : 1
α : 7
β : +
v : +
α : 7
β : +
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

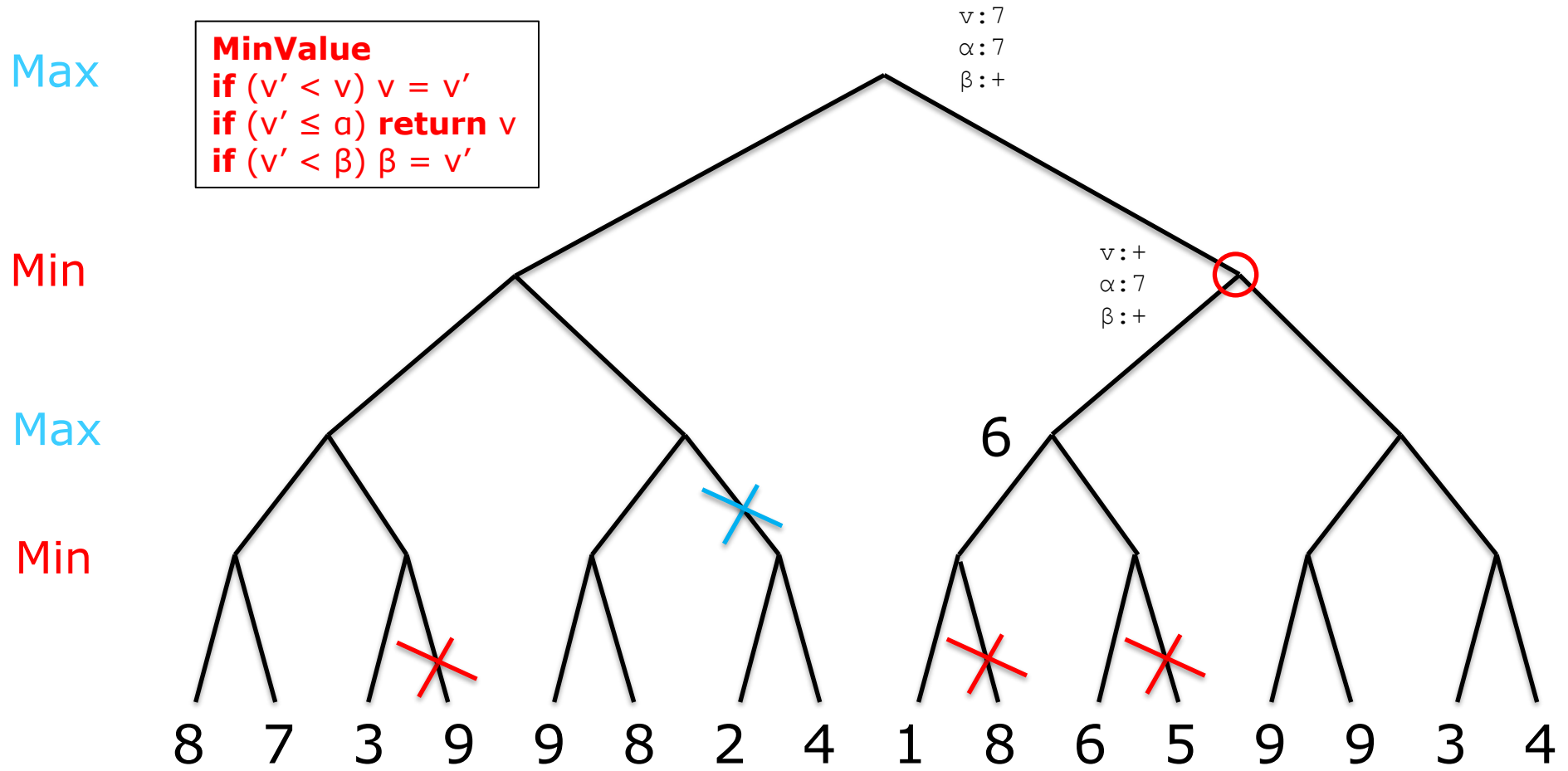

MinValue
if (v′ < v) v = v′
if (v′ ≤ α) return v
if (v′ < β) β = v′
Max
Min
Max
Min
v : 7
α : 7
β : +
v : +
α : 7
β : +
v : 1
α : 7
β : +
v : +
α : 7
β : +
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

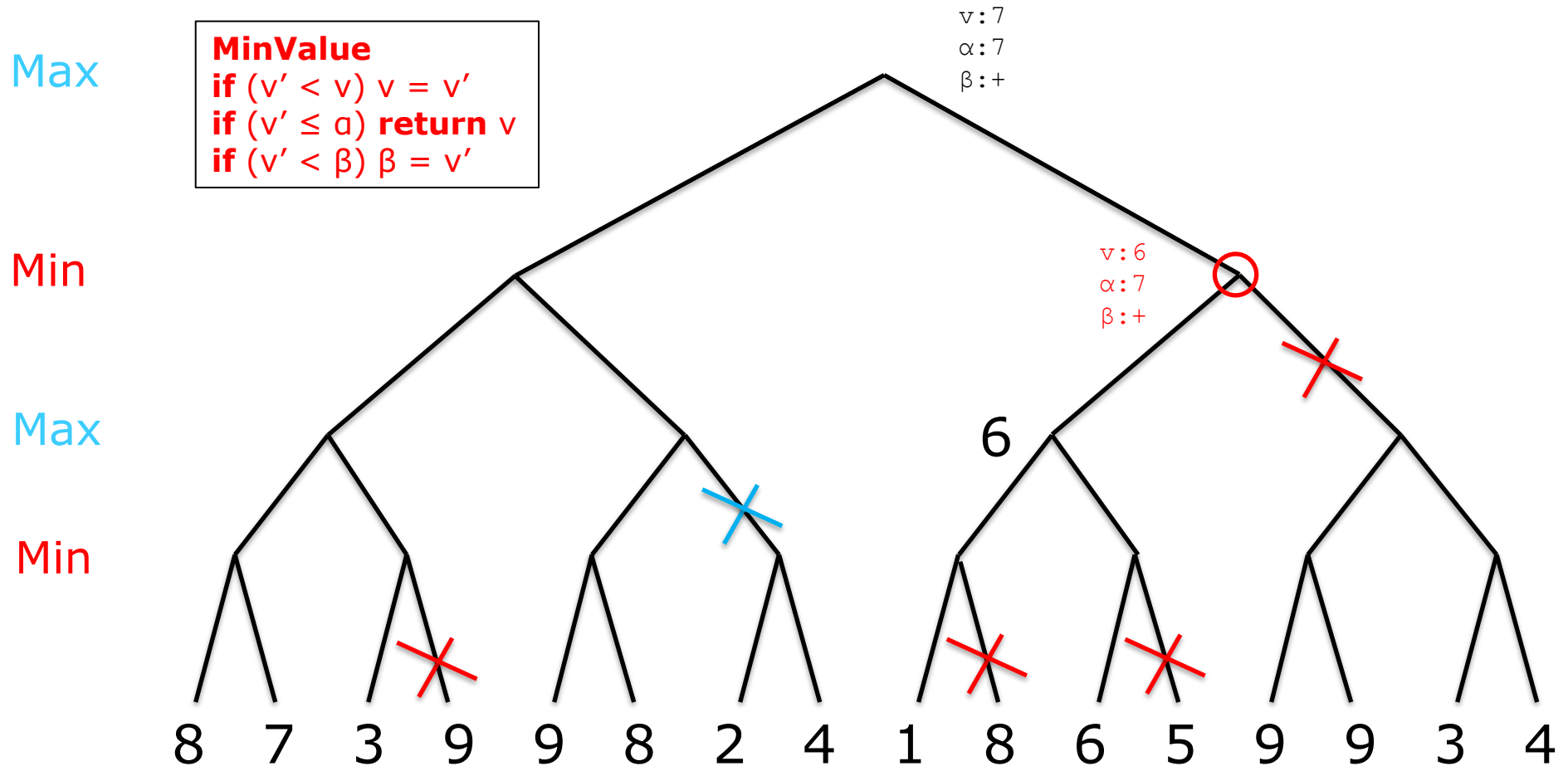

MinValue
if (v′ < v) v = v′
if (v′ ≤ α) return v
if (v′ < β) β = v′
Max
Min
Max
Min
v : 7
α : 7
β : +
v : +
α : 7
β : +
v : 1
α : 7
β : +
v : 6
α : 7
β : +
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

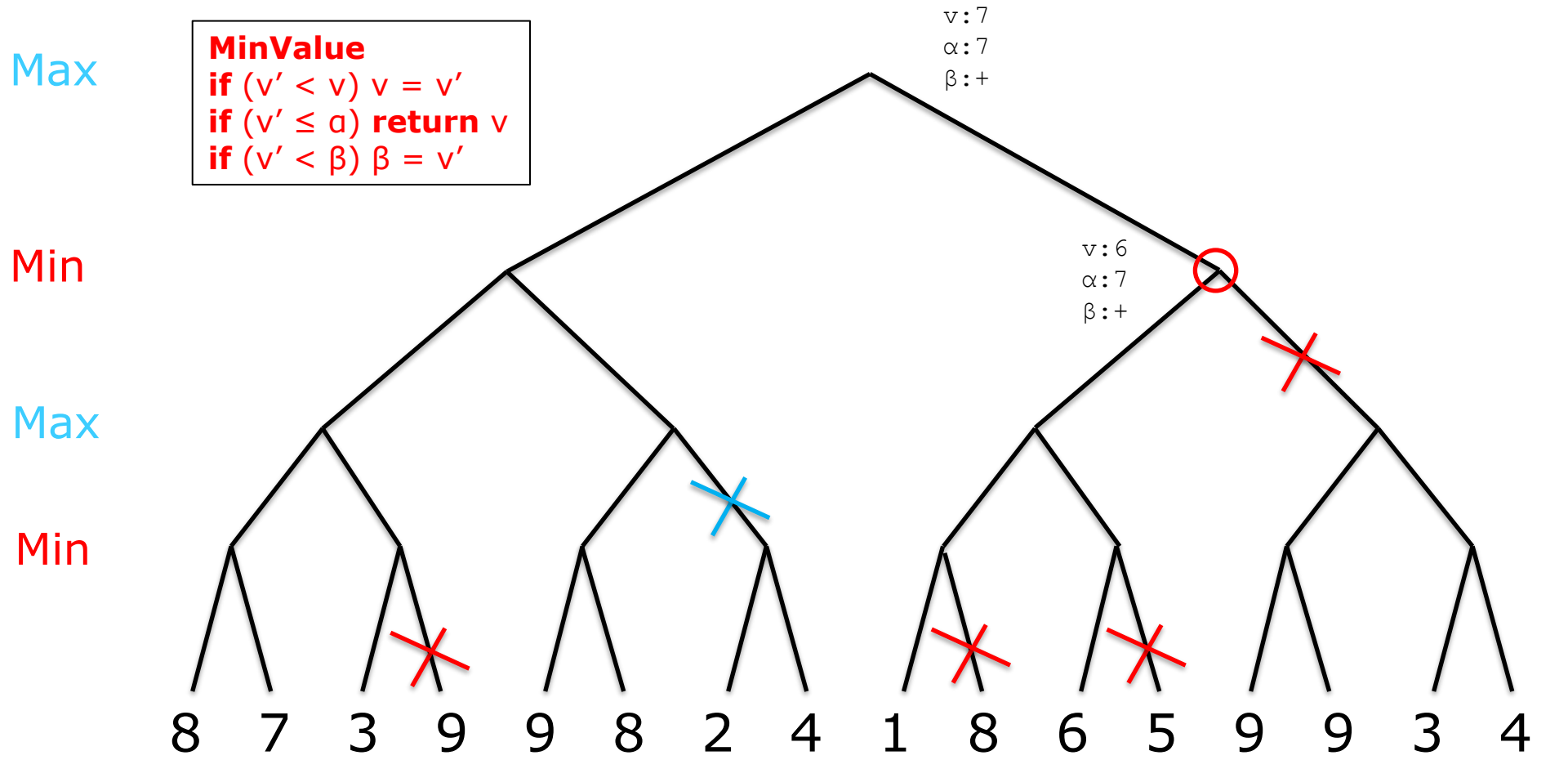

Max
Min
Max
Min
MaxValue
if (v′ > v) v = v′
if (v′ ≥ β) return v
if (v′ > α) α = v′
v : 7
α : 7
β : +
v : +
α : 7
β : +
v : 1
α : 7
β : +
6
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

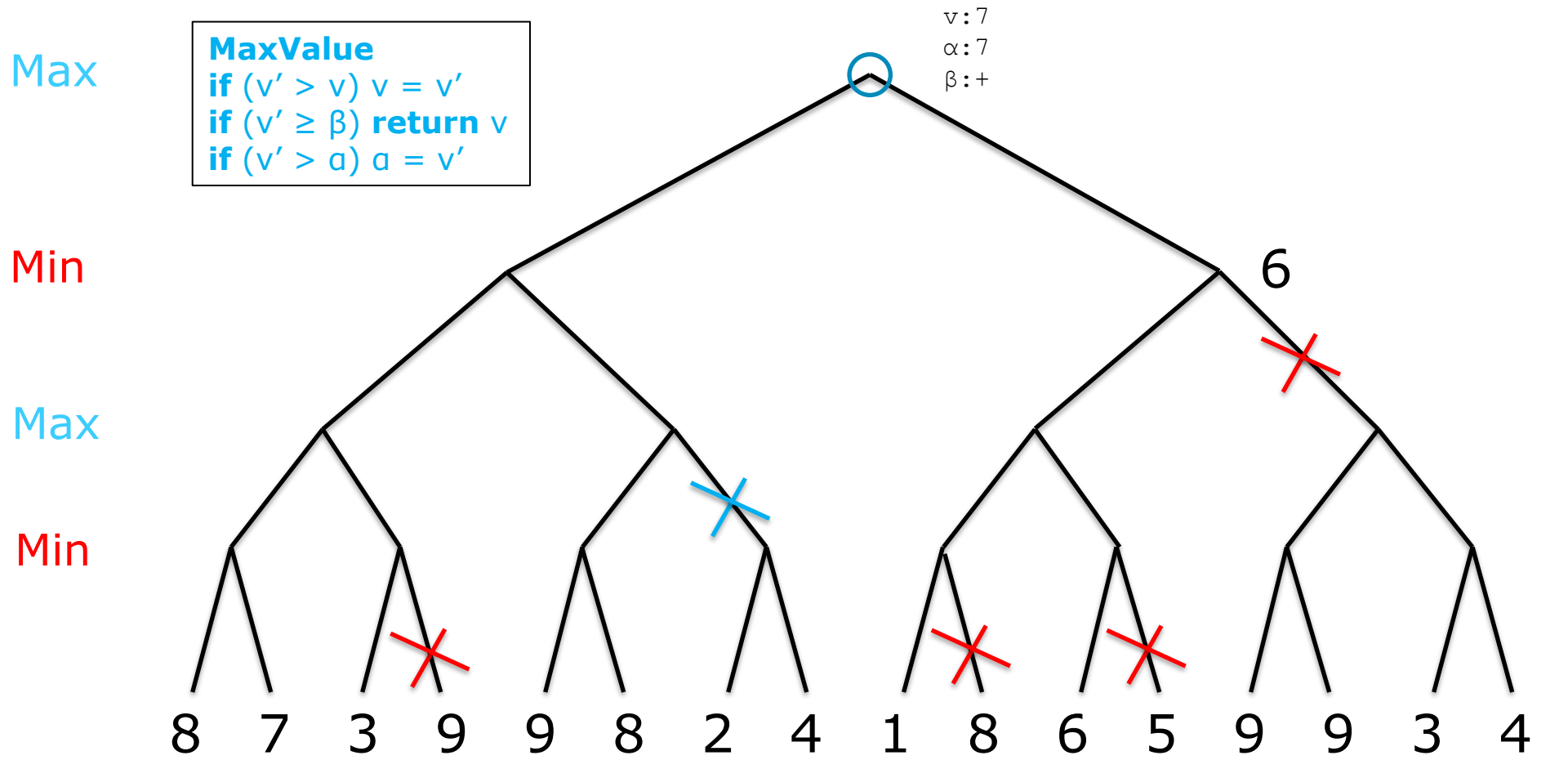

Max
Min
Max
Min
MaxValue
if (v′ > v) v = v′
if (v′ ≥ β) return v
if (v′ > α) α = v′
v : 7
α : 7
β : +
v : +
α : 7
β : +
v : 6
α : 7
β : +
6
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

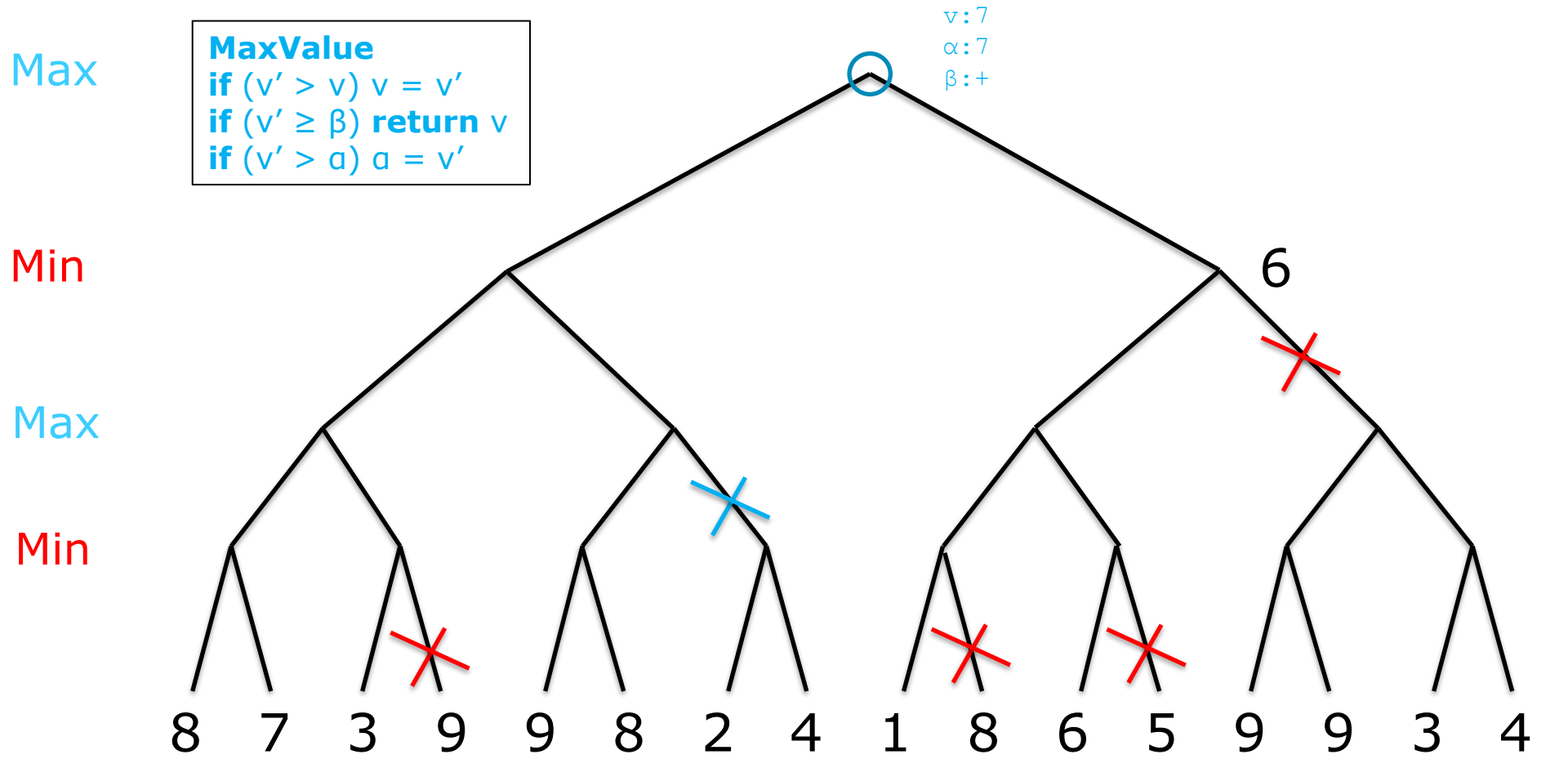

MaxValue
if (v′ > v) v = v′
if (v′ ≥ β) return v
if (v′ > α) α = v′
Max
Min
Max
Min
v : 7
α : 7
β : +
v : +
α : 7
β : +
v : 6
α : 7
β : +
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

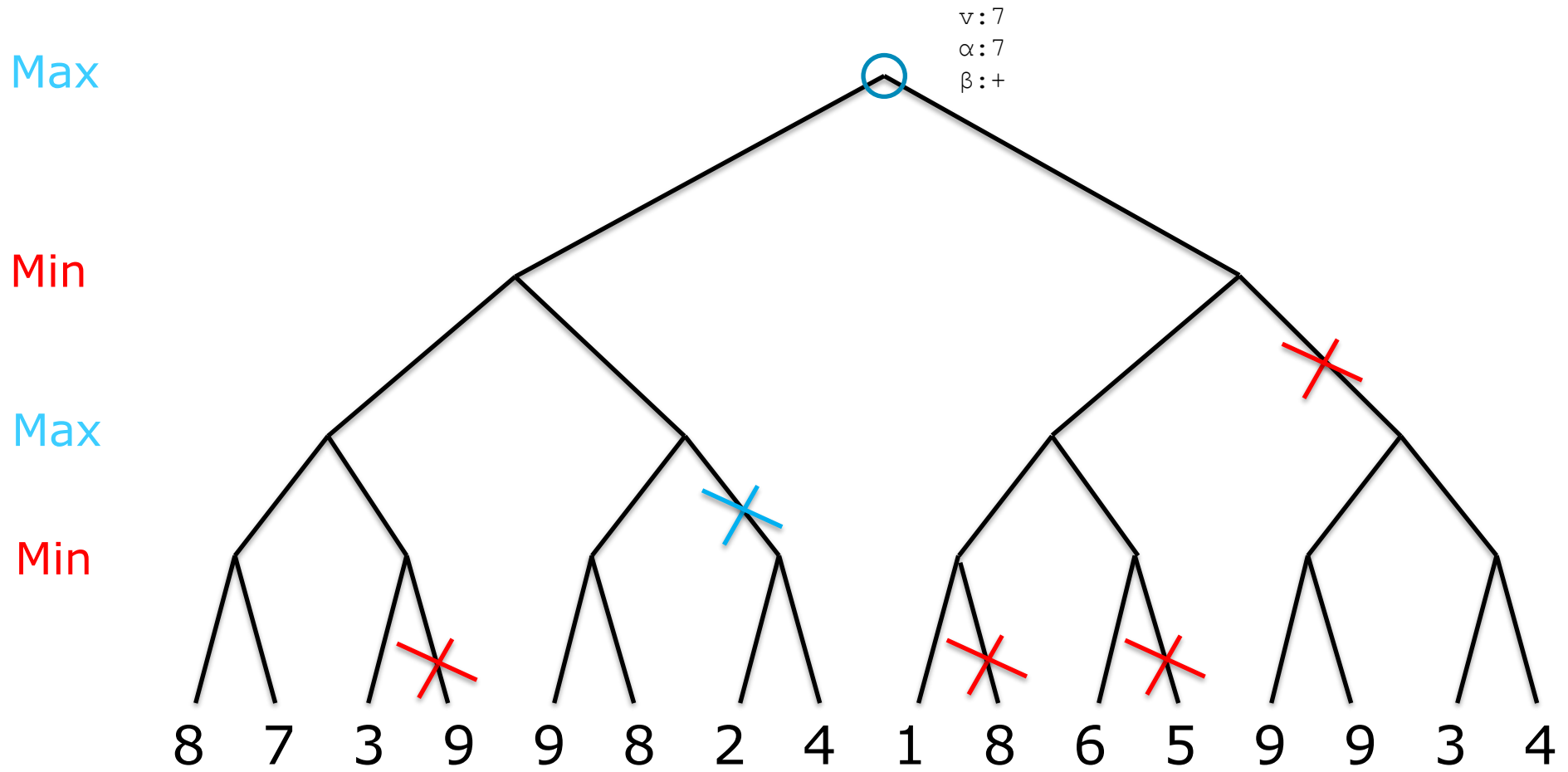

MinValue
if (v′ < v) v = v′
if (v′ ≤ α) return v
if (v′ < β) β = v′
Max
Min
Max
Min
v : 7
α : 7
β : +
v : +
α : 7
β : +
6
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

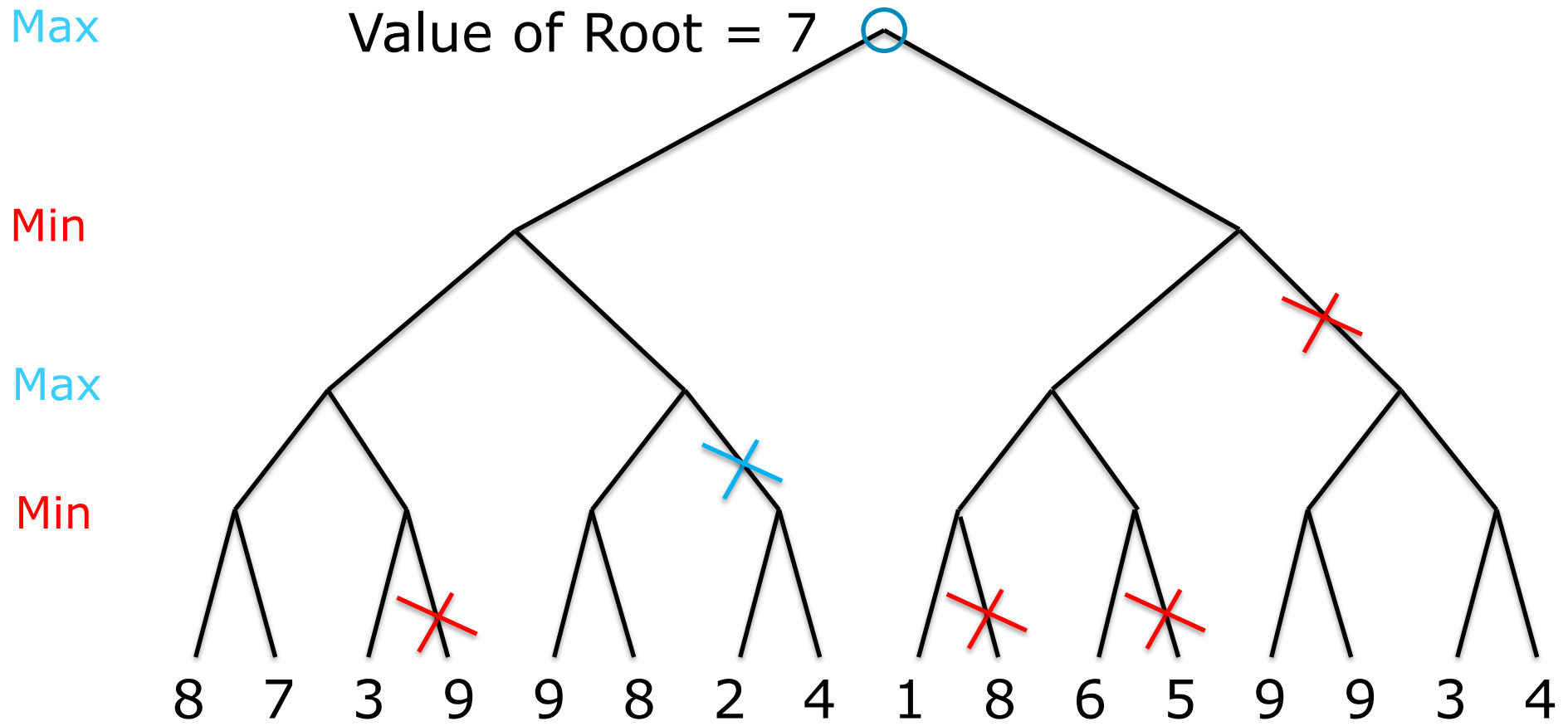

MinValue
if (v′ < v) v = v′
if (v′ ≤ α) return v
if (v′ < β) β = v′
Max
Min
Max
Min
v : 7
α : 7
β : +
v : 6
α : 7
β : +
6
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4


MinValue
if (v′ < v) v = v′
if (v′ ≤ α) return v
if (v′ < β) β = v′
Max
Min
Max
Min
v : 7
α : 7
β : +
v : 6
α : 7
β : +
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4


Max
MaxValue
if (v′ > v) v = v′
if (v′ ≥ β) return v
if (v′ > α) α = v′
v : 7
α : 7
β : +
Min
6
Max
Min
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4


MaxValue
if (v′ > v) v = v′
if (v′ ≥ β) return v
if (v′ > α) α = v′
v : 7
α : 7
β : +
Max
Min
Max
Min
6
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4


v : 7
α : 7
β : +
Max
Min
Max
Min
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4


Max
Value of Root = 7
Min
Max
Min
8 7 3 9 9 8 2 4 1 8 6 5 9 9 3 4

# Computational Savings

- MiniMax Tree Search
  - Nodes searched = $b^d$
- Alpha-Beta Search
  - Nodes searched ~ $2b^{d/2}$ (optimal savings)
- Depth 7 search becomes depth 14
  - Bad program beats world champion

# Recall MaxValue and MinValue

```
1.  Function MaxValue(s, α, β, d)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      v = -infinity
5.      for (c in children(s))
6.          v'= MinValue(c, α, β, d+1)
7.          if (v' > v) v = v'
8.          if (v' ≥ β) return v
9.          if (v' > α) α = v'
10.     return v
```

```
1.  Function MinValue(s, α, β, d)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      v = +infinity
5.      for (c in children(s))
6.          v'= MaxValue(c, α, β, d+1)
7.          if (v' < v) v = v'
8.          if (v' ≤ α) return v
9.          if (v' < β) β = v'
10.     return v
```

eval(s) returns score w.r.t. the maximizing player

# Alpha-Beta Pruning (minimax)

```
1.  Function AlphaBeta(s, α, β, d, max)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      if (max) // maximizing player
5.          v = -infinity
6.          for (c in children(s))
7.              v'= AlphaBeta(c, α, β, d+1, false)
8.              if (v' > v) v = v'
9.              if (v' ≥ β) return v
10.             if (v' > α) α = v'
11.         return v
12.     else  // minimizing player
13.         v = +infinity
14.         for (c in children(s))
15.             v'= AlphaBeta(c, α, β, d+1, true)
16.             if (v' < v) v = v'
17.             if (v' ≤ α) return v
18.             if (v' < β) β = v'
19.         return v
```

# Can we shorten it?

# MaxValue

```
1.  Function MaxValue(s, α, β, d)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      v = -infinity
5.      for (c in children(s))
6.          v'= MinValue(c, α, β, d+1)
7.          if (v' > v) v = v'
8.          if (v' ≥ β) return v
9.          if (v' > α) α = v'
10.     return v
```

# MaxValue

```
1.  Function MaxValue(s, α, β, d)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      v = -infinity
5.      for (c in children(s))
6.          v'= MinValue(c, α, β, d+1)
7.          if (v' > v) v = v'
8.          if (v' > α) α = v'
9.          if (v' ≥ β) return v
10.     return v
```

Re-order the return statement, has same net effect

# MaxValue

```
1.  Function MaxValue(s, α, β, d)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      v = -infinity
5.      for (c in children(s))
6.          v'= MinValue(c, α, β, d+1)
7.          if (v' > v) v = v'
8.          if (v' > α) α = v'
9.          if (α ≥ β) return v
10.     return v
```

Re-ordering allows us to compare α which will now store max(v')

# MaxValue

```
1.  Function MaxValue(s, α, β, d)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      v = -infinity
5.      for (c in children(s))
6.          v'= MinValue(c, α, β, d+1)
7.          if (v' > v) v = v'
8.          if (v' > α) α = v'
9.          if (α ≥ β) return α
10.     return v
```

We can also return α since it is storing max(v)

# MaxValue

```
1.  Function MaxValue(s, α, β, d)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      v = -infinity
5.      for (c in children(s))
6.          v'= MinValue(c, α, β, d+1)
7.          if (v' > v) v = v'
8.          if (v' > α) α = v'
9.          if (α ≥ β) return α
10.     return v
```

v is now only used as a max calculation placeholder

# MaxValue

```
1.  Function MaxValue(s, α, β, d)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.
5.      for (c in children(s))
6.          v'= MinValue(c, α, β, d+1)
7.
8.          if (v' > α) α = v'
9.          if (α ≥ β) return α
10.     return α
```

We can completely remove v and use α

# MaxValue

```
1.  Function MaxValue(s, α, β, d)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.
5.      for (c in children(s))
6.          α = max(α, MinValue(c, α, β, d+1))
7.
8.
9.          if (α ≥ β) return α
10.     return α
```

we can now get rid of the placeholder variable v' as well

# MaxValue

```
1.  Function MaxValue(s, α, β, d)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      for (c in children(s))
5.          α = max(α, MinValue(c, α, β, d+1))
6.          if (α ≥ β) return α
7.      return α
```

This is the most compact version of MaxValue

# MaxValue

```
1.  Function MaxValue(s, α, β, d)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      for (c in children(s))
5.          α = max(α, MinValue(c, α, β, d+1))
6.          if (α ≥ β) break
7.      return α
```

We can use break instead of return, since we return outside the loop

# MaxValue and MinValue

```
1. Function MaxValue(s, α, β, d)
2.     if (terminal(s) or d > maxD)
3.         return eval(s)
4.     for (c in children(s))
5.         α = max(α, MinValue(c, α, β, d+1))
6.         if (α ≥ β) break
7.     return α
```

```
1. Function MinValue(s, α, β, d)
2.     if (terminal(s) or d > maxD)
3.         return eval(s)
4.     for (c in children(s))
5.         β = min(β, MaxValue(c, α, β, d+1))
6.         if (β ≤ α) break
7.     return β
```

Same shortening can be applied in MinValue algorithm

# MaxValue and MinValue

```
1. Function MaxValue(s, α, β, d)
2.     if (terminal(s) or d > maxD)
3.         return eval(s)
4.     for (c in children(s))
5.         α = max(α, MinValue(c, α, β, d+1))
6.         if (α ≥ β) break
7.     return α
```

```
1. Function MinValue(s, α, β, d)
2.     if (terminal(s) or d > maxD)
3.         return eval(s)
4.     for (c in children(s))
5.         β = min(β, MaxValue(c, α, β, d+1))
6.         if (α ≥ β) break
7.     return β
```

Reverse comparison in MinValue to get the same one as MaxValue

# Shortened Max/Min

```
1.  Function AlphaBeta(s, α, β, d, max)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      if (max) // maximizing player
5.          for (c in children(s))
6.              α = max(α, AlphaBeta(c, α, β, d+1, !max))
7.              if (α ≥ β) break
8.          return α
9.      else                  // minimizing player
10.         for (c in children(s))
11.             β = min(β, AlphaBeta(c, α, β, d+1, !max))
12.             if (α ≥ β) break
13.         return β
```

# Notice Similarities

```
1.  Function AlphaBeta(s, α, β, d, max)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      if (max) // maximizing player
5.          for (c in children(s))
6.              α = max(α, AlphaBeta(c, α, β, d+1, !max))
7.              if (α ≥ β) break
8.          return α
9.      else                // minimizing player
10.         for (c in children(s))
11.             β = min(β, AlphaBeta(c, α, β, d+1, !max))
12.             if (α ≥ β) break
13.         return β
```

There is duplicated code required due to if(max) being on the outside of the child loop

# Bring Max Condition Inside the Loop

```
1. Function AlphaBeta(s, α, β, d, max)
2.     if (terminal(s) or d > maxD)
3.         return eval(s)
4.     for (c in children(s))
5.         if (max)  α = max(α, AlphaBeta(c, α, β, d+1, !max))
6.         if (!max) β = min(β, AlphaBeta(c, α, β, d+1, !max))
7.         if (α ≥ β) break
8.     return max ? α : β
```

Final shortened version of Alpha-Beta

# Alpha-Beta (short version)

```
1. Function AlphaBeta(s, α, β, d, max)
2.     if (terminal(s) or d > maxD)
3.         return eval(s)
4.     for (c in children(s))
5.         if (max)  α = max(α, AlphaBeta(c, α, β, d+1, !max))
6.         if (!max) β = min(β, AlphaBeta(c, α, β, d+1, !max))
7.         if (α ≥ β) break
8.     return max ? α : β
```

Final shortened version of AB, returns VALUE of state

Result is exactly the same as the long version

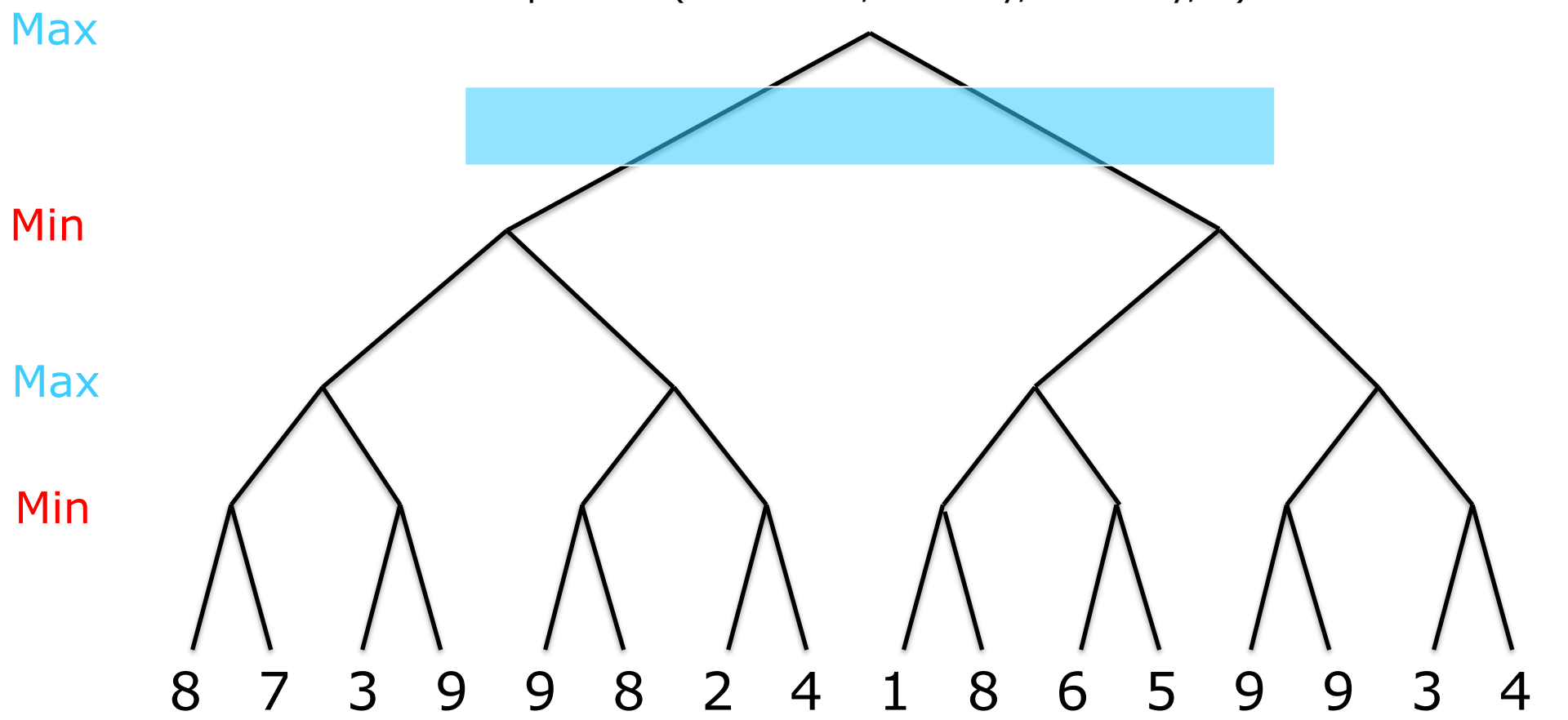
Implement whichever one you feel more comfortable with

# Which action do we take?

```
1.  Function AlphaBeta(s, α, β, d, max)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      for (c in children(s))
5.          if (max)  α = max(α, AlphaBeta(c, α, β, d+1, !max))
6.          if (!max) β = min(β, AlphaBeta(c, α, β, d+1, !max))
7.          if (α ≥ β) break
8.      return max ? α : β
```

**So far only calculating value, need to record the action**

AlphaBeta(startState, -infinity, +infinity, 0)



# Record Best Action

```
1.  Function AlphaBeta(s, α, β, d, max)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      for (c in children(s))
5.          v' = AlphaBeta(c, α, β, d+1, !max)
6.          if (max)  α = max(α, v')
7.          if (!max) β = min(β, v')
8.          if (α ≥ β) break
9.      return max ? α : β
```

Re-introduce temporary value variable v' and is it for comparison

# Record Best Action

```
1.  Function AlphaBeta(s, α, β, d, max)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      for (c in children(s))
5.          v' = AlphaBeta(c, α, β, d+1, !max)
6.          if (max and (v' > α))  α = v'
7.          if (!max and (v' < β)) β = v'
8.          if (α ≥ β) break
9.      return max ? α : β
```

Re-introduce temporary value variable v' and is it for comparison

# Record Best Action

```
1.  Function AlphaBeta(s, α, β, d, max)
2.      if (terminal(s) or d > maxD)
3.          return eval(s)
4.      for (c in children(s))
5.          v' = AlphaBeta(c, α, β, d+1, !max)
6.          if (max and (v' > α))
7.              a = v'
8.              if (d == 0) bestAction = c.action
9.          if (!max and (v' < β)) β = v'
10.         if (α ≥ β) break
11.     return max ? α : β
```

We can use a global/state variable to store best action at the root node (d=0)

# Record Action AB (Long Version)

```
1.  Function AlphaBeta(s, α, β, d, max)
2.      if (terminal(s) or d > self.currentMaxDepth)
3.          return eval(s)
4.      if (max) // maximizing player
5.          v = -infinity
6.          for (c in children(s))
7.              v'= AlphaBeta(c, α, β, d+1, !max)
8.              if (v' > v) v = v'
9.              if (v' ≥ β) return v
10.             if (v' > α)
11.                 α = v'
12.                 if (d == 0) self.currentBestAction = c.action
13.         return v
14.     else    // minimizing player
15.         v = +infinity
16.         for (c in children(s))
17.             v'= AlphaBeta(c, α, β, d+1, !max)
18.             if (v' < v) v = v'
19.             if (v' ≤ α) return v
20.             if (v' < β) β = v'
21.         return v
```

# Time Limit

- What if we want a time limit?
- Search to a given depth time varies
- Incorporate iterative deepening
- “Iterative Deepening Alpha Beta”

# Iterative Deepening AlphaBeta

- For d=1, 2, …, infinity
- Perform AB search to depth d
- When time runs out, return the best action from the last fully completed depth
- Can’t trust a search that didn’t complete
- How to escape AlphaBeta recursion?
  - Throw exception, catch it outside

# Assignment 3 - IDAB

```
1.  class Player_Student
2.  {
3.      this.config = { timeLimit, maxDepth };
4.      this.currentMaxDepth   = 1;
5.      this.bestAction        = null;
6.      this.currentBestAction = null;
7.      this.maxPlayer         = null;
8.      eval         (state, player);
9.      IDAlphaBeta(state);
10.     AlphaBeta  (state, alpha, beta, depth, max);
11. }
```

# Assignment 3 - IDAB

```
1.  Function IDAlphaBeta(s)
2.      this.bestAction = null;
3.      this.maxPlayer = s.player;
4.      for (d=1; d < this.config.maxDepth; d++)
5.          this.currentMaxDepth = d;
6.          try
7.              AlphaBeta(s, -infinity, infinity, 0, true);
8.              // update bestAction only if AB completes new depth
9.              this.bestAction = this.currentBestAction;
10.         catch (TimeOutException)
11.             break;
12.     return this.bestAction
```

# Assignment 3 - AlphaBeta

```
1.    Function AlphaBeta(s, α, β, d, max)
2.        if (terminal(s) or d >= this.currentMaxDepth) return eval(s, this.maxPlayer)
3.        if (timeElapsed > this.config.timeLimit) throw TimeOutException;
4.        if (max) // maximizing player
5.            v = -infinity
6.            for (c in children(s))
7.                v'= AlphaBeta(c, α, β, d+1, !max)
8.                if (v' > v) v = v'
9.                if (v' ≥ β) return v
10.               if (v' > α)
11.                   α = v'
12.                   if (d == 0) this.currentBestAction = c.action
13.           return v
14.       else // minimizing player
15.           v = +infinity
16.           for (c in children(s))
17.               v'= AlphaBeta(c, α, β, d+1, !max)
18.               if (v' < v) v = v'
19.               if (v' ≤ α) return v
20.               if (v' < β) β = v'
21.           return v
```

# Assignment 3 - MiniMax

```
1.  Function MiniMax(s, d, max)
2.      if (terminal(s) or d >= this.currentMaxDepth) return eval(s, this.maxPlayer)
3.      if (timeElapsed > this.config.timeLimit) throw TimeOutException;
4.      if (max)
5.          v = -infinity
6.          for (c in children(s))
7.              v'= MiniMax(c, d+1, false)
8.              if (v' > v)
9.                  v = v'
10.                 if (d == 0) this.currentBestAction = c.action
11.         return v
12.     else
13.         v = +infinity
14.         for (c in children(s))
15.             v'= MiniMax(c, d+1, true)
16.             if (v' < v) v = v'
17.         return v
```

# Assignment 3 – AlphaBeta (short)

```
1.  Function AlphaBeta(s, α, β, d, max)
2.      if (terminal(s) or d >= this.currentMaxDepth)
3.          return eval(s, this.maxPlayer)
4.      if (timeElapsed > this.config.timeLimit) throw TimeOutException;
5.      for (c in children(s))
6.          v' = AlphaBeta(c, α, β, d+1, !max)
7.          if (max and (v' > α))
8.              α = v'
9.              if (d == 0) this.currentBestAction = c.action
10.         if (!max and (v' < β)) β = v'
11.         if (α ≥ β) break
12.     return max ? α : β
```

# Exam Questions

- Know relationship between MiniMax search and Nash Equilibrium
- Know MiniMax / AlphaBeta algorithms
- Why Iterative Deepening is necessary
- How AlphaBeta pruning helps MiniMax
- Given a tree with leaf values, compute the MiniMax or AlphaBeta value returned