Mini-Max vs Alpha-Beta

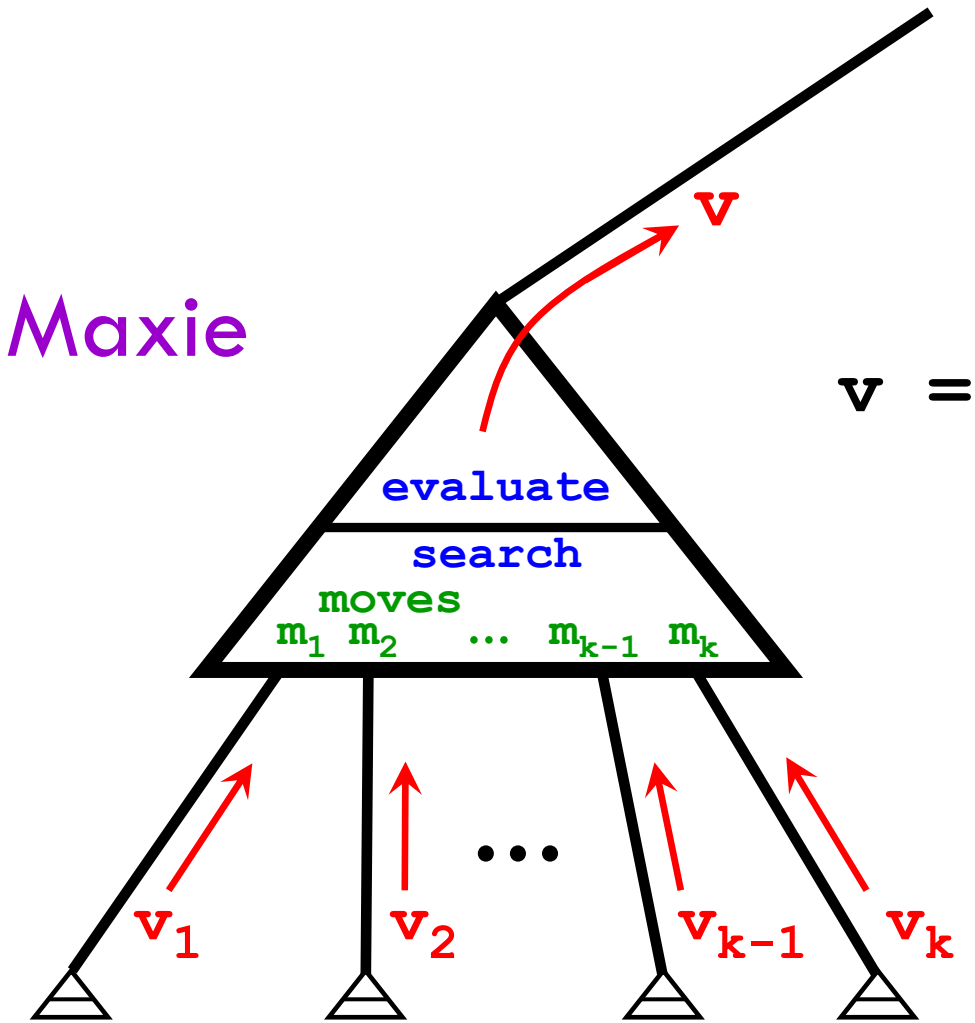
15-150

Principles of Functional Programming

April 14, 2020

Michael Erdmann

Mini-Max at a Maxie Node



$v = \max\{v_1, \ldots, v_k\}$

alpha-beta pruning

- Rather than blindly evaluate children of a node, we pass an accumulator argument:

$$(\alpha\ ,\ \beta)$$ (REQUIRE $\alpha < \beta$)

- α is the best Maxie can achieve in the tree explored so far.
- β is the best Minnie can achieve in the tree explored so far.

Think of (α , β) as an *open* interval of potential game values that *both* players would like better than their current best options.

alpha-beta pruning

- Rather than blindly evaluate children of a node, we pass an accumulator argument:

$$(\alpha\ ,\ \beta) \qquad \text{(REQUIRE } \alpha < \beta\text{)}$$

- α is the best Maxie can achieve in the tree explored so far.
- β is the best Minnie can achieve in the tree explored so far.

KEY POINT: If Maxie sees a move to a node with value $v \geq \beta$, then Maxie should stop searching from the current node.

Q: Why?

A: Minnie can prevent the game from reaching the current node.

(Dually, with roles of Maxie and Minnie reversed, now using "value $v \leq \alpha$".)

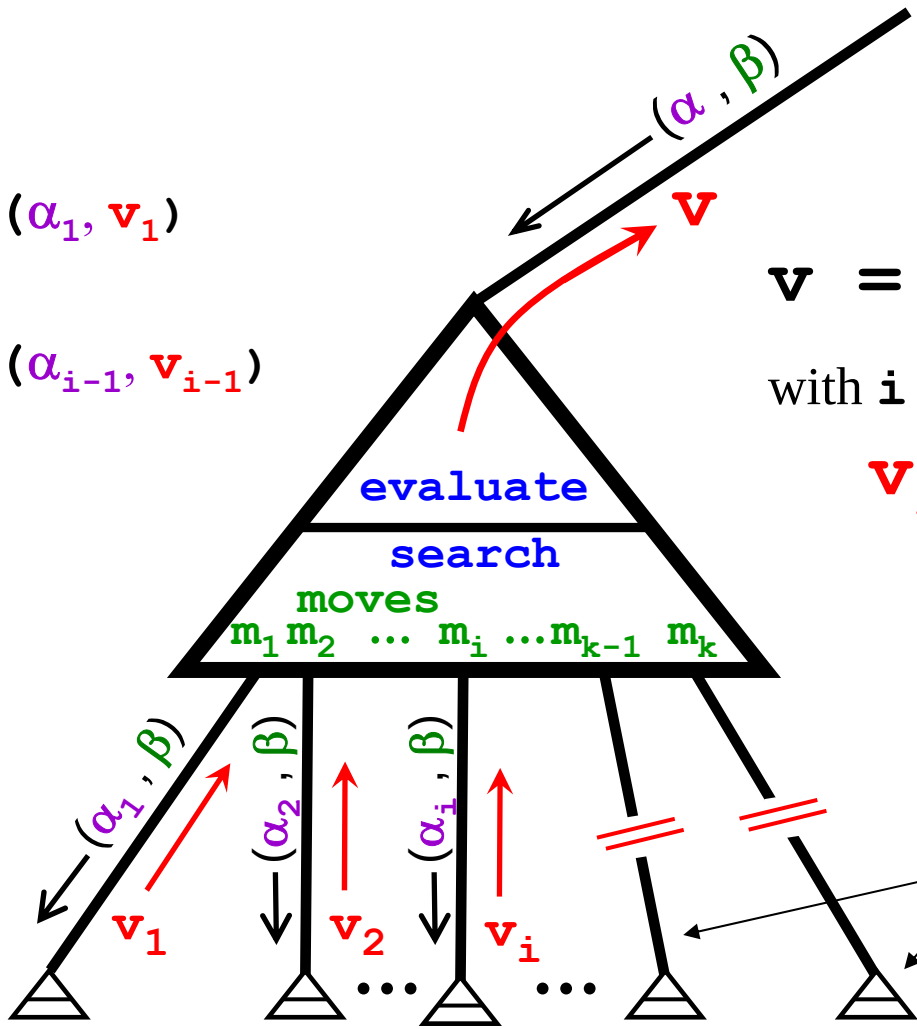
Alpha-Beta at a Maxie Node

$\alpha_1 = \alpha$

$\alpha_2 = \max(\alpha_1, v_1)$

$\vdots$

$\alpha_i = \max(\alpha_{i-1}, v_{i-1})$



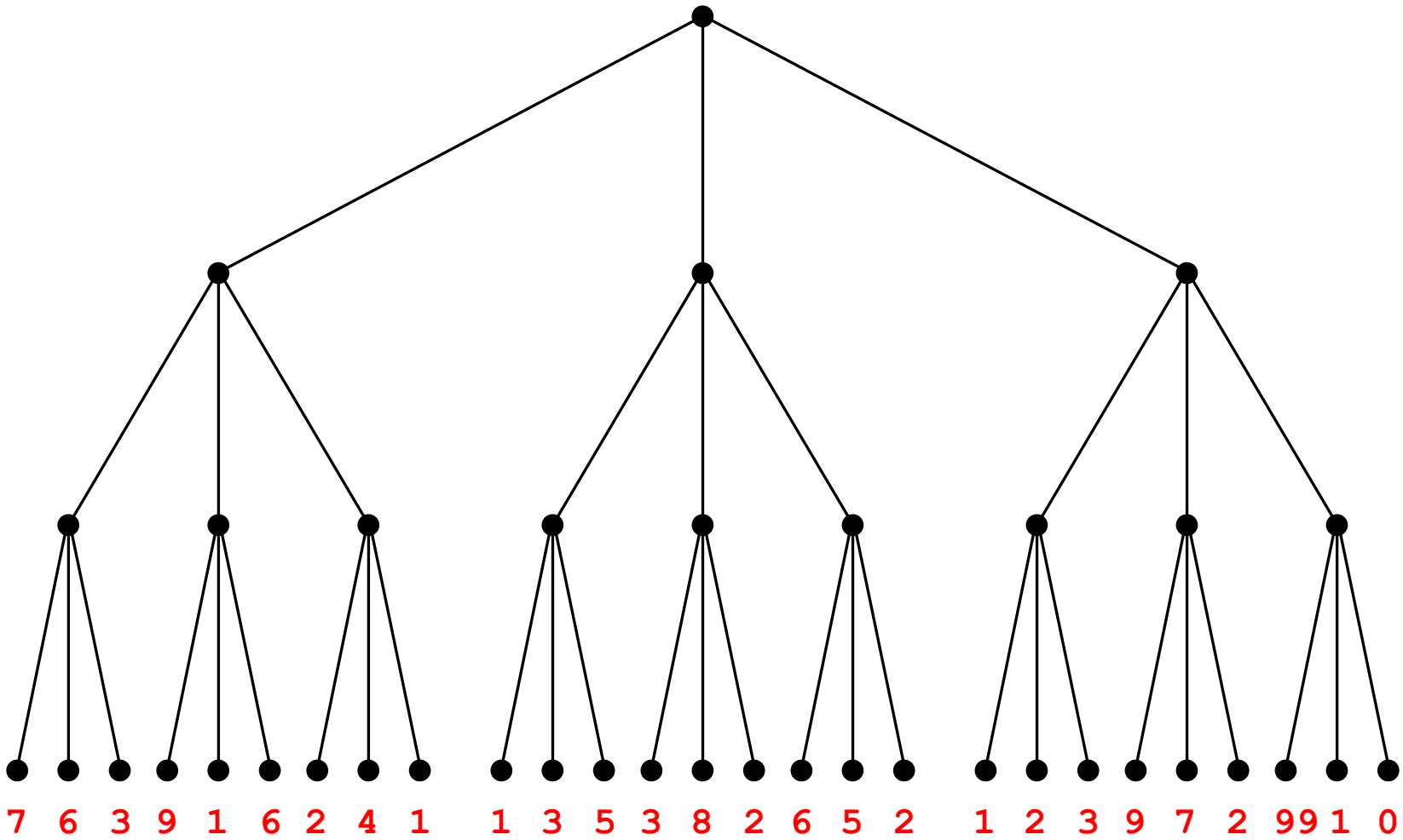
$v = \max\{v_1, \ldots, v_i\}$,

with **i** the smallest index such that

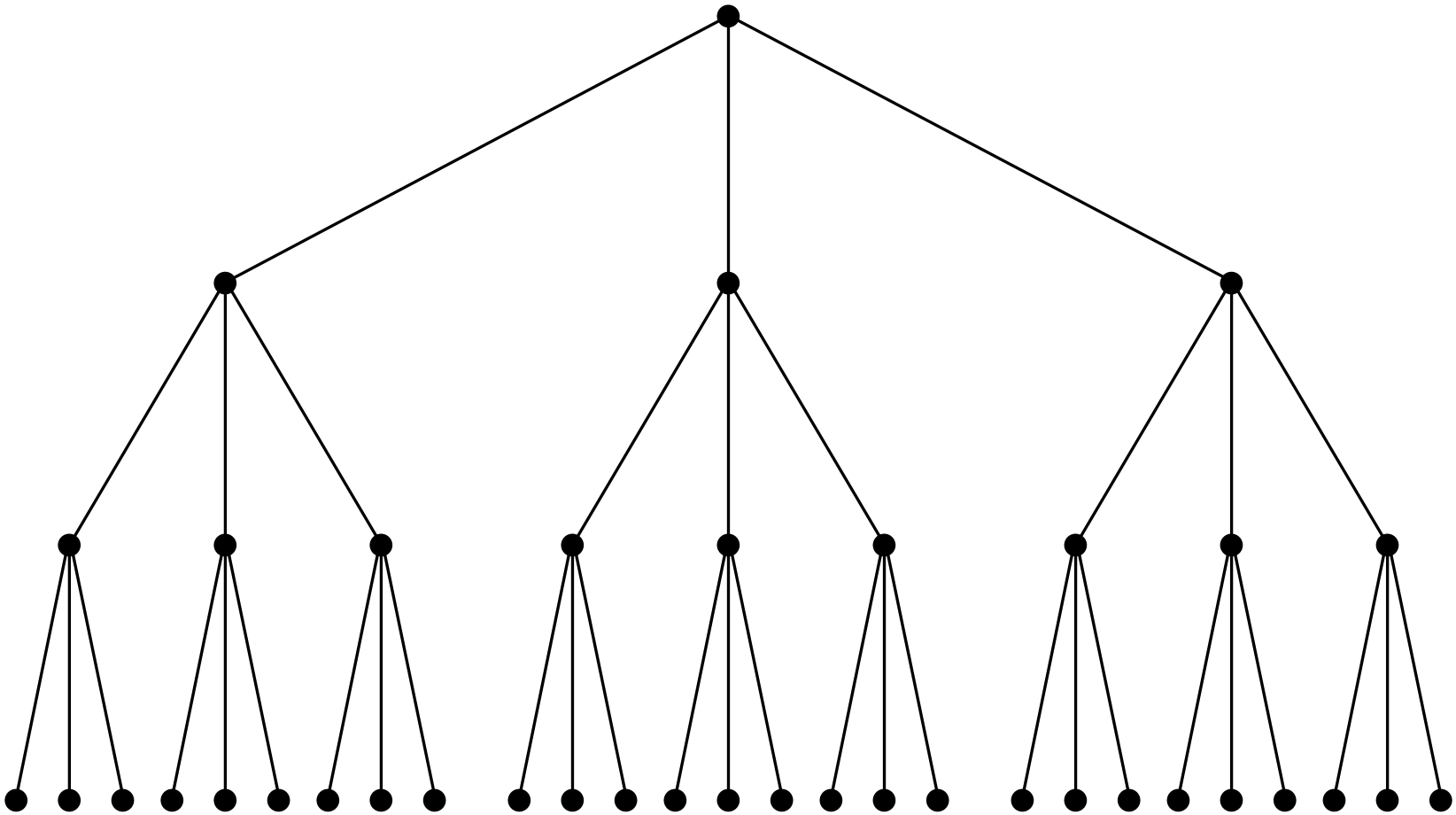
$v_i \geq \beta$, if that occurs,

and **i=k** otherwise.

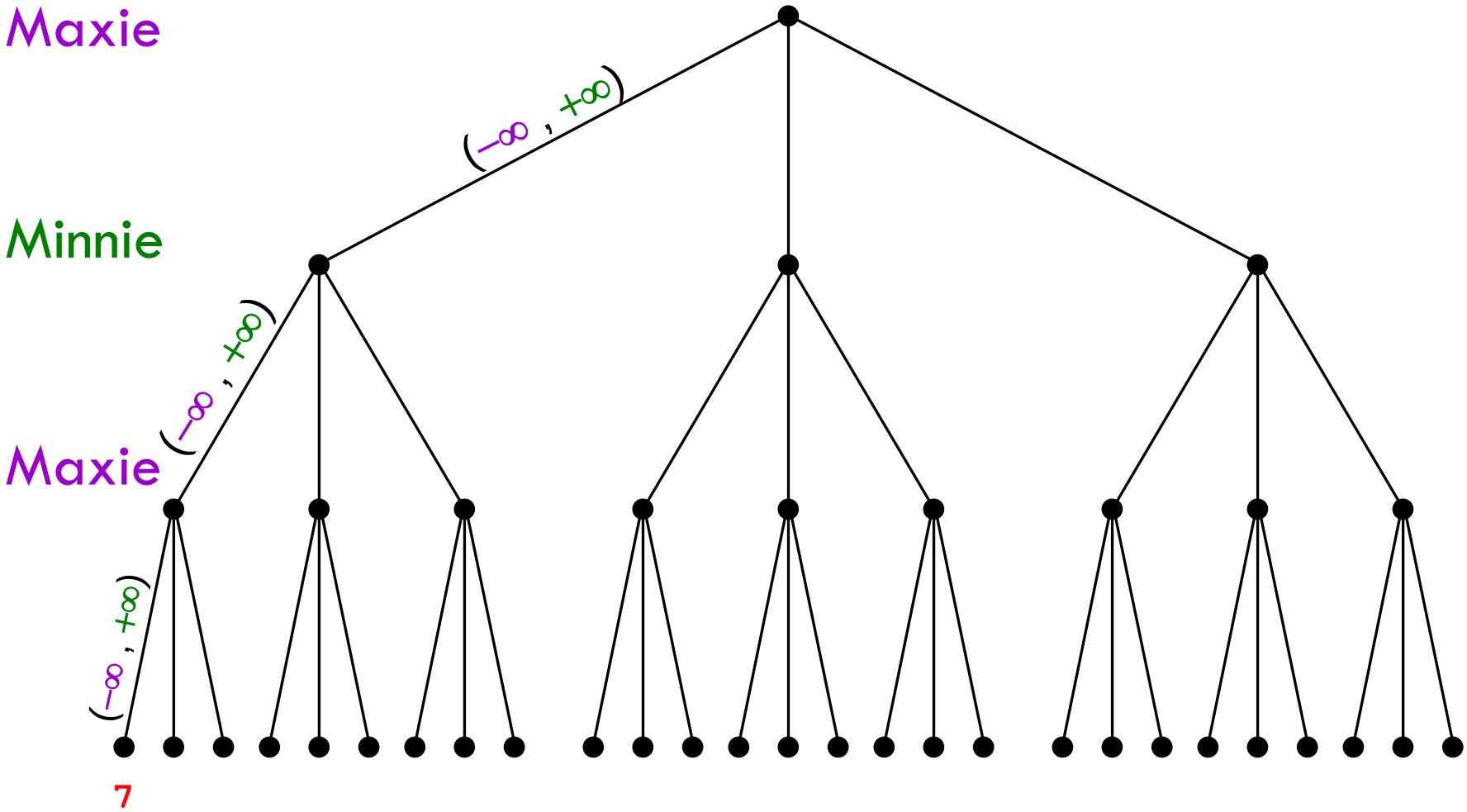
Sample Pruning (P.H. Winston, *Artificial Intelligence*, 1977, p.118):

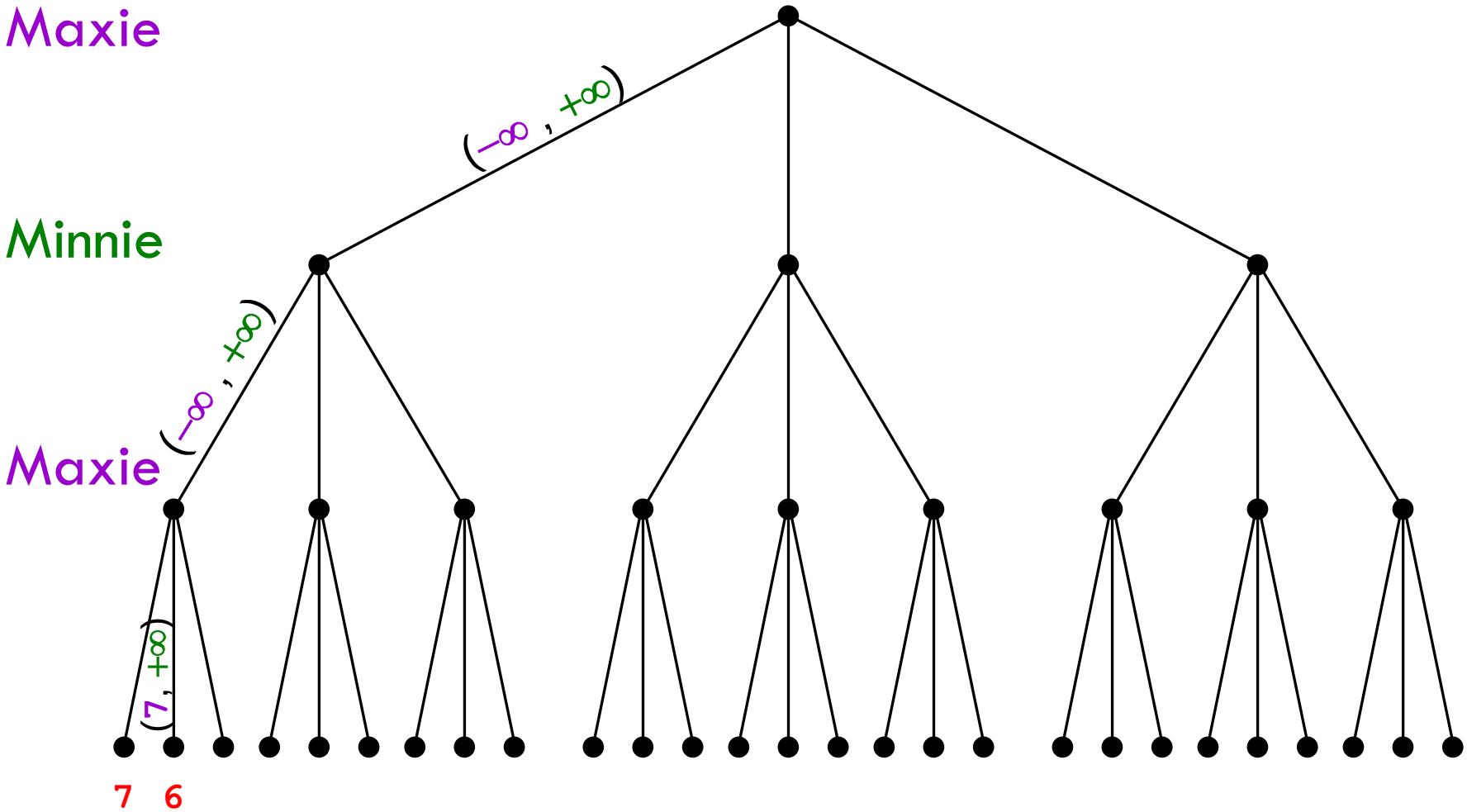


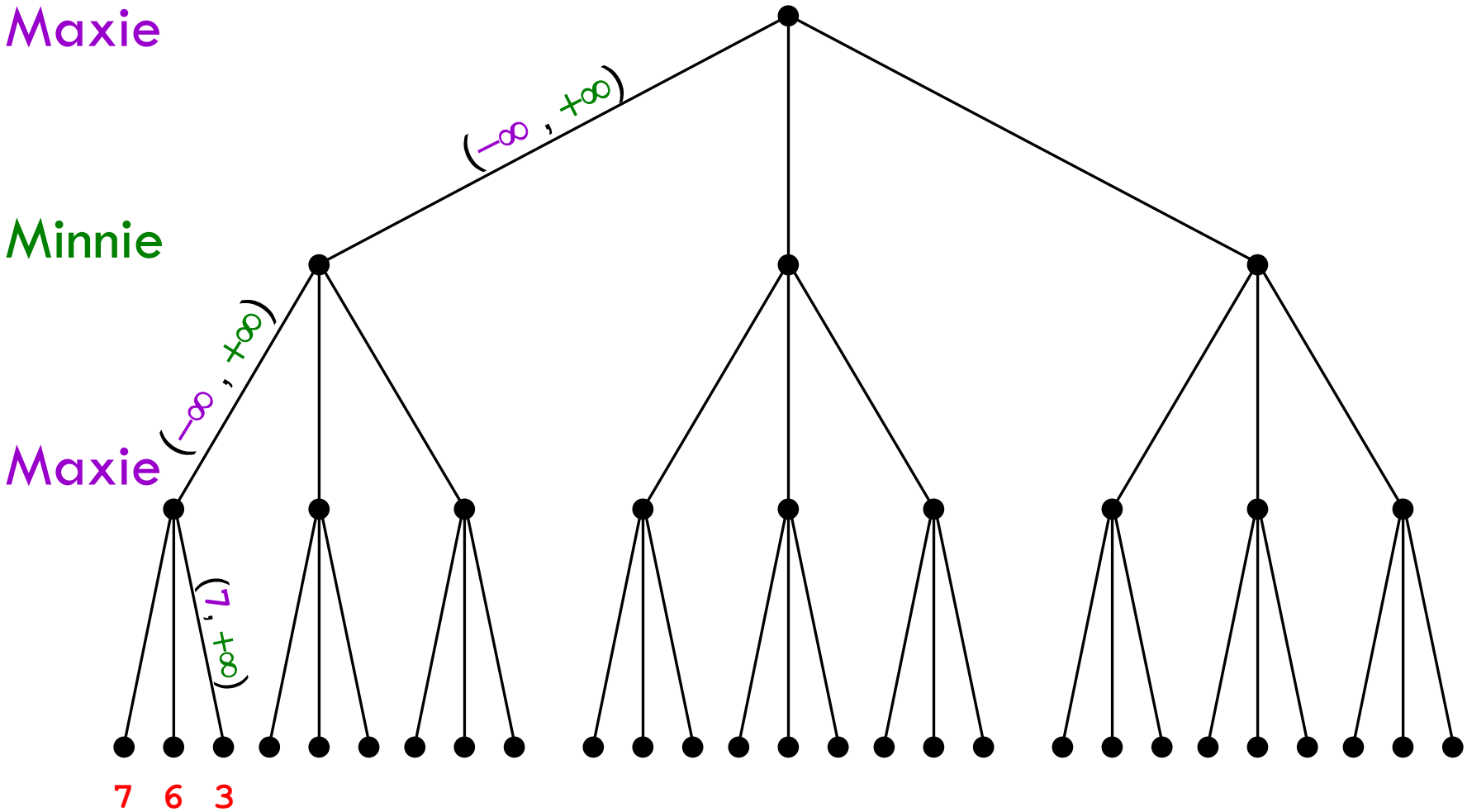
Let’s hide the leaf values until the search needs them.

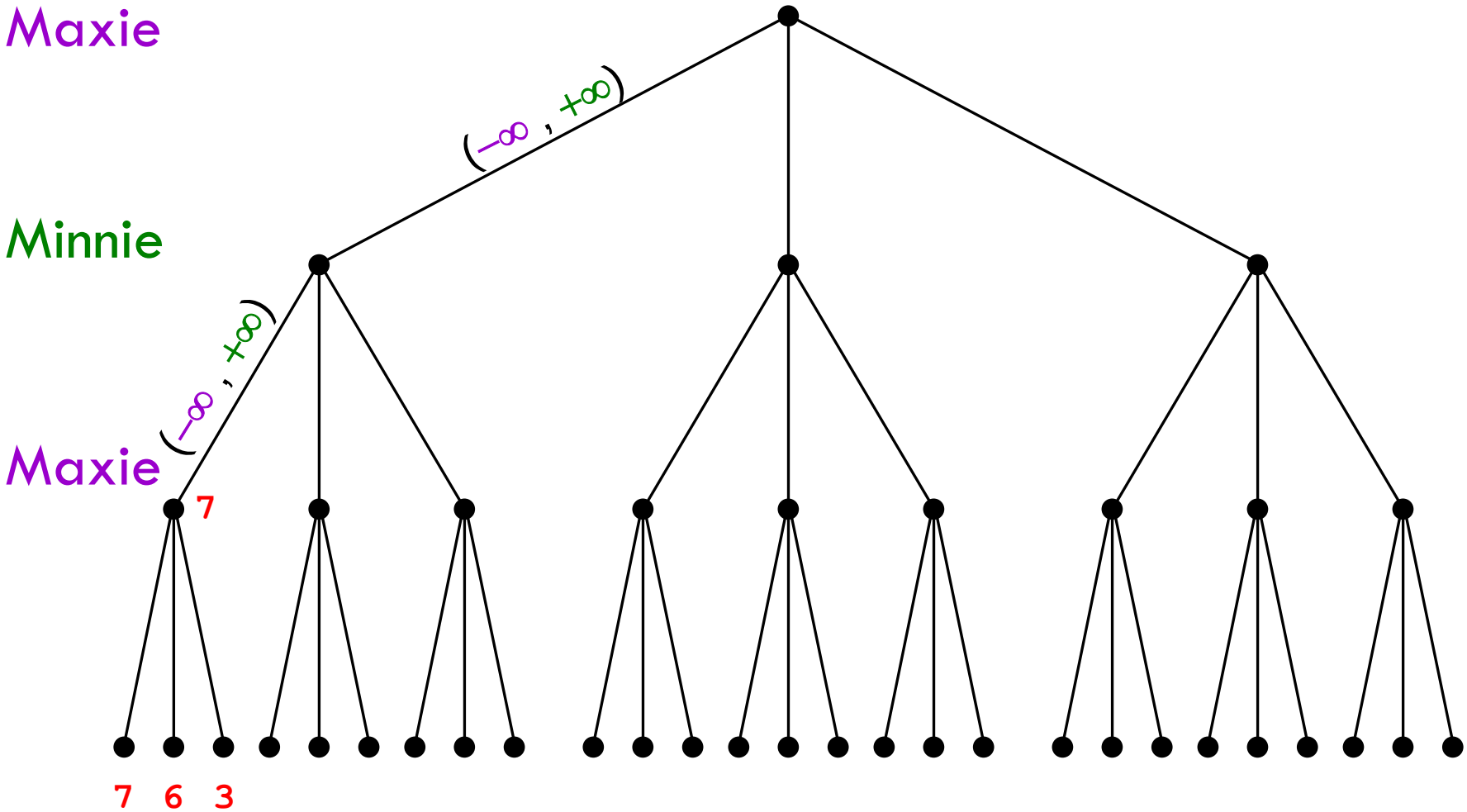


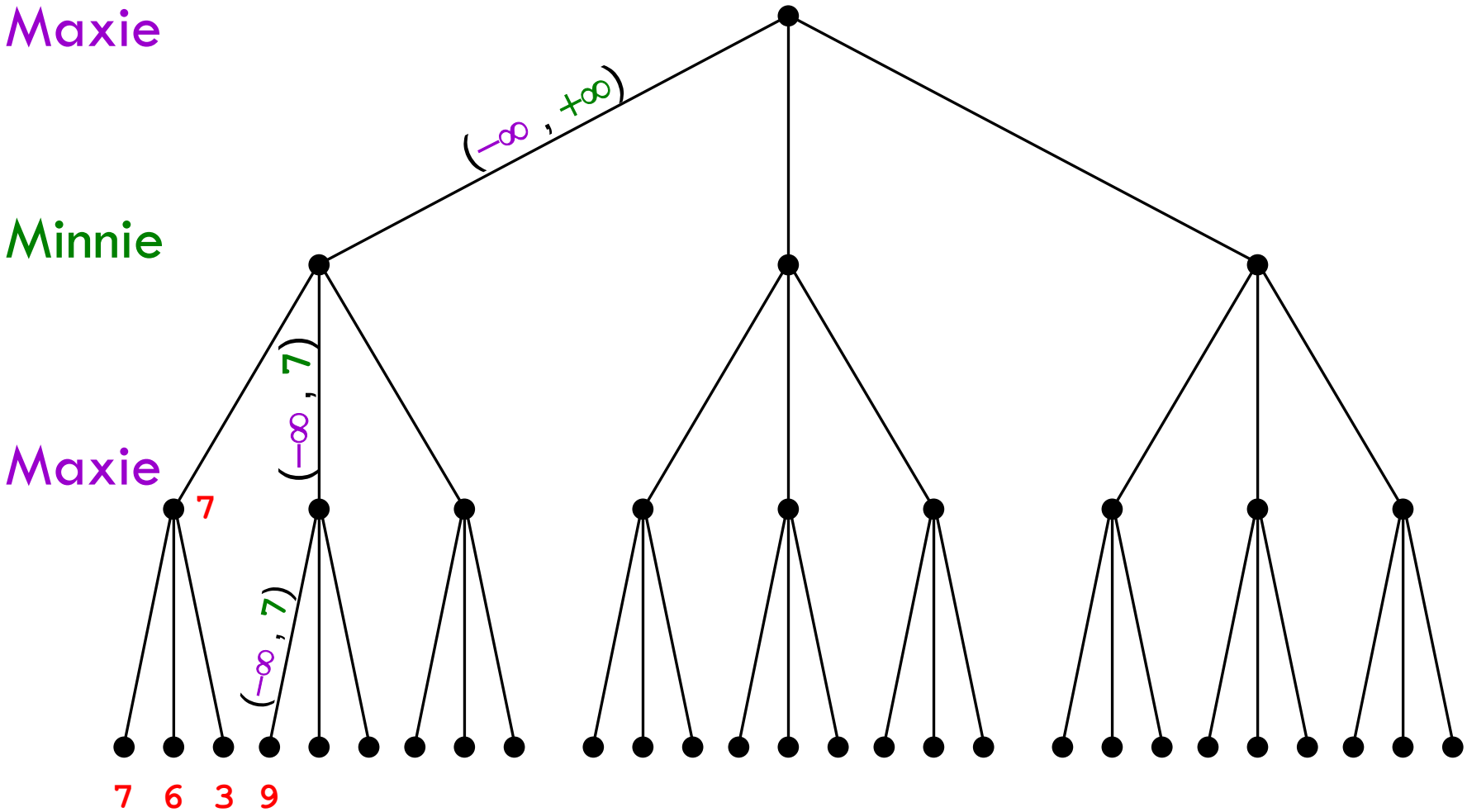
We'll label nodes with values **v** and edges with (α,β) intervals and update these dynamically as the search progresses.

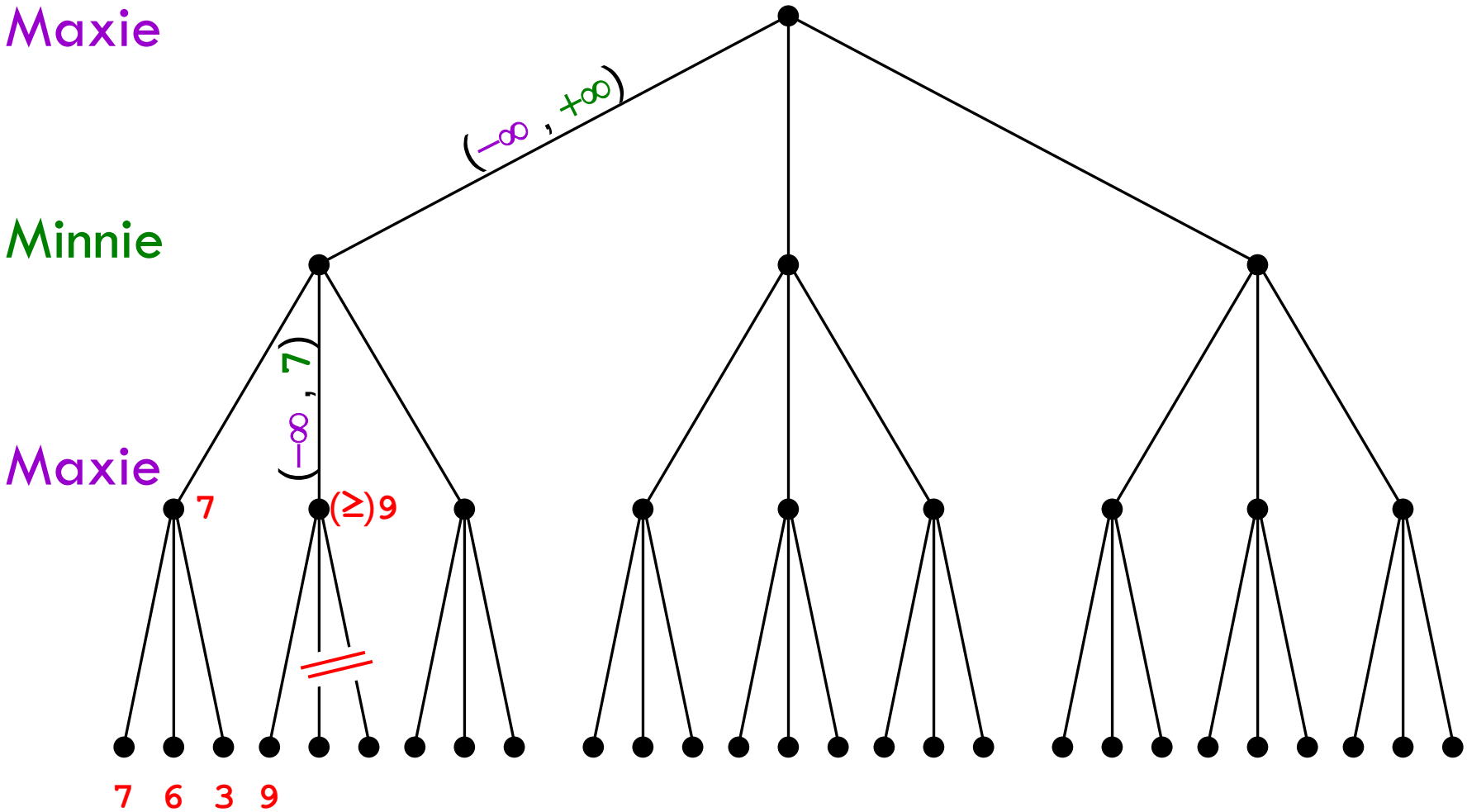



Maxie
Minnie
Maxie
(−∞ , +∞)
(−∞ , +∞)
(7, +∞)
7
6

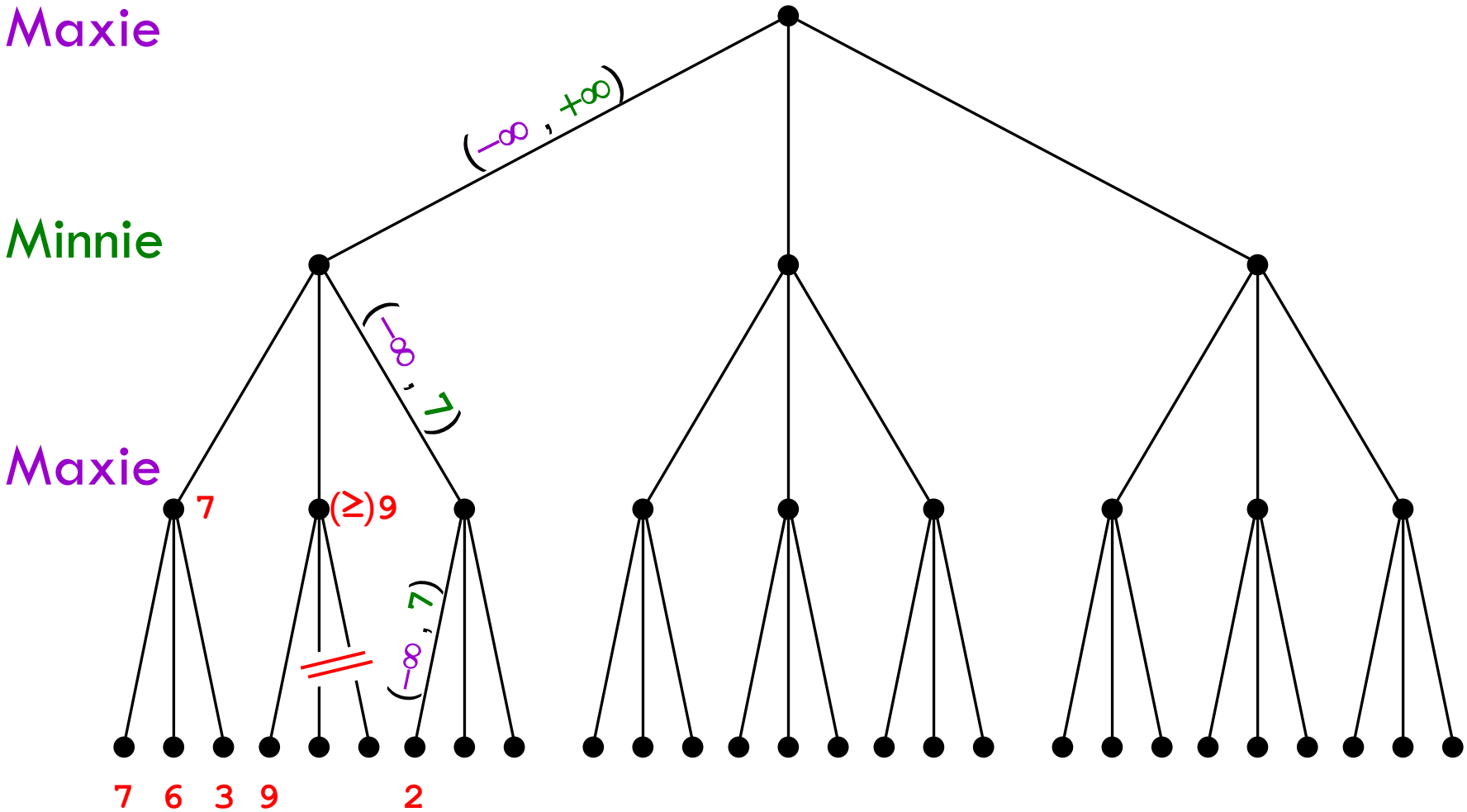

Maxie
Minnie
Maxie
(−∞ , +∞)
(−∞ , +∞)
(7, +∞)
7 6 3

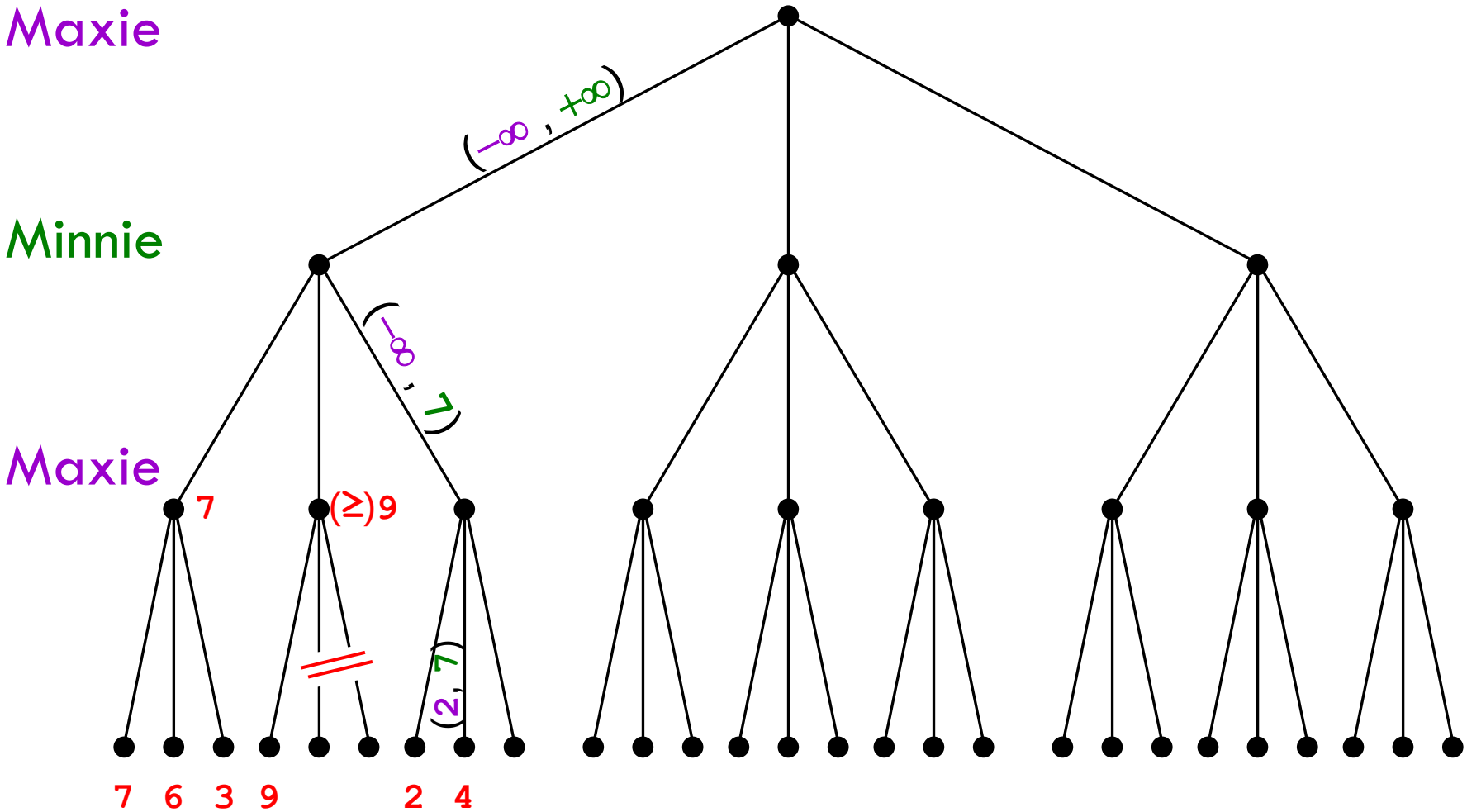

Maxie
Minnie
Maxie
(−∞ , +∞)
(−∞ , +∞)
7
7 6 3

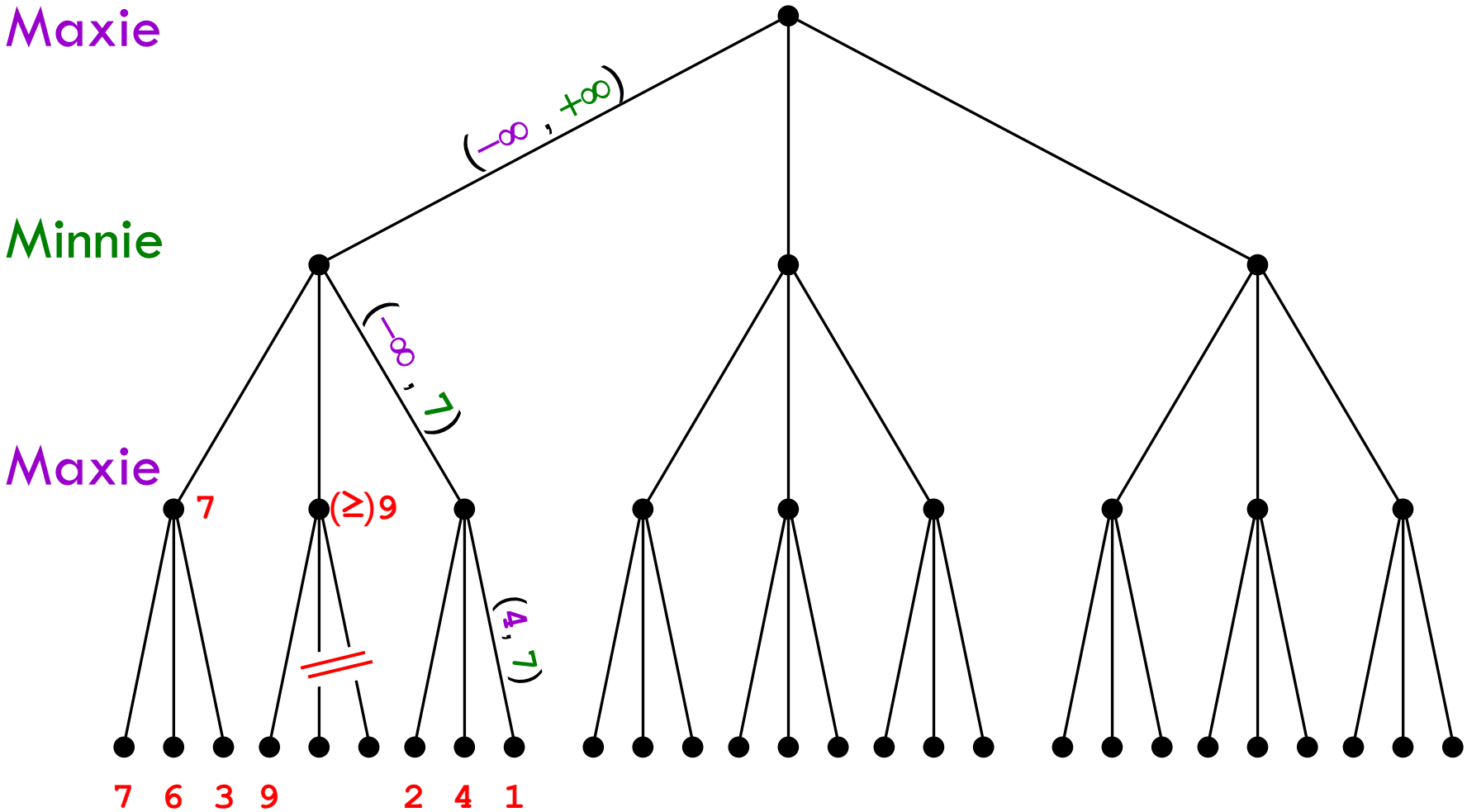

Maxie
Minnie
Maxie
(−∞ , +∞)
(−∞ , 7)
(−∞ , 7)
7
7 6 3 9

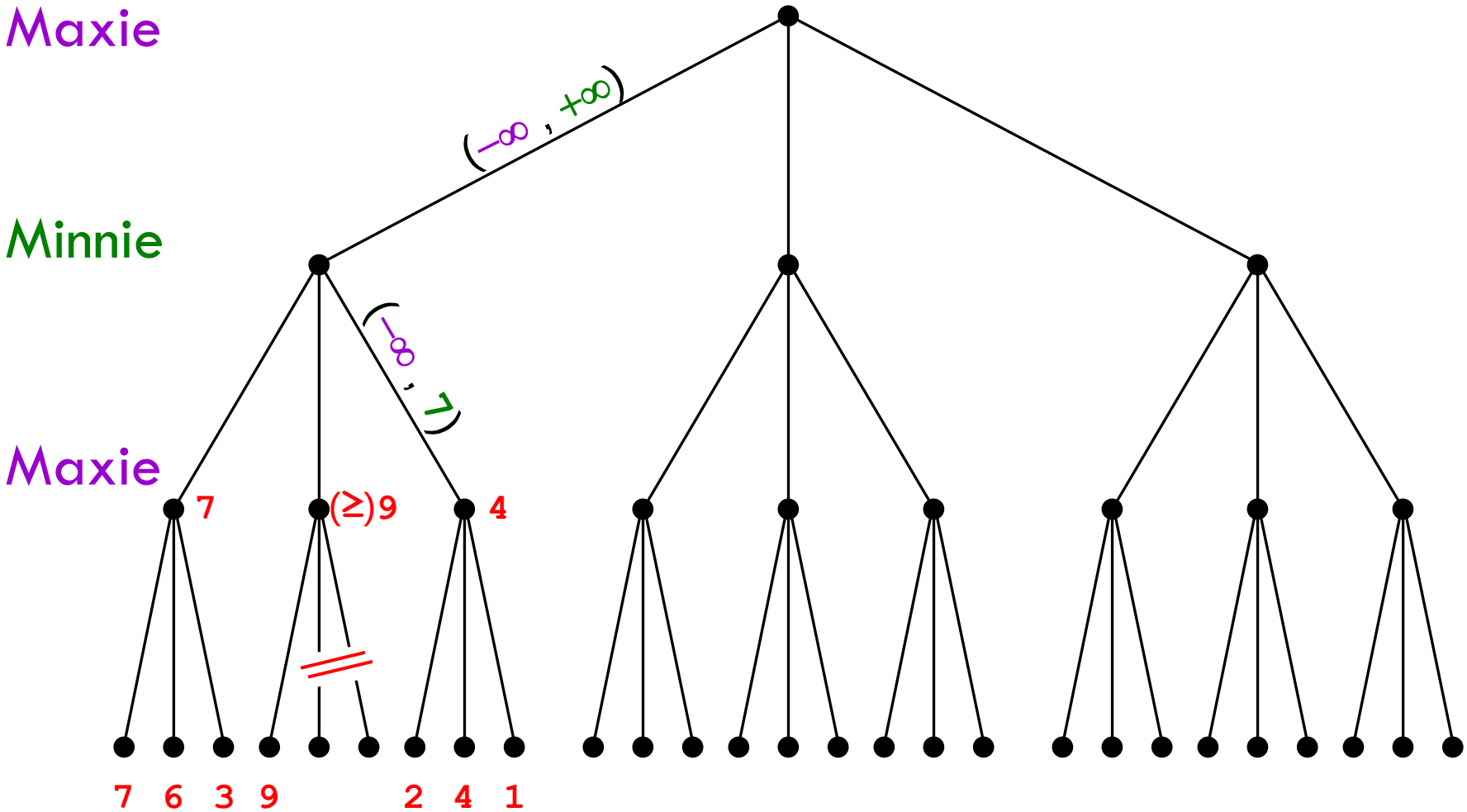


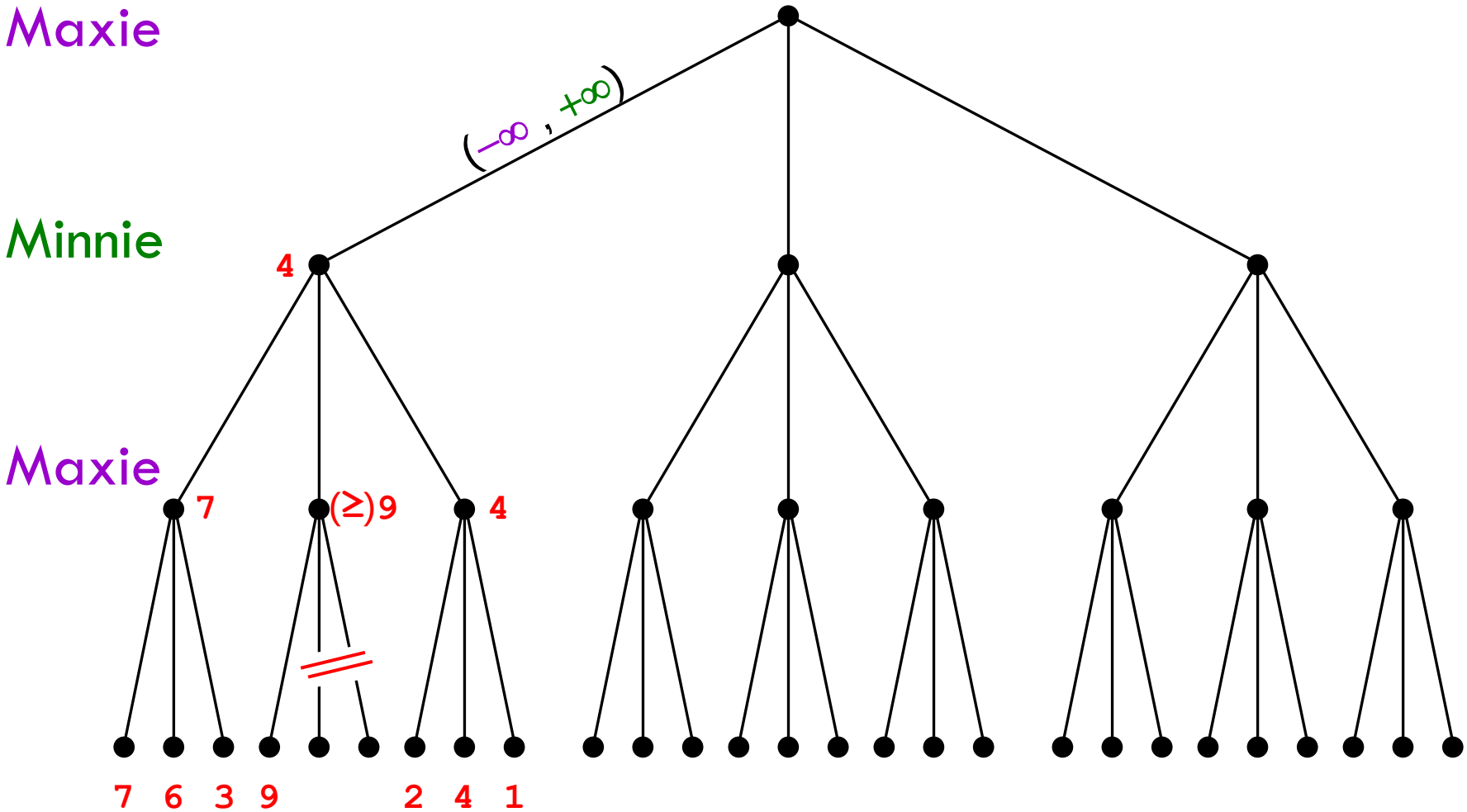
// means "pruned" (in this case because at a **Maxie** level $v \geq 9 > 7 = \beta$).

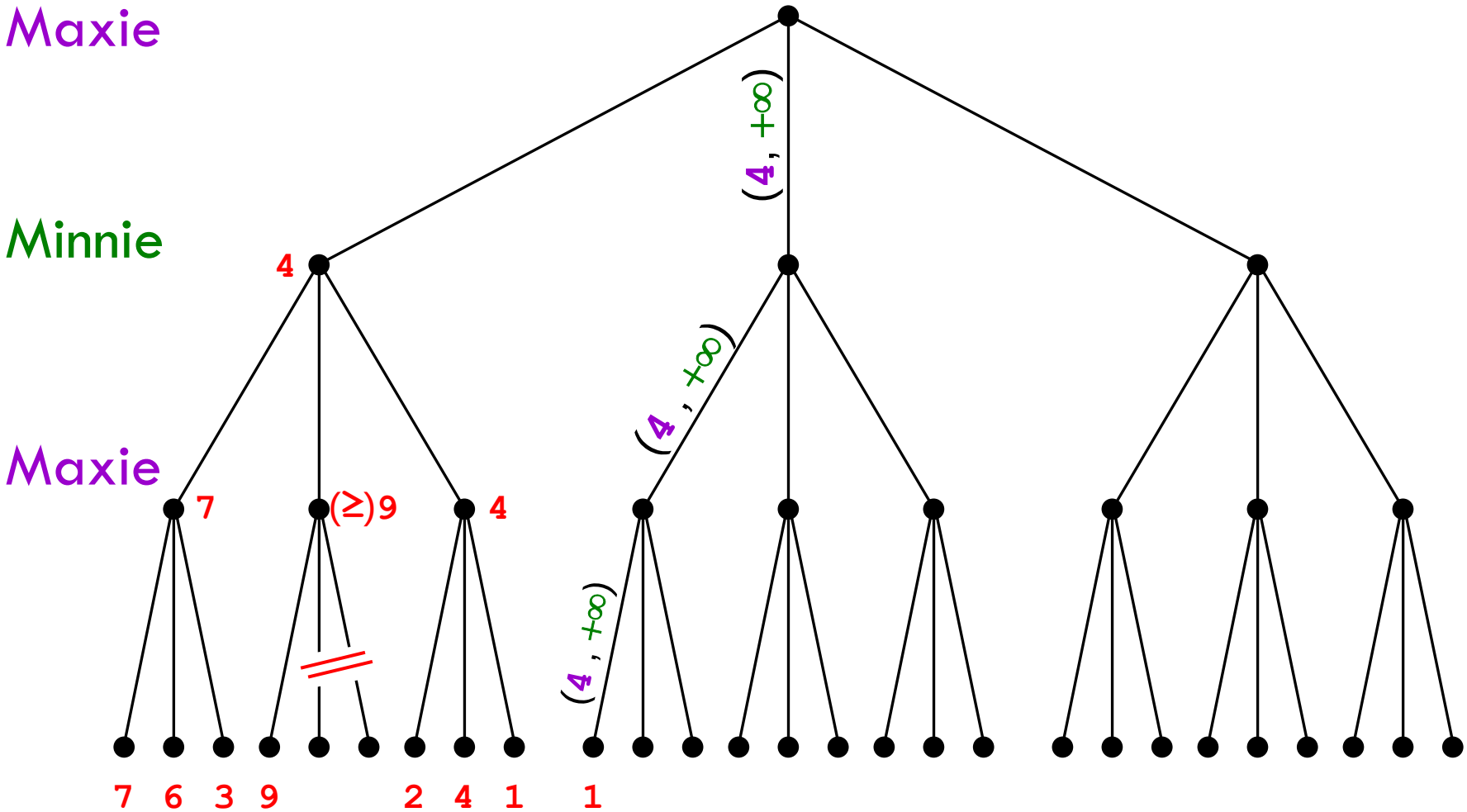

Maxie
Minnie
Maxie
(−∞ , +∞)
(−∞ , 7)
(−∞ , 7)
7
(≥)9
7 6 3 9 2

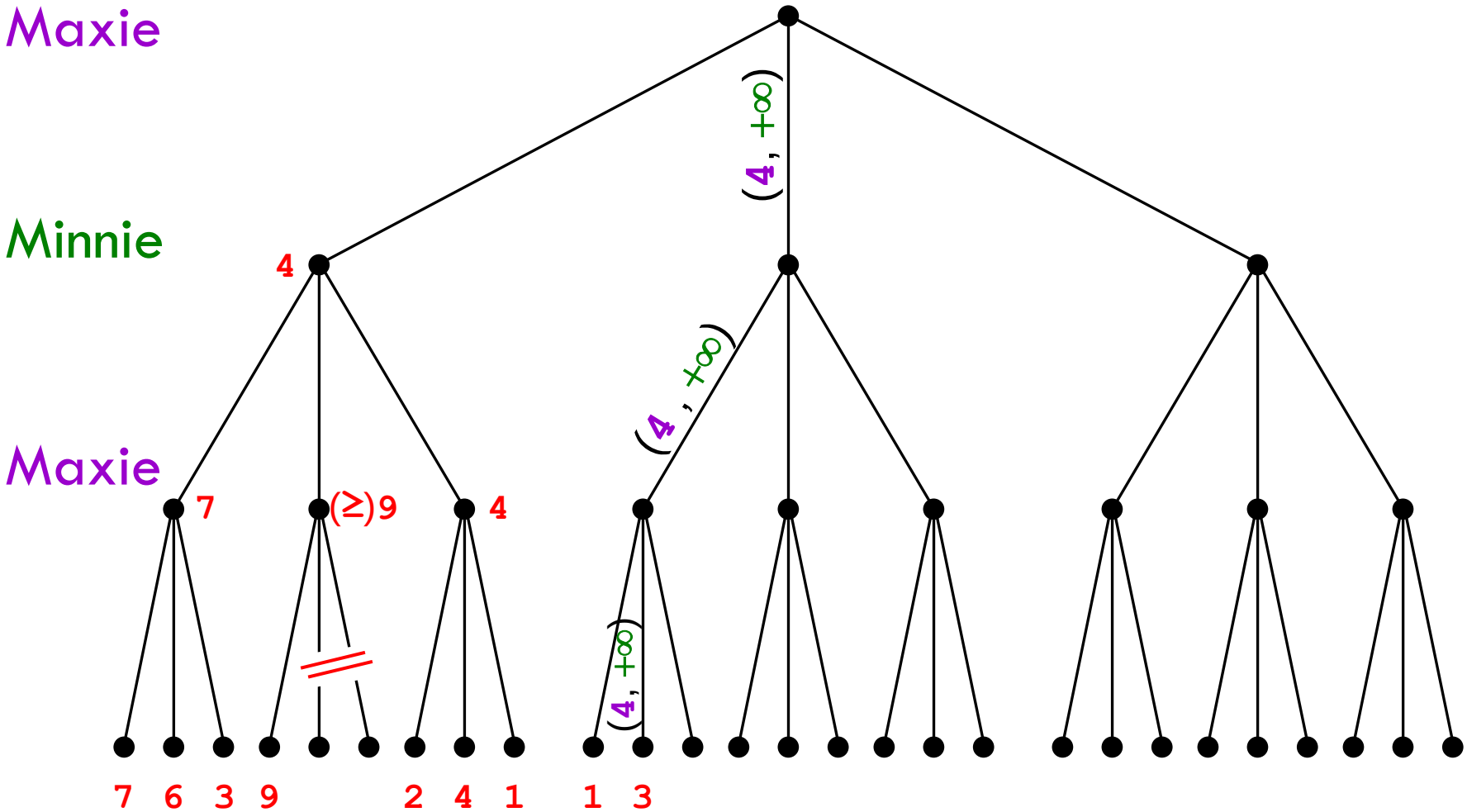

Maxie
Minnie
Maxie
(−∞ , +∞)
(−∞ , 7)
(2, 7)
7
(≥)9
7 6 3 9 2 4

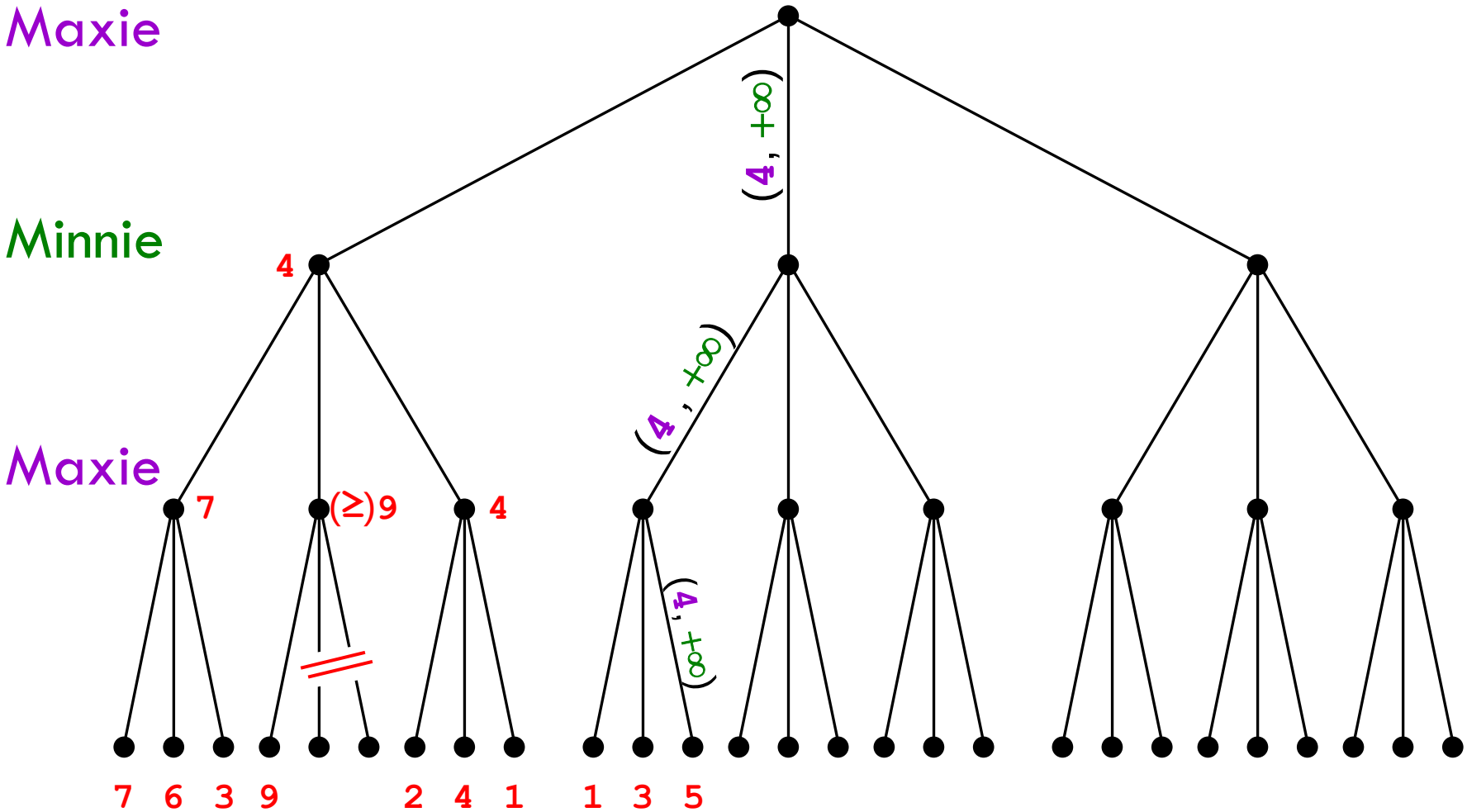

Maxie
Minnie
Maxie
(−∞ , +∞)
(−∞ , 7)
(4 , 7)
7
(≥)9
7 6 3 9 2 4 1

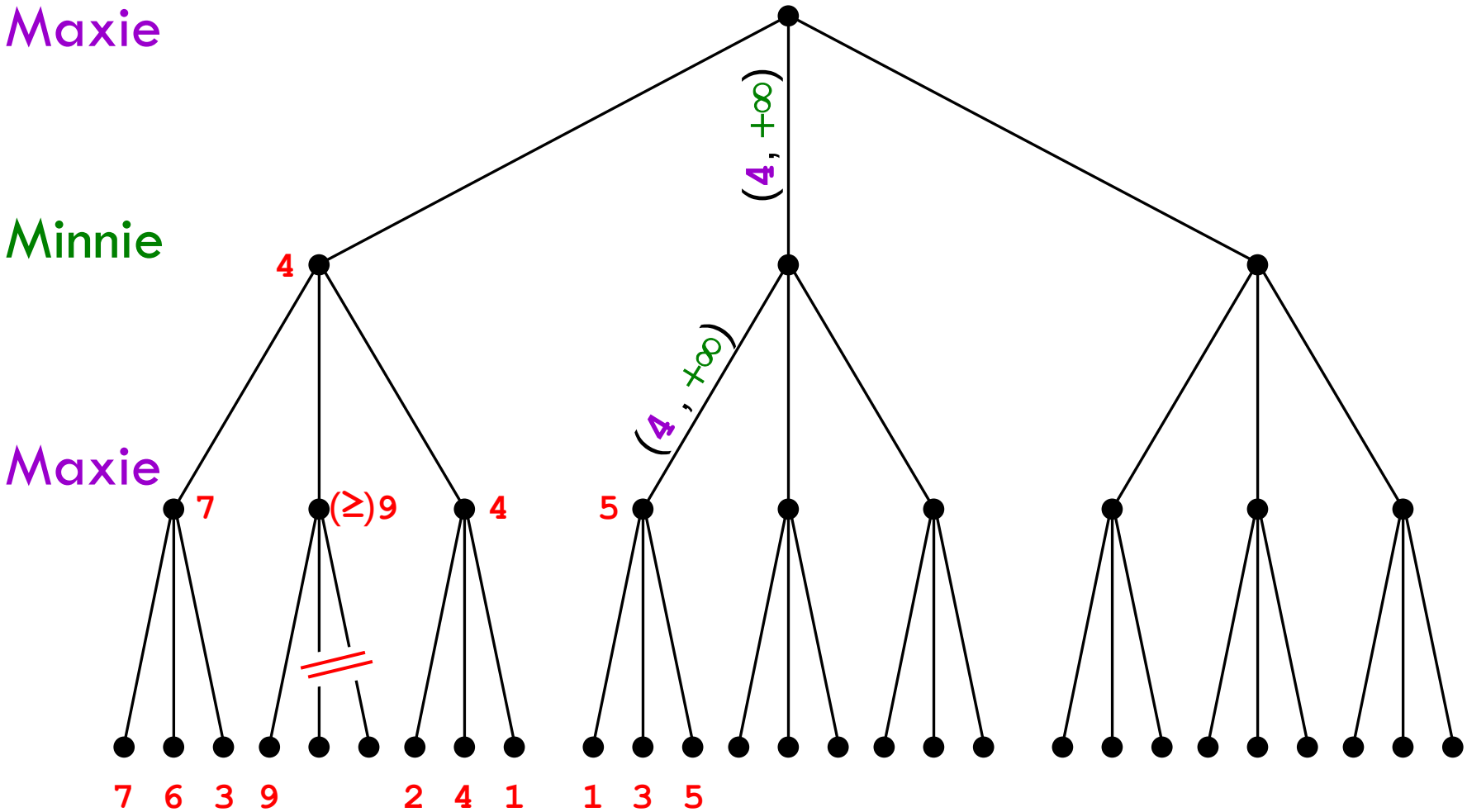

Maxie
Minnie
Maxie
(−∞ , +∞)
(−∞ , 7)
7
(≥)9
4
7 6 3 9
2 4 1

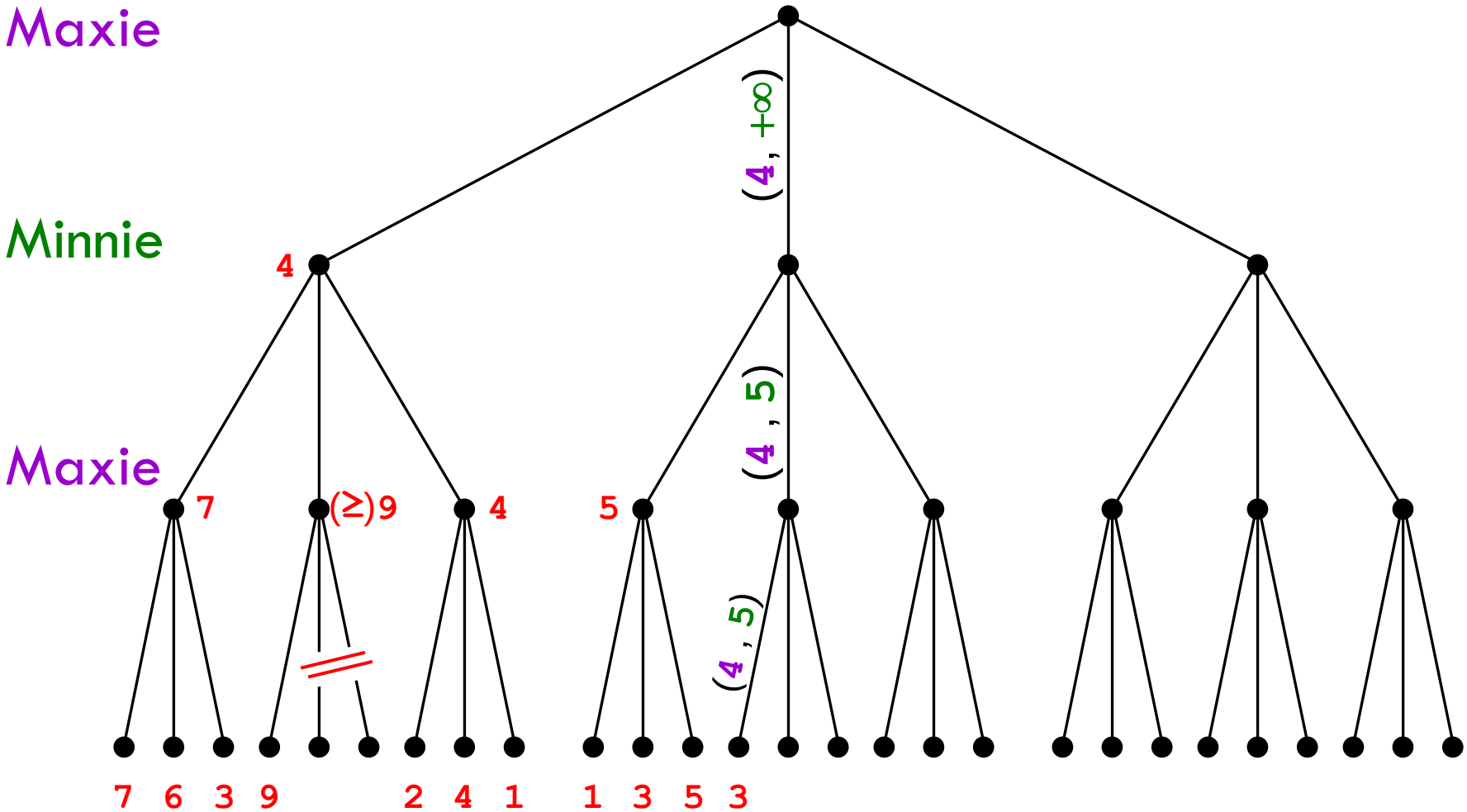

Maxie
Minnie
Maxie
(−∞ , +∞)
4
7
(≥)9
4
7 6 3 9
2 4 1

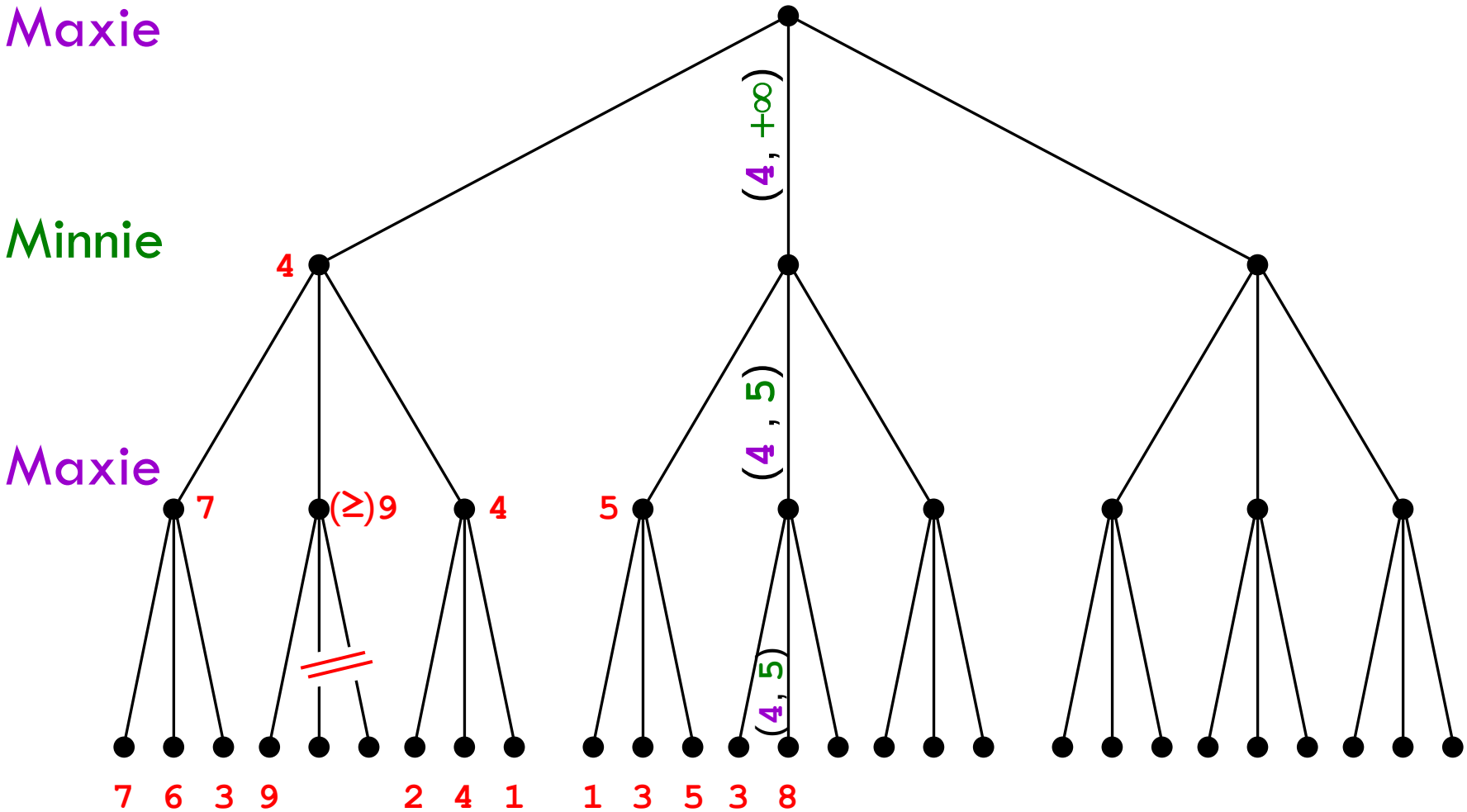

Maxie
Minnie
Maxie
4
7
(≥)9
4
(4, +∞)
(4, +∞)
(4, +∞)
7 6 3 9 2 4 1 1

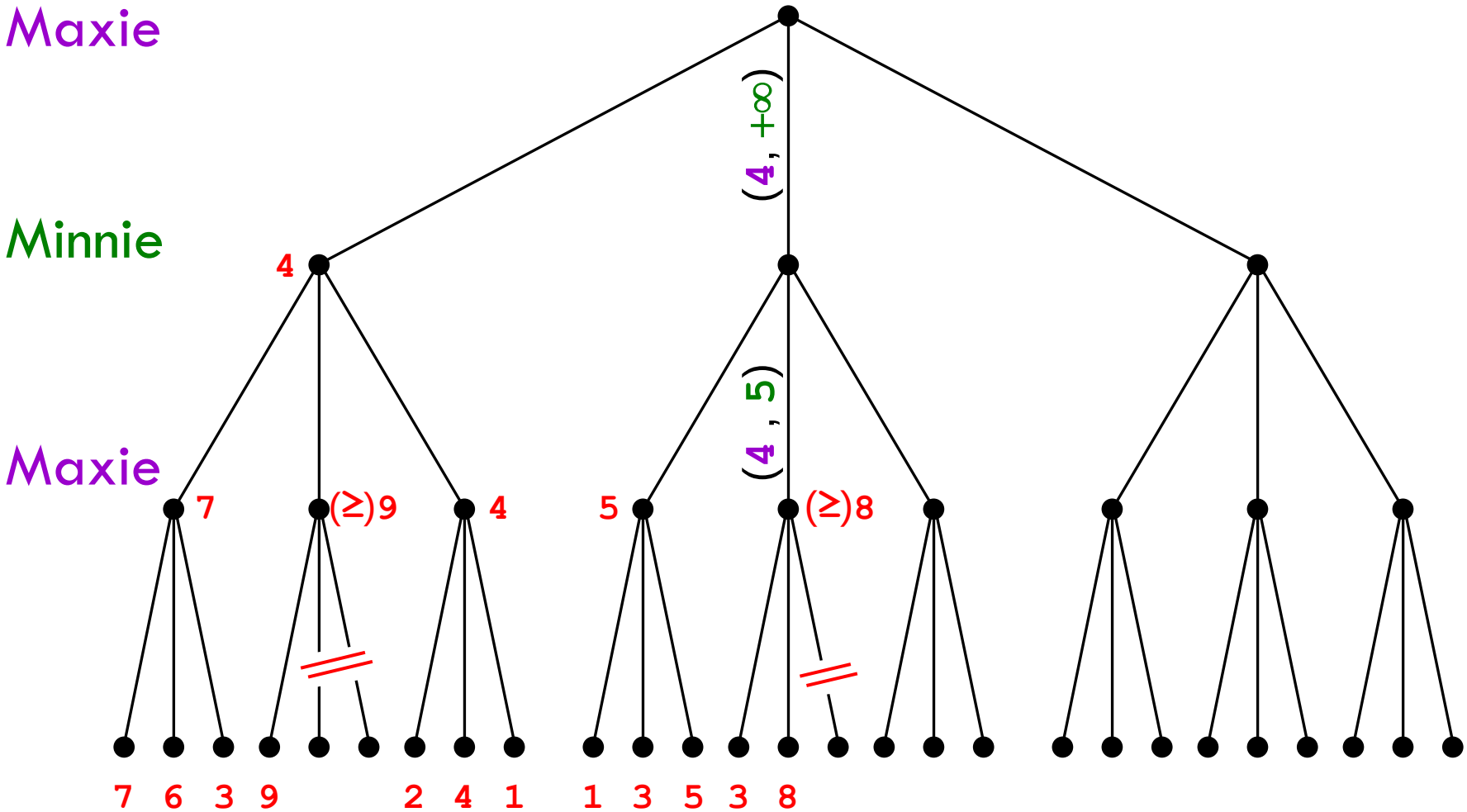

Maxie
Minnie
Maxie
4
7
(≥)9
4
(4, +∞)
(4, +∞)
(4, +∞)
7 6 3 9
2 4 1
1 3

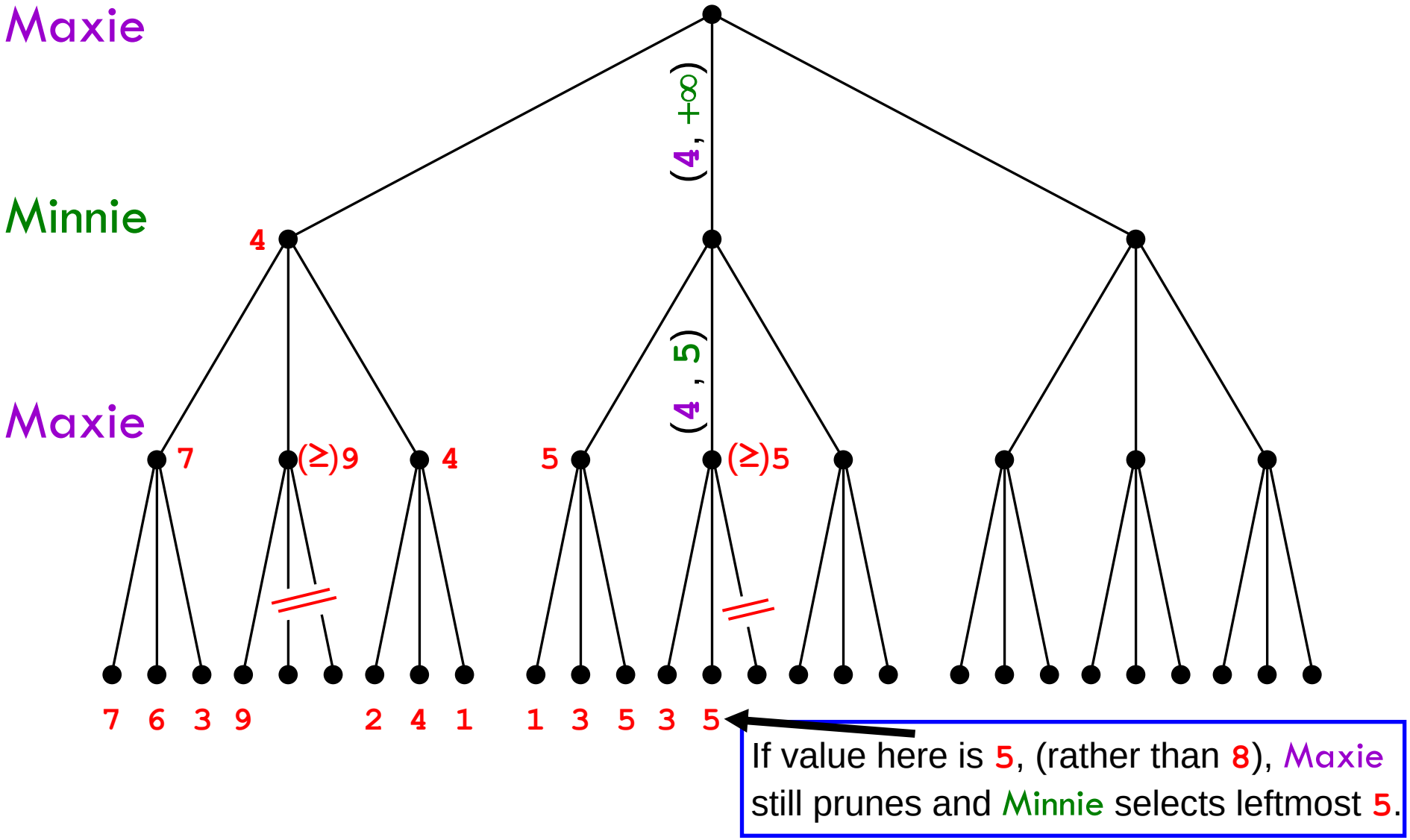

Maxie
Minnie
Maxie
4
7
(≥)9
4
(4, +∞)
(4, +∞)
(4, +∞)
7 6 3 9
2 4 1
1 3 5

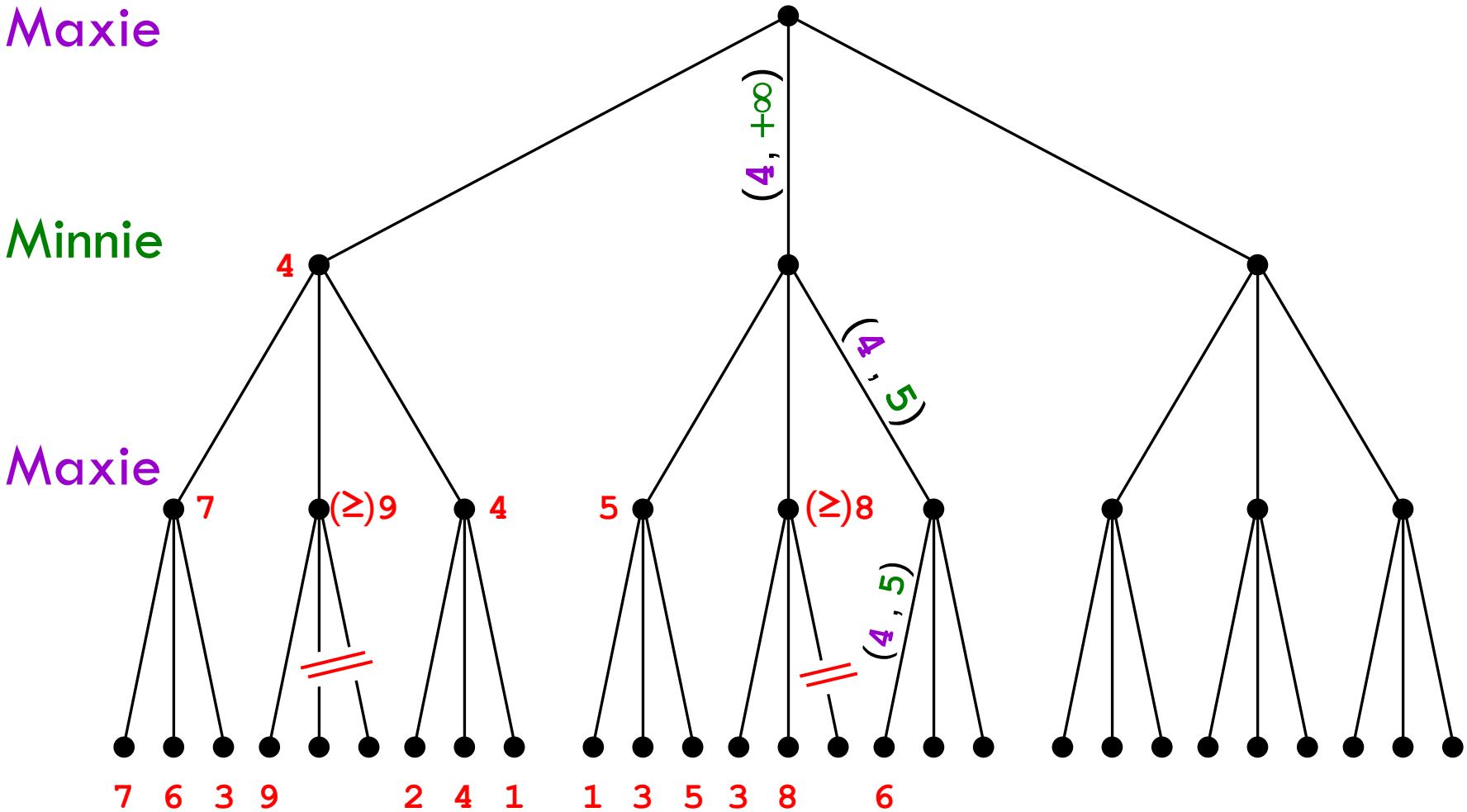

Maxie
Minnie
Maxie
4
7
(≥)9
4
5
(4, +∞)
(4, +∞)
7 6 3 9
2 4 1
1 3 5

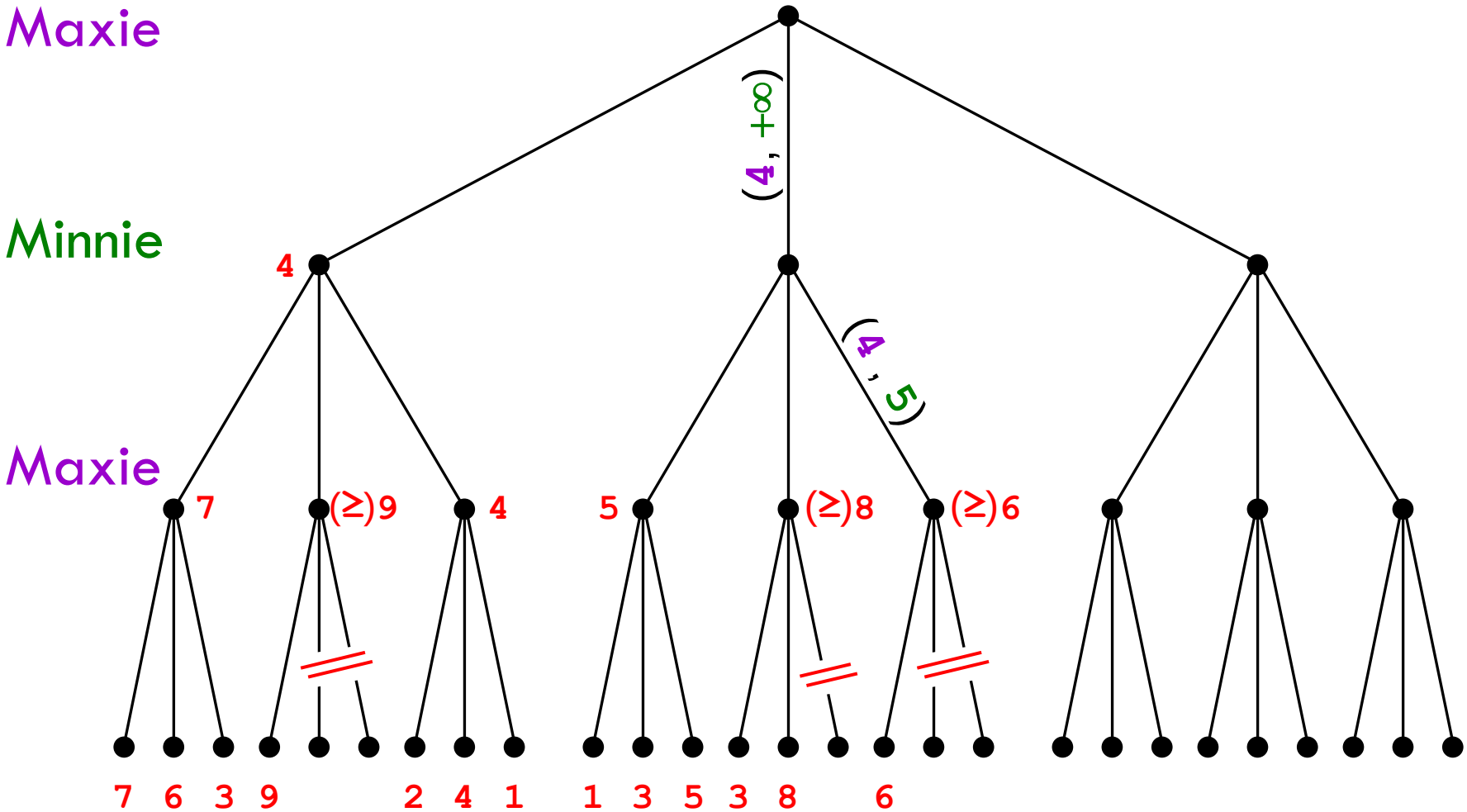

Maxie
Minnie
Maxie
(4, +∞)
(4, 5)
(4, 5)
4
7
(≥)9
4
5
7 6 3 9
2 4 1
1 3 5 3

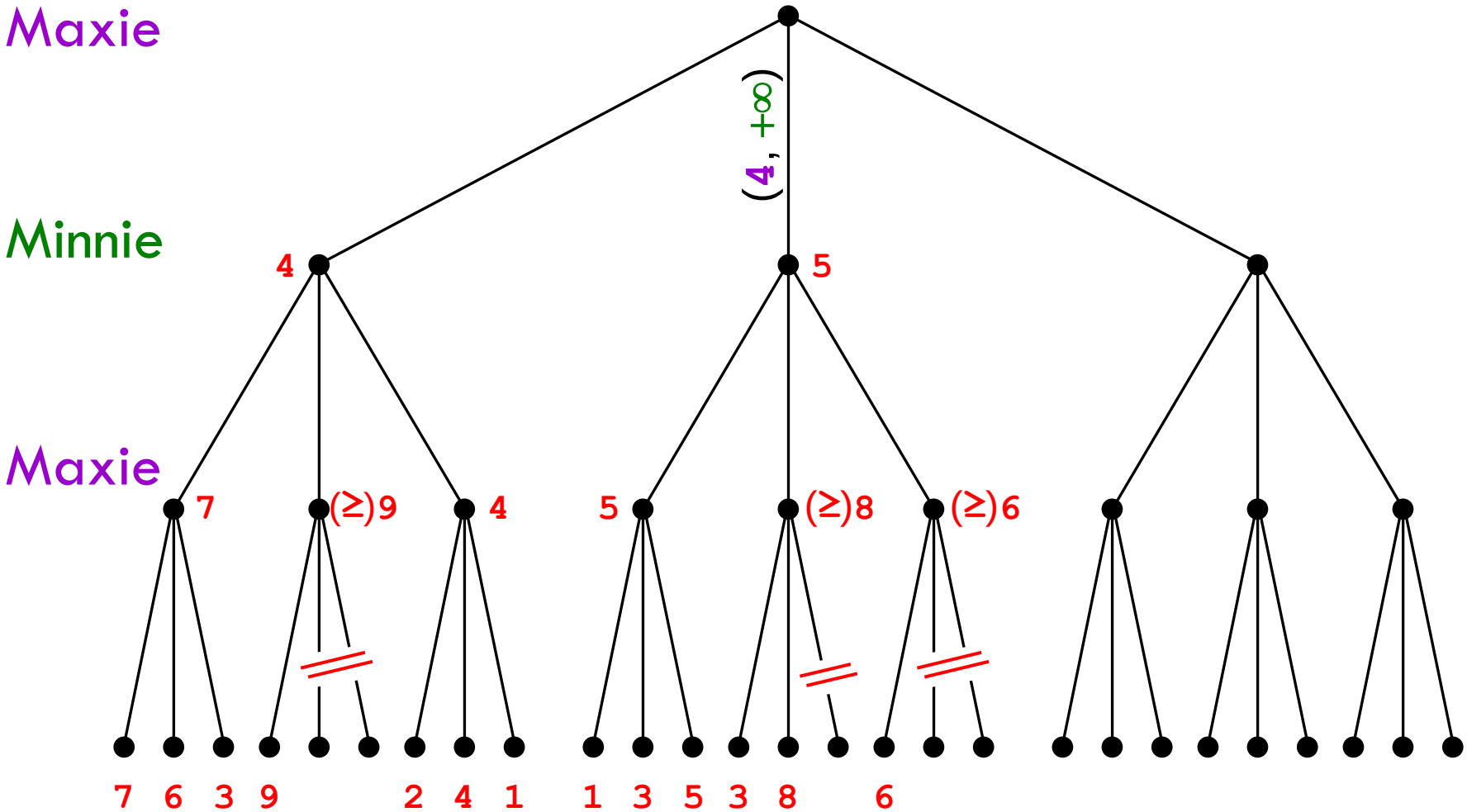

Maxie
Minnie
Maxie
(4, +∞)
(4 , 5)
(4, 5)
4
7
(≥)9
4
5
7 6 3 9
2 4 1
1 3 5 3 8

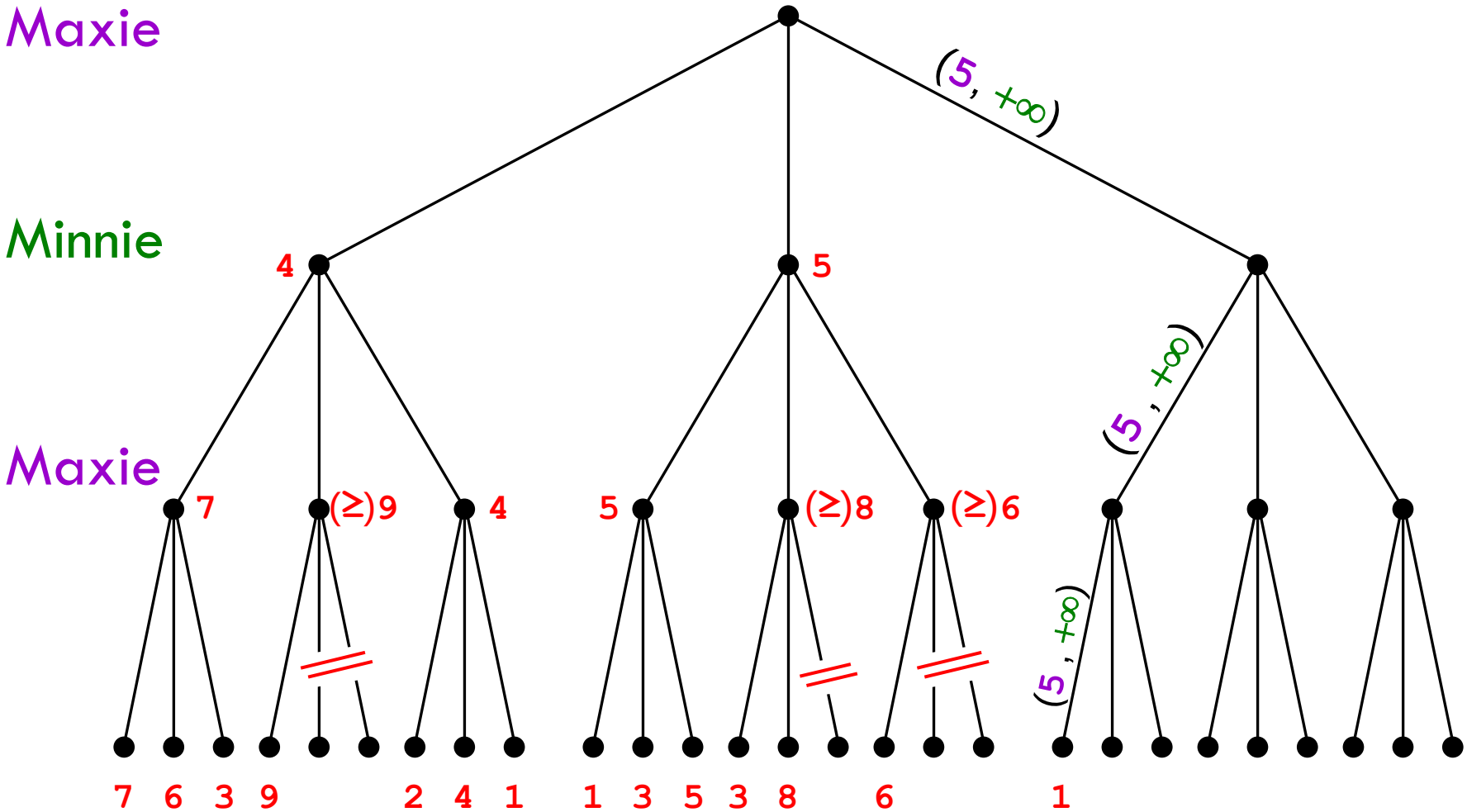

Maxie
Minnie
Maxie
(4, +∞)
(4 , 5)
4
7
(≥)9
4
5
(≥)8
7 6 3 9
2 4 1
1 3 5 3 8

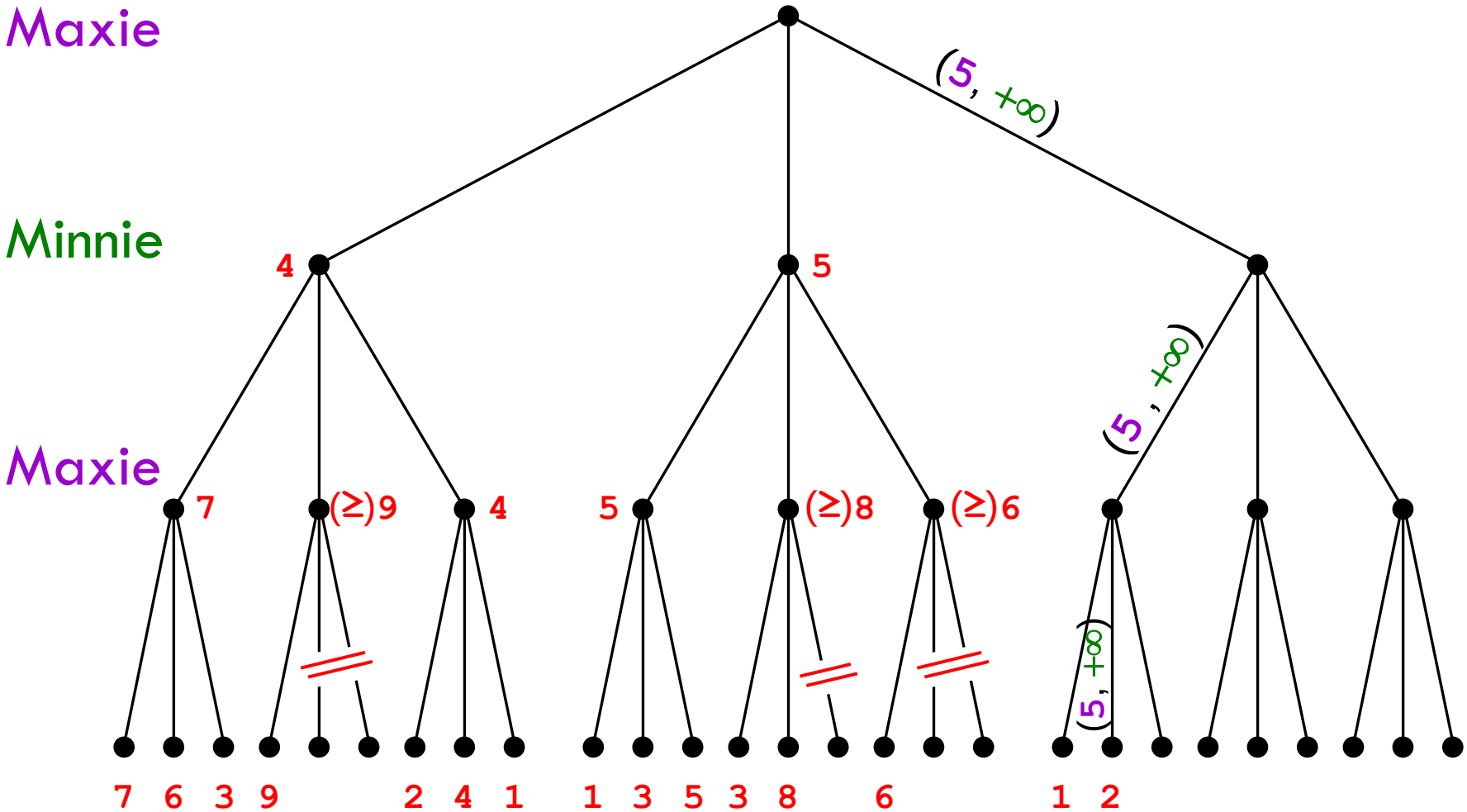

Maxie
Minnie
Maxie
(4, +∞)
(4 , 5)
4
7
(≥)9
4
5
(≥)5
7 6 3 9
2 4 1
1 3 5 3 5
If value here is 5, (rather than 8), Maxie still prunes and Minnie selects leftmost 5.

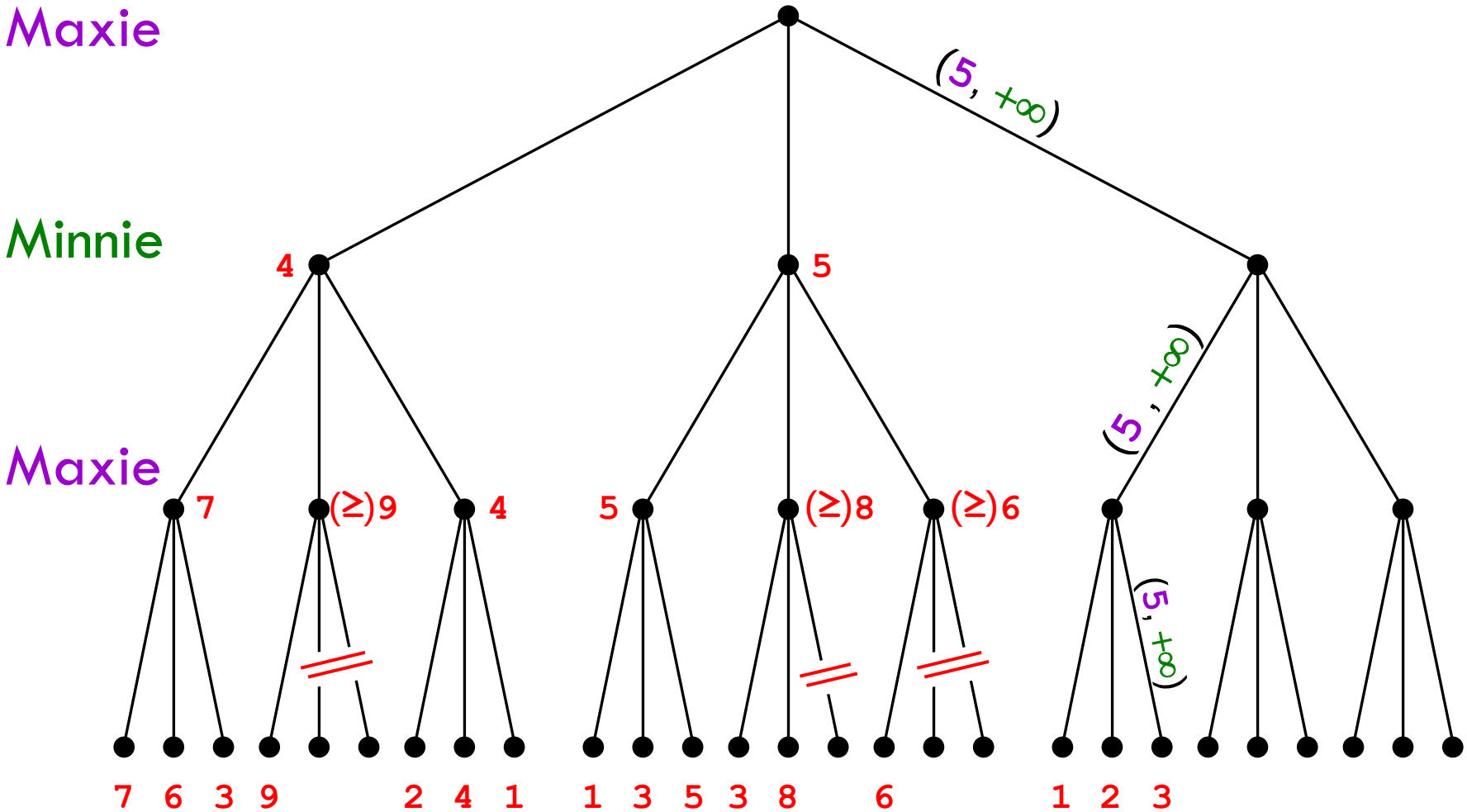

Maxie
Minnie
Maxie
(4, +∞)
(4, 5)
(4, 5)
4
7
(≥)9
4
5
(≥)8
7 6 3 9
2 4 1
1 3 5 3 8
6

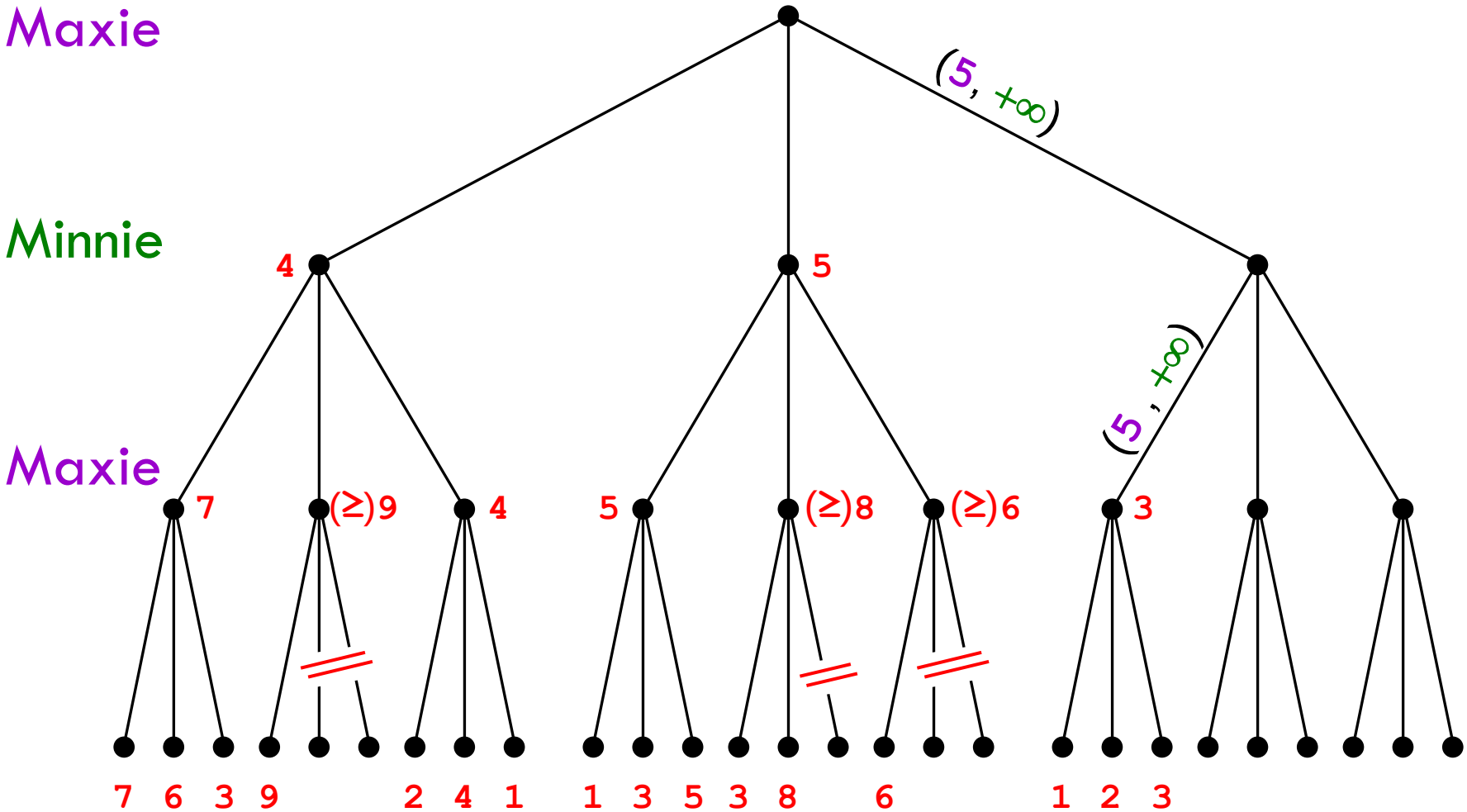

Maxie
Minnie
Maxie
(4, +∞)
(4, 5)
4
7
(≥)9
4
5
(≥)8
(≥)6
7 6 3 9
2 4 1
1 3 5 3 8
6

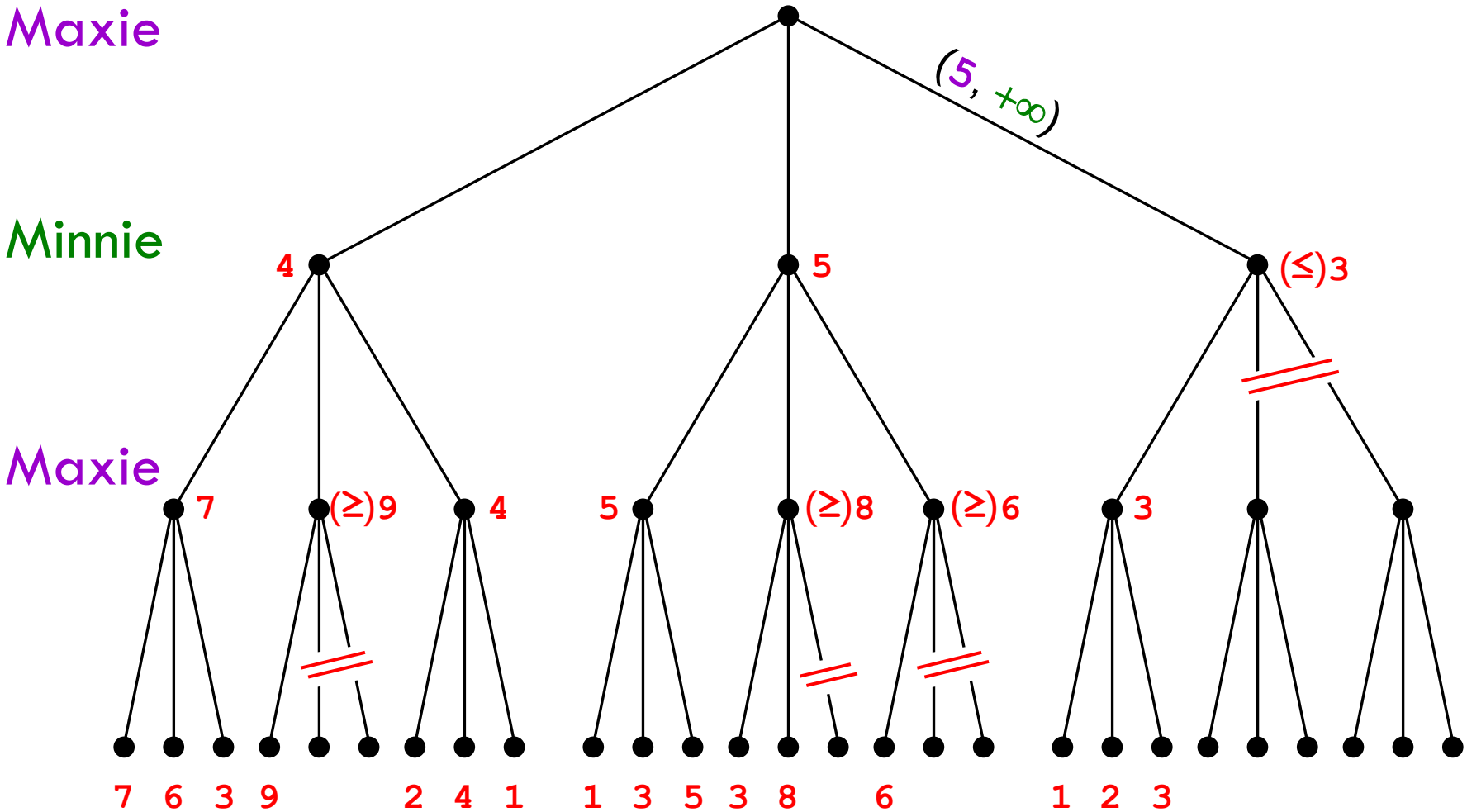

Maxie
Minnie
Maxie
(4, +∞)
4
5
7
(≥)9
4
5
(≥)8
(≥)6
7 6 3 9
2 4 1
1 3 5 3 8
6


Maxie
Minnie
Maxie
(5, +∞)
4
5
(5, +∞)
(5, +∞)
7
(≥)9
4
5
(≥)8
(≥)6
7 6 3 9
2 4 1
1 3 5 3 8
6
1

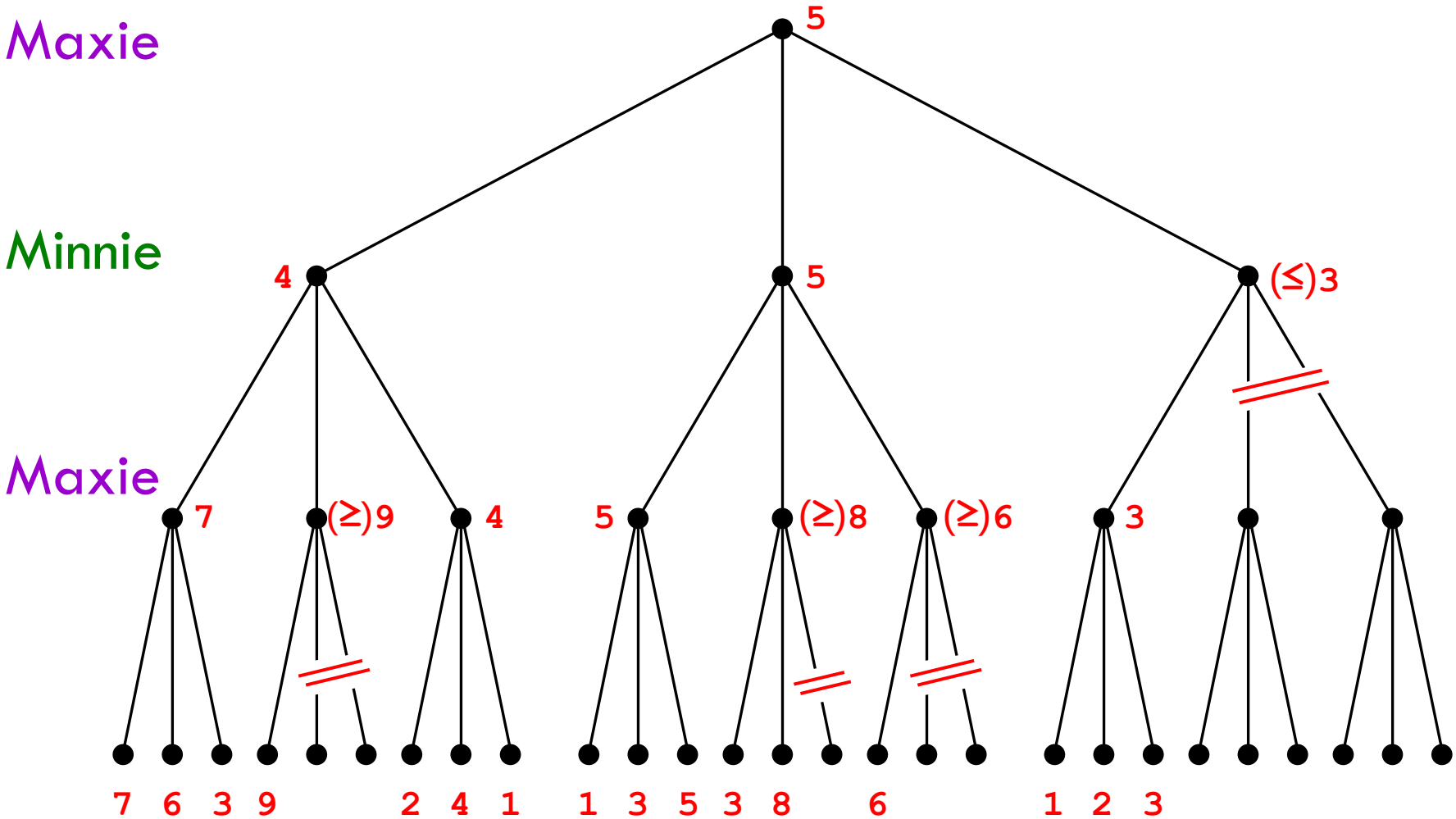

Maxie
Minnie
Maxie
(5, +∞)
4
5
(5, +∞)
7
(≥)9
4
5
(≥)8
(≥)6
(5, +∞)
7 6 3 9
2 4 1
1 3 5 3 8
6
1 2


Maxie
Minnie
Maxie
(5, +∞)
4
5
(5, +∞)
7
(≥)9
4
5
(≥)8
(≥)6
(5, +∞)
7 6 3 9
2 4 1
1 3 5 3 8
6
1 2 3

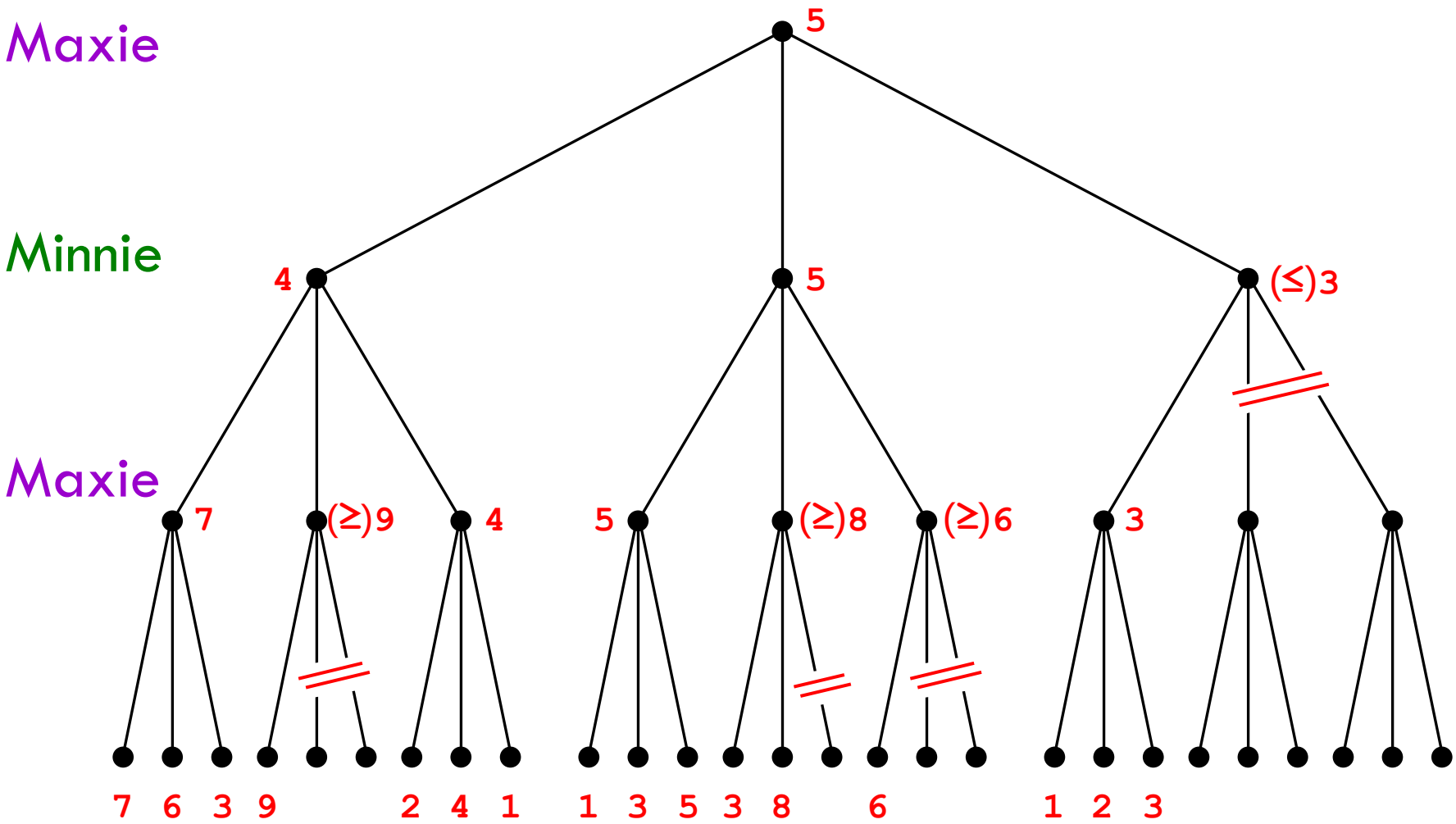

Maxie
Minnie
Maxie
(5, +∞)
(5, +∞)
4
5
7
(≥)9
4
5
(≥)8
(≥)6
3
7 6 3 9
2 4 1
1 3 5 3 8
6
1 2 3



The “deep pruning” on the right occurs because at a **Minnie** level $v \leq 3 < 5 = \alpha$.

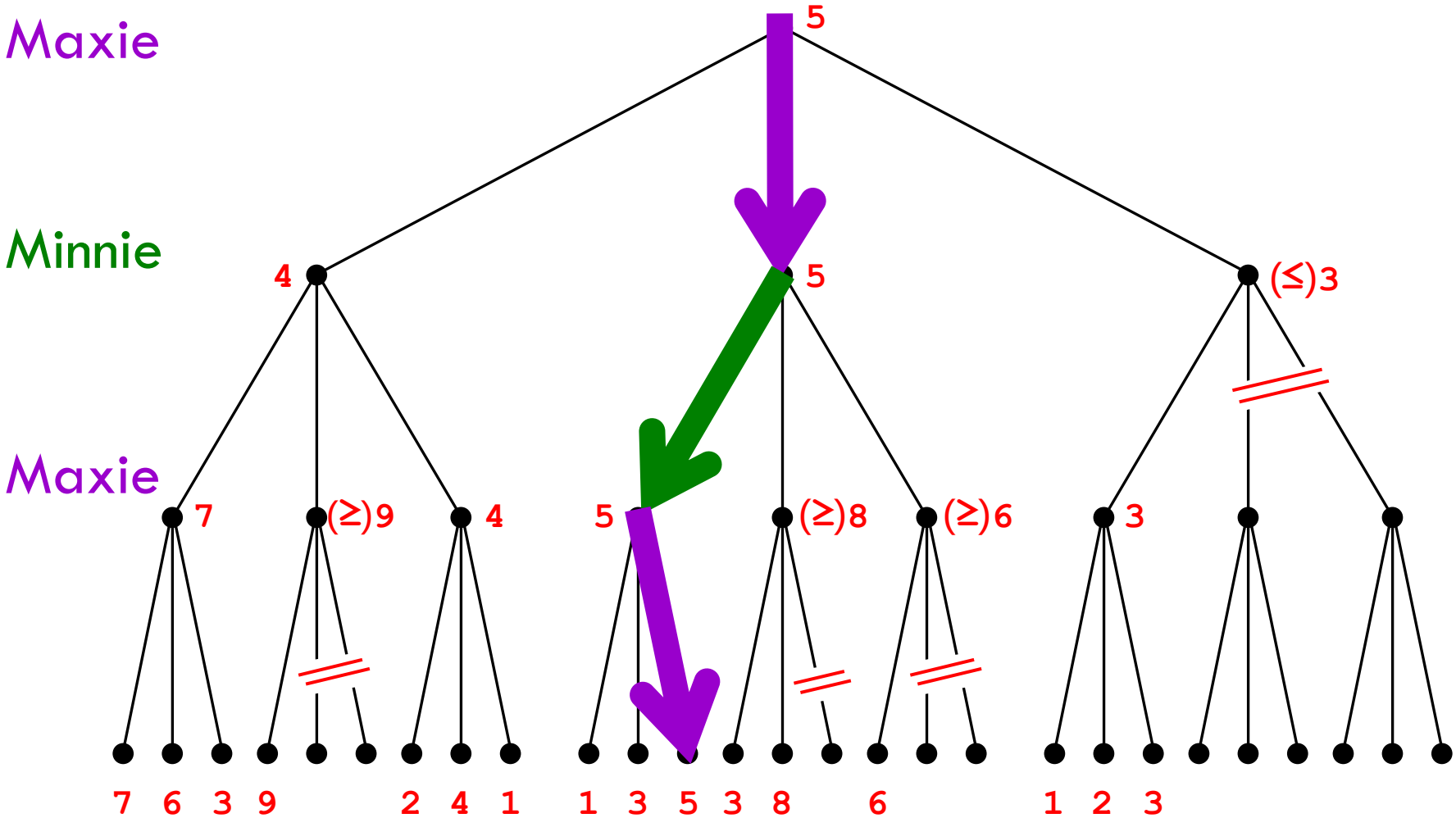

Maxie
Minnie
Maxie
5
4
5
(≤)3
7
(≥)9
4
5
(≥)8
(≥)6
3
7 6 3 9
2 4 1
1 3 5 3 8
6
1 2 3

Alpha-Beta saved 11 of 27 node evaluations



Alpha-Beta saved 11 of 27 node evaluations

(Optimal game play indicated by arrows.)



Functorize AlphaBeta Player

```
functor AlphaBeta (Settings : SETTINGS) : PLAYER =
struct
    structure Game = Settings.Game
```

Much as before, with some new helper types and functions.

```
    datatype bound = NEGINF | Bound of Game.est | POSINF
    type alphabeta = bound * bound
```

Remember: `(alpha,beta)` describes *knowledge* that a search has.

(EXAMPLE: `POSINF` and `Definitely(Winner(Maxie))` are different.
`POSINF` means the search does not yet know how well `Minnie` can do.)

Combination of `(alpha,beta)` knowledge and move `est` guides search.

```
    datatype orderAB = BELOW | INTERIOR | ABOVE

    (* compareAB : alphabeta -> Game.est -> orderAB *)
```

You will write the new search and evaluation code.

```
end
```

That is all.

Have a good Wednesday!

On Thursday, we will discuss writing lazy code.