COMP 4752
Computational Intelligence

Lecture 12

Two-Player Games

Mini-Max / Alpha-Beta Search

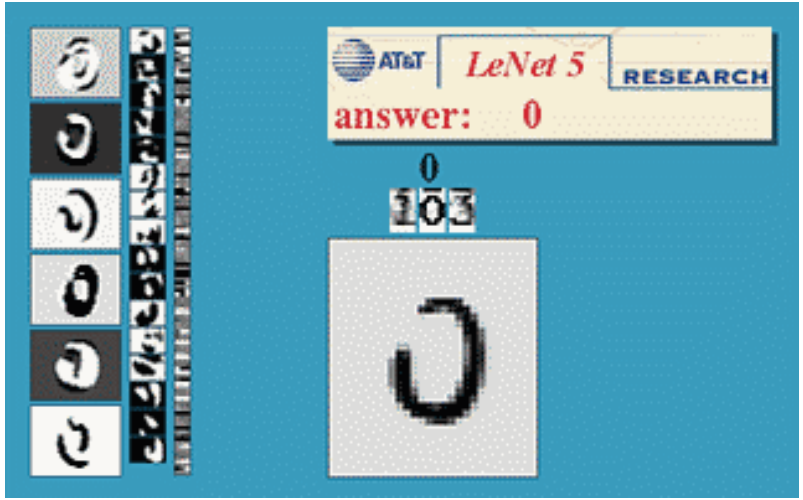
Multi-Player Games

- 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

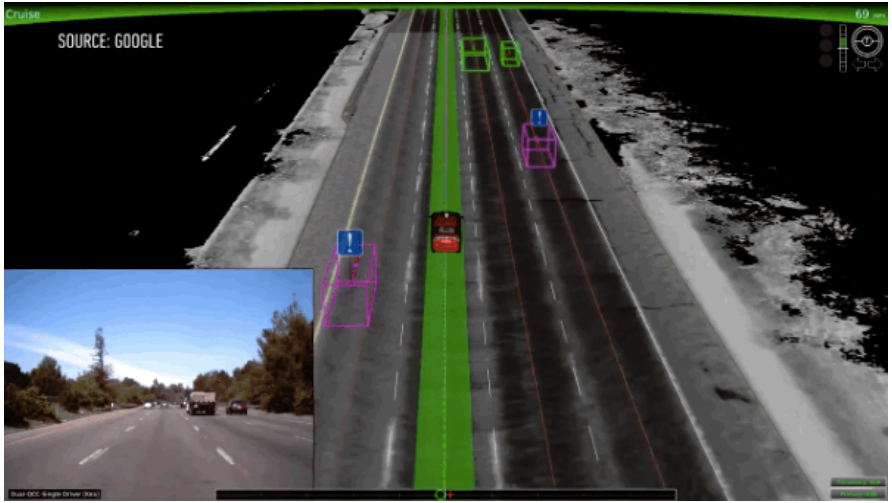
Deterministic vs. Stochastic

Episodic vs. Sequential



Dynamic vs. Static

Discrete vs. Continuous



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

Games in this Course

- Two-Player
- Zero Sum
- Alternating Move
- Fully Observable (Perfect Information)
- Deterministic
- Discrete

The Chess-Playing Computer

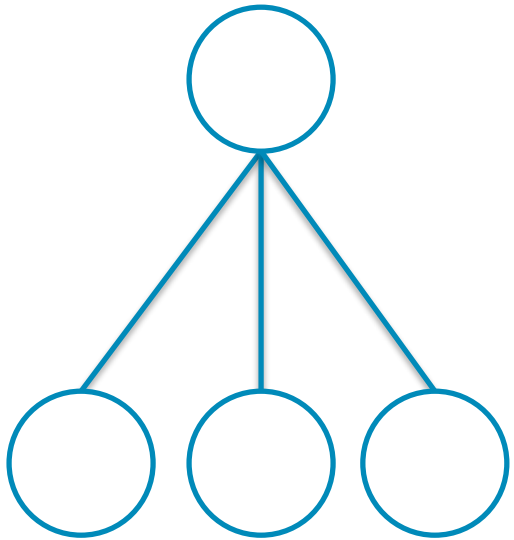
Game-Playing Computer

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

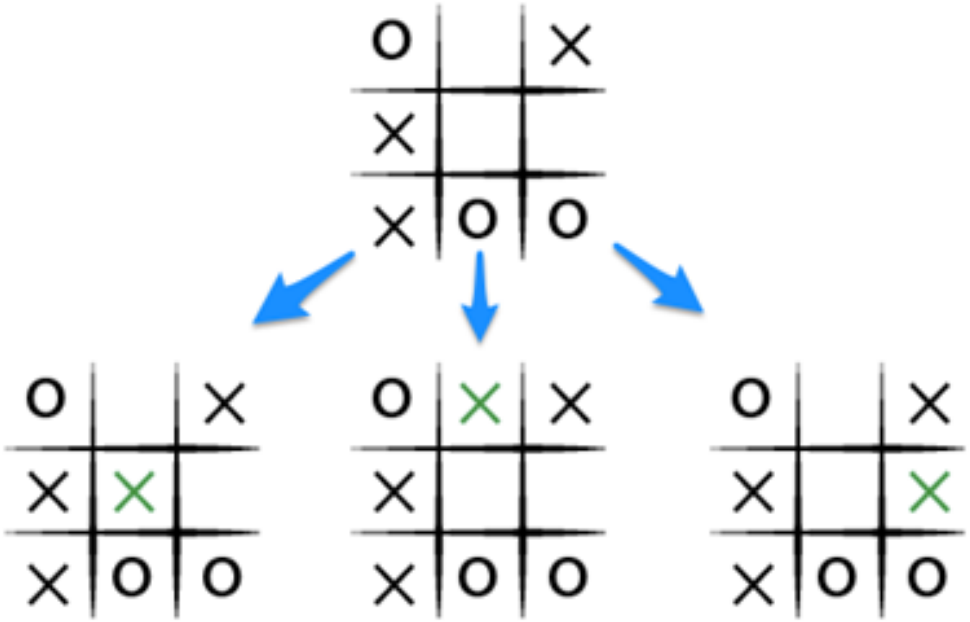
1. Analysis / Strategy / Tactics
2. Thousands of If-Then Statements
3. Look-Ahead and Evaluate

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



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. Look-Ahead and Evaluate
4. Search the Entire Game Tree

How big is the game tree?

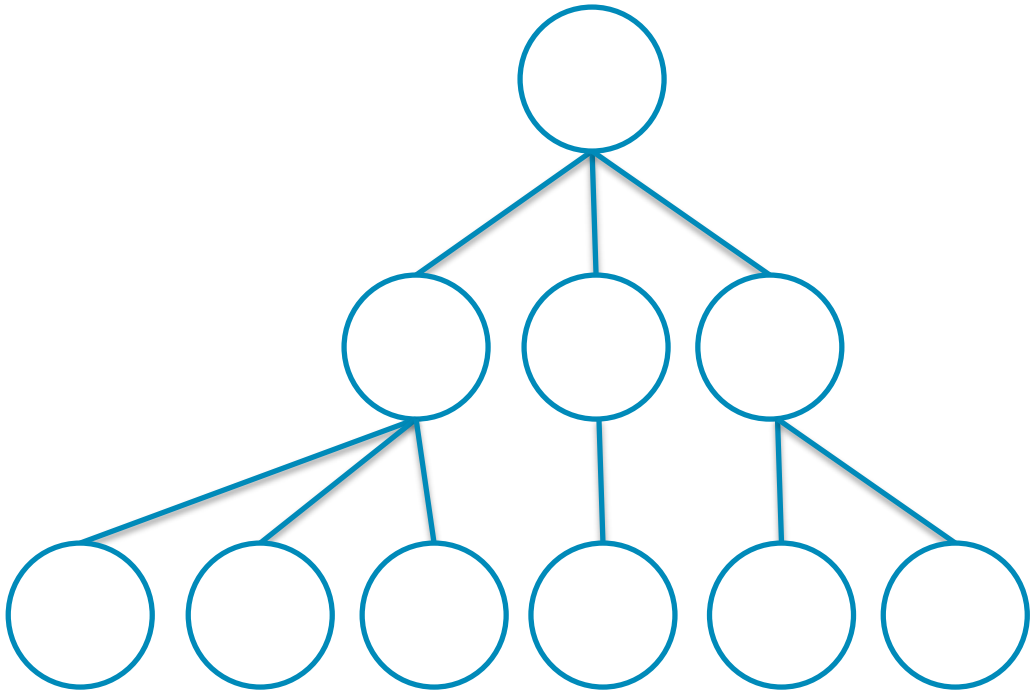
- Some number of actions at each state
  - b = “Branching Factor”
- Game ends after some number of moves
  - d = Search Depth
- Game tree = b^d
- Chess = 10^{120}

Game-Playing Computer

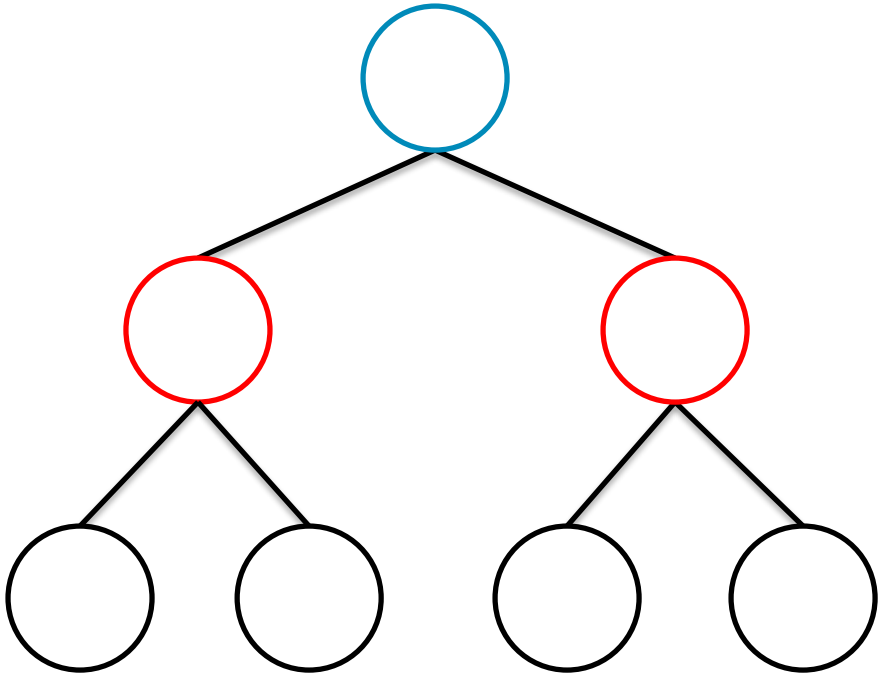
- How can we design an algorithm to play a two-player alternating move game?

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

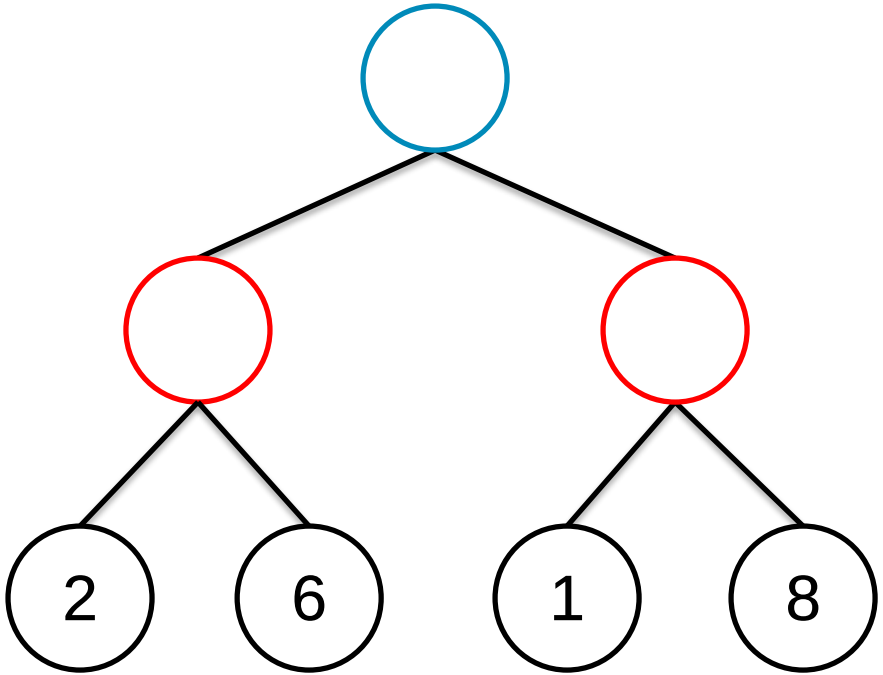
Look Ahead as Far as Possible



Competing Players



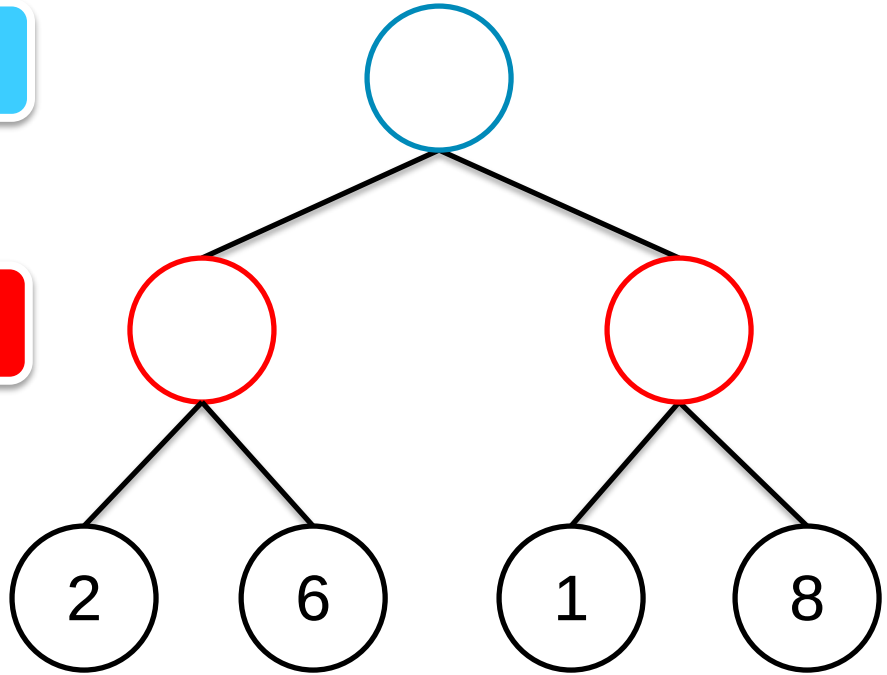
Competing Players



Competing Players

Max Player

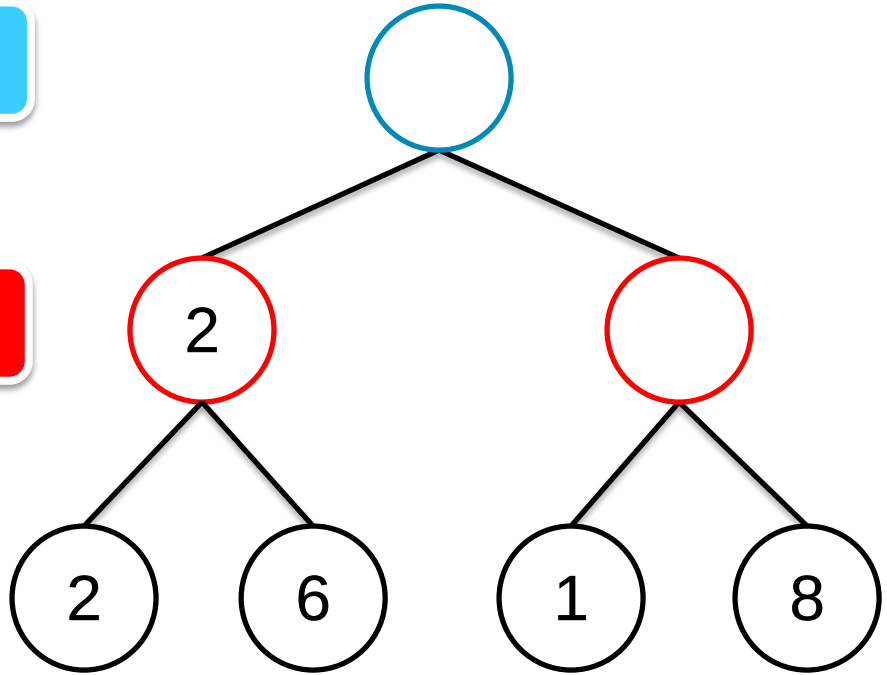
Min Player



Competing Players

Max Player

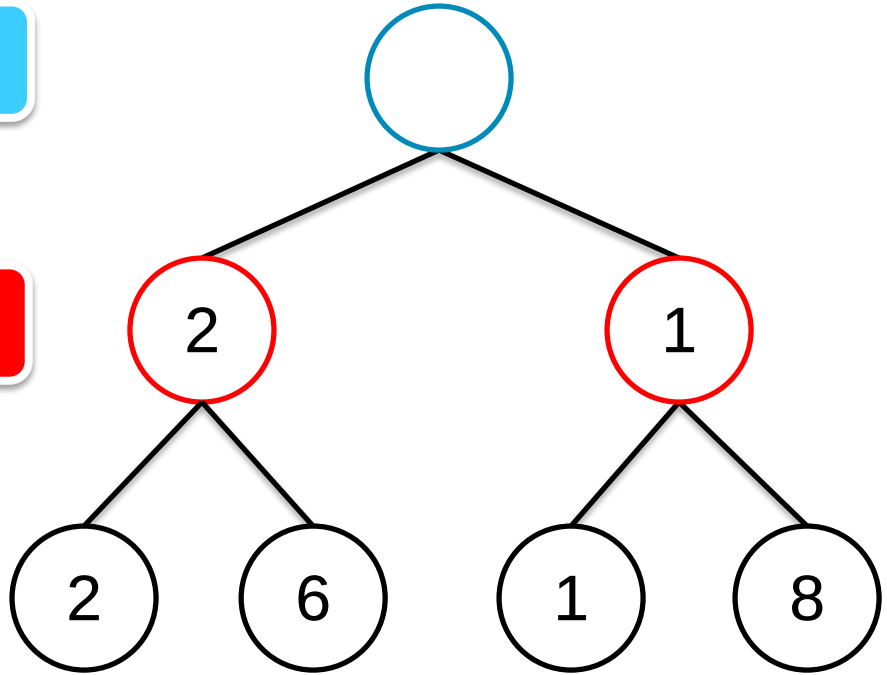
Min Player



Competing Players

Max Player

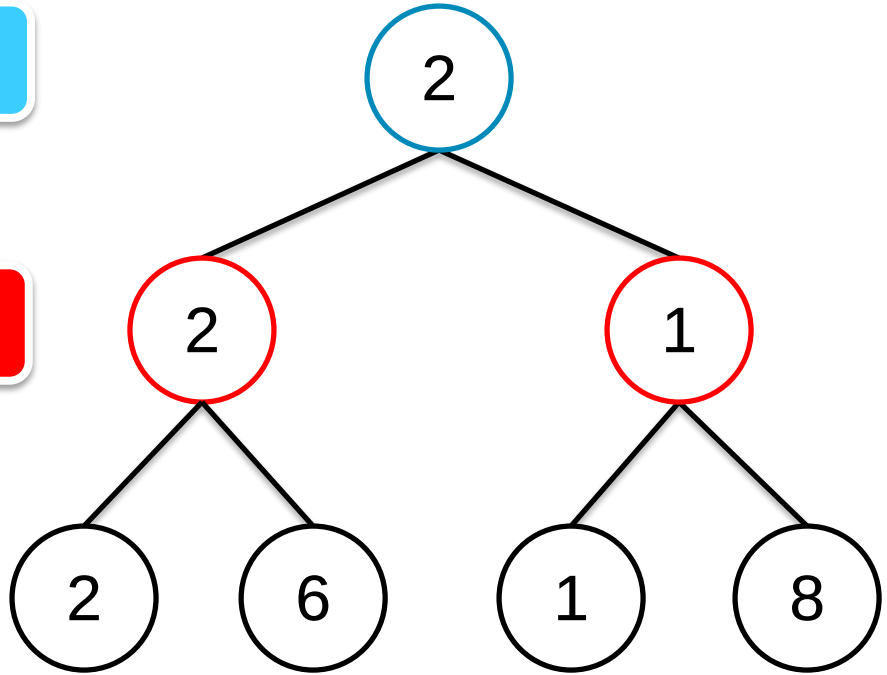
Min Player



Competing Players

Max Player

Min Player



Minimax Algorithm

```
1.  Function MiniMax(node, depth, maxPlayer)
2.          if (depth > maxDepth or node.terminal)
3.              return eval(node)
4.          if (node.playerToMove == maxPlayer)
5.              bestValue = -infinity
6.              for each child of node
7.                  value = MiniMax(child, depth+1, maxPlayer)
8.                  bestValue = max(bestValue, value)
9.          else
10.             bestValue = infinity
11.             for each child of node
12.                 value = MiniMax(child, depth+1, maxPlayer)
13.                 bestValue = min(bestValue, value)
14. Initial Call: MiniMax(rootNode, 0, maxPlayer)
```

Negamax Algorithm

```
Function NegaMax(node, depth, player)
        if (depth > maxDepth or node.terminal)
            return eval(node)
        bestValue = -infinity
        for each child of node
            value = -NegaMax(child, depth+1, opponent(player))
            bestValue = max(bestValue, value)
```

Initial Call: NegaMax(root, 0, 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

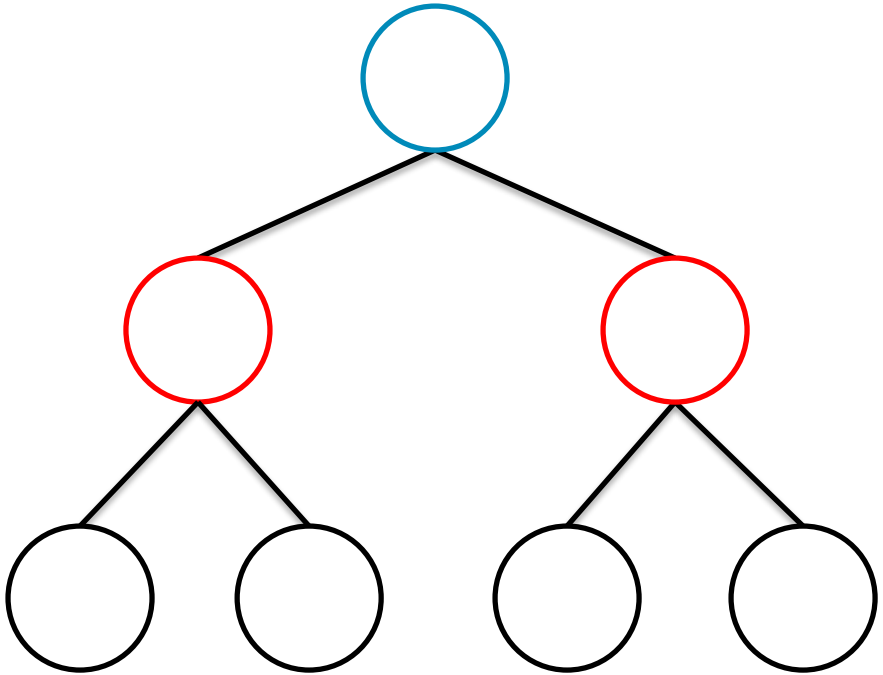
Nash Equilibrium (NE)

- 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

Max

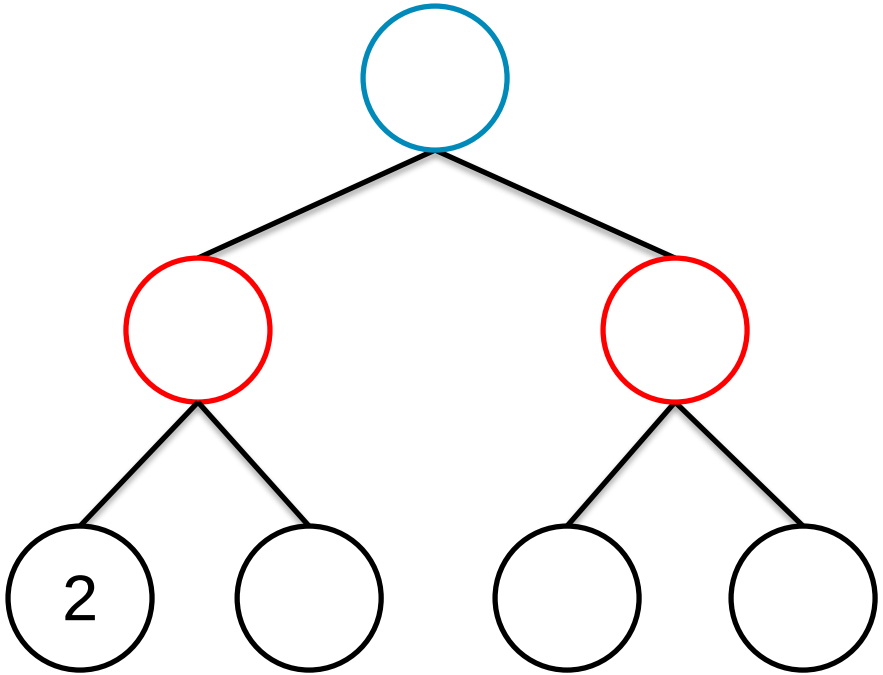
Min



Improving MiniMax

Max

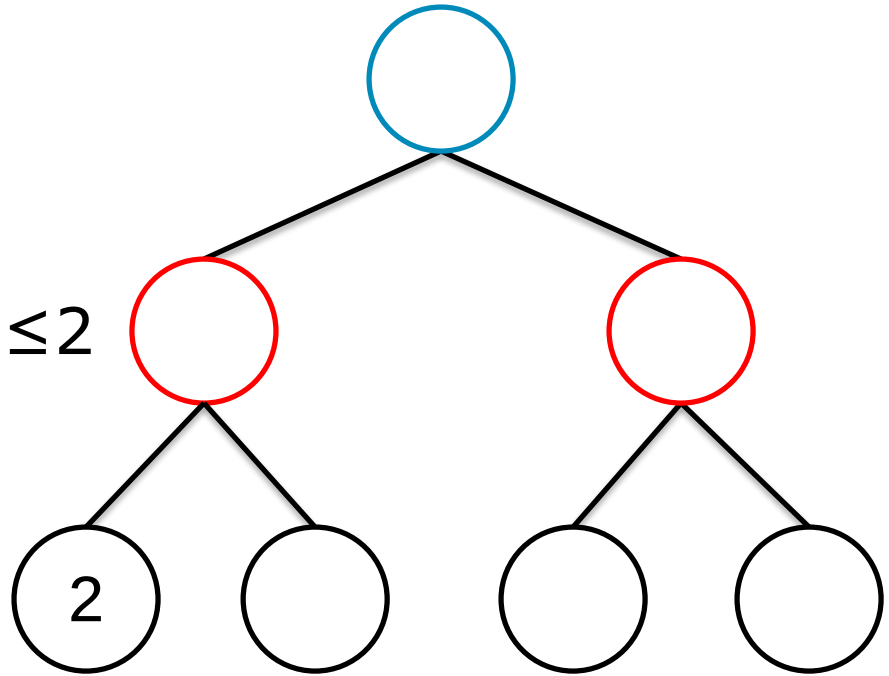
Min



Improving MiniMax

Max

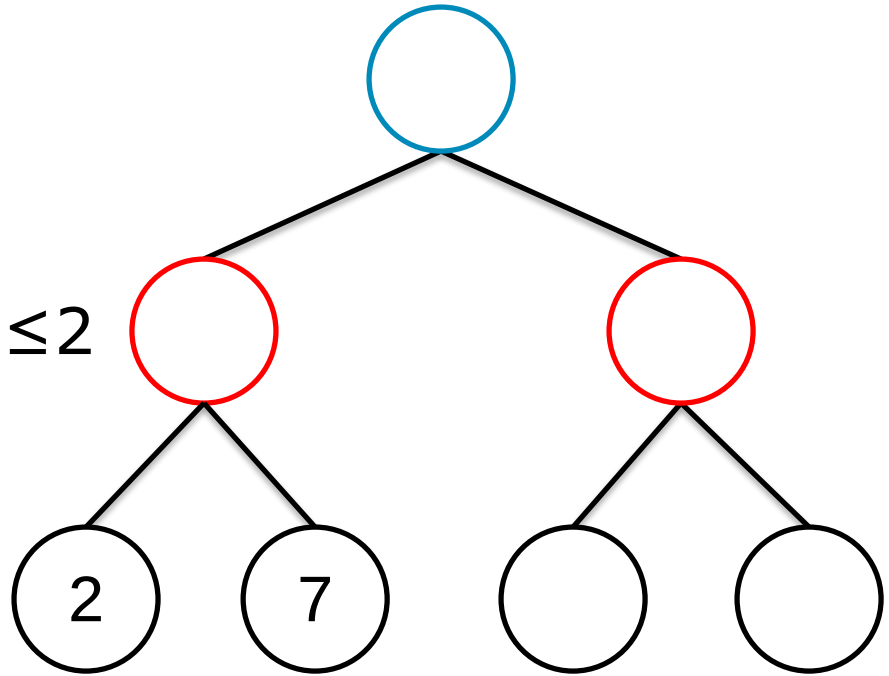
Min



Improving MiniMax

Max

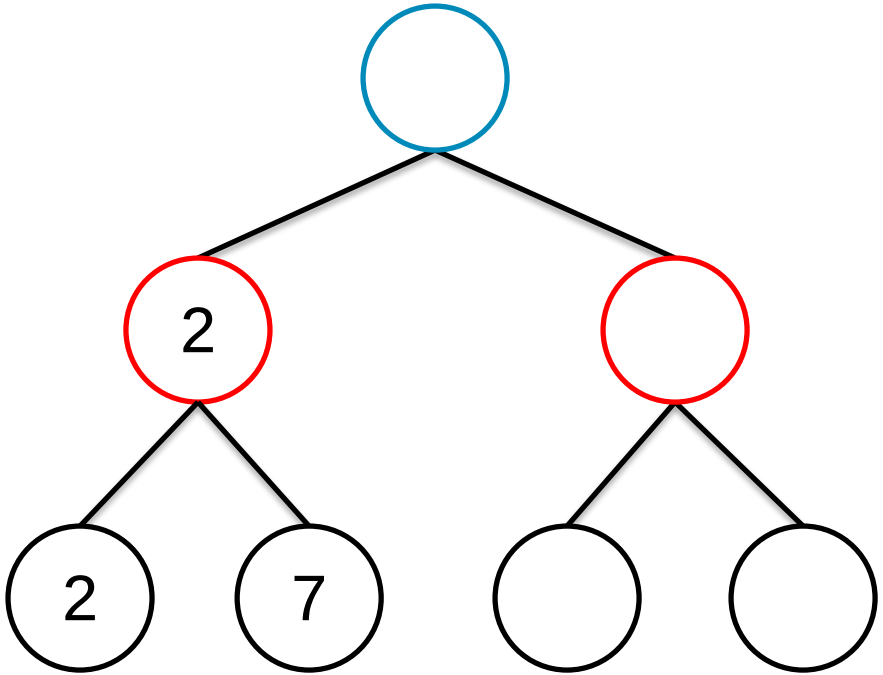
Min



Improving MiniMax

Max

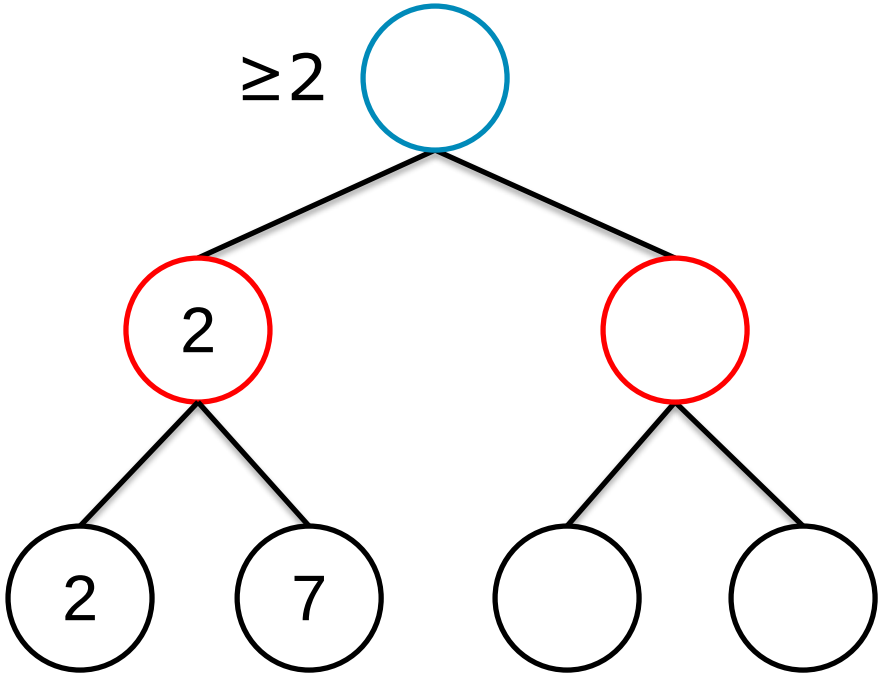
Min



Improving MiniMax

Max

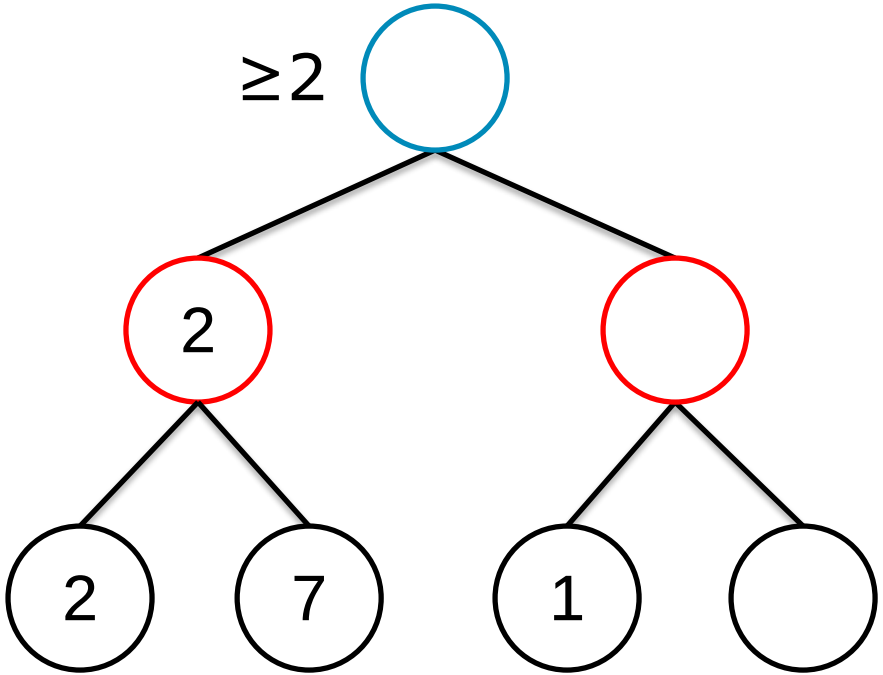
Min



Improving MiniMax

Max

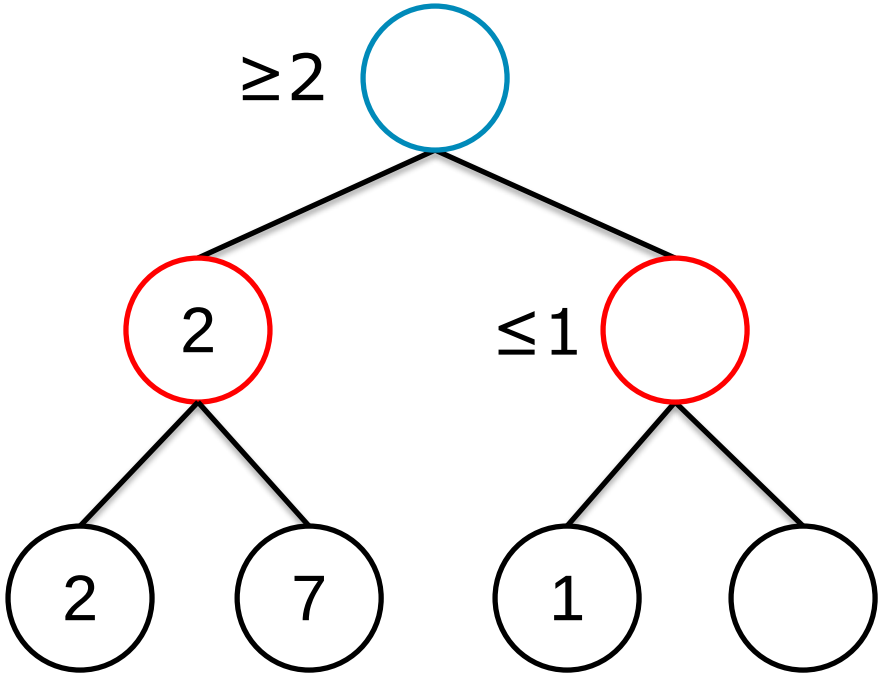
Min



Improving MiniMax

Max

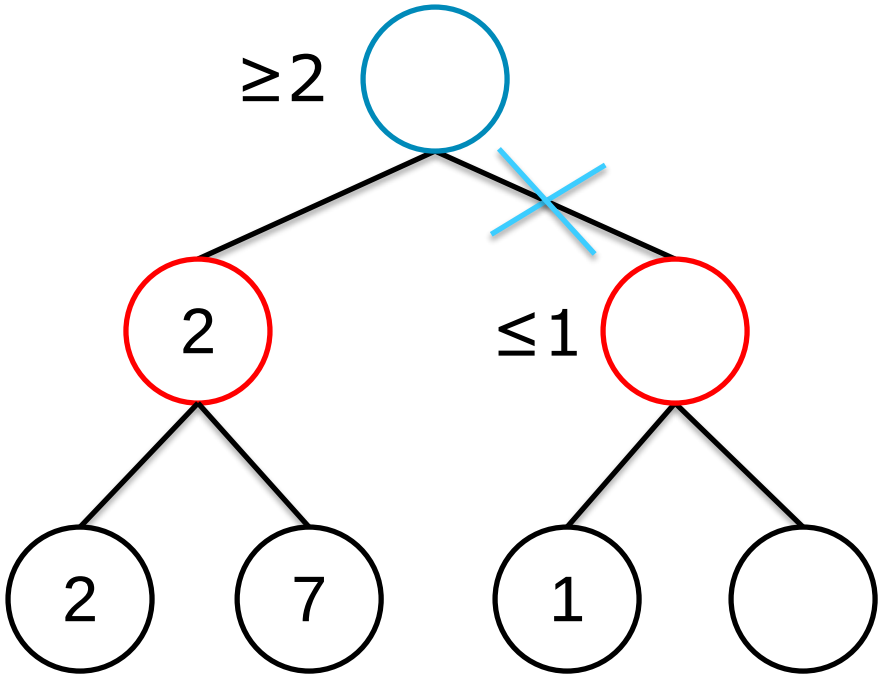
Min



Improving MiniMax

Max

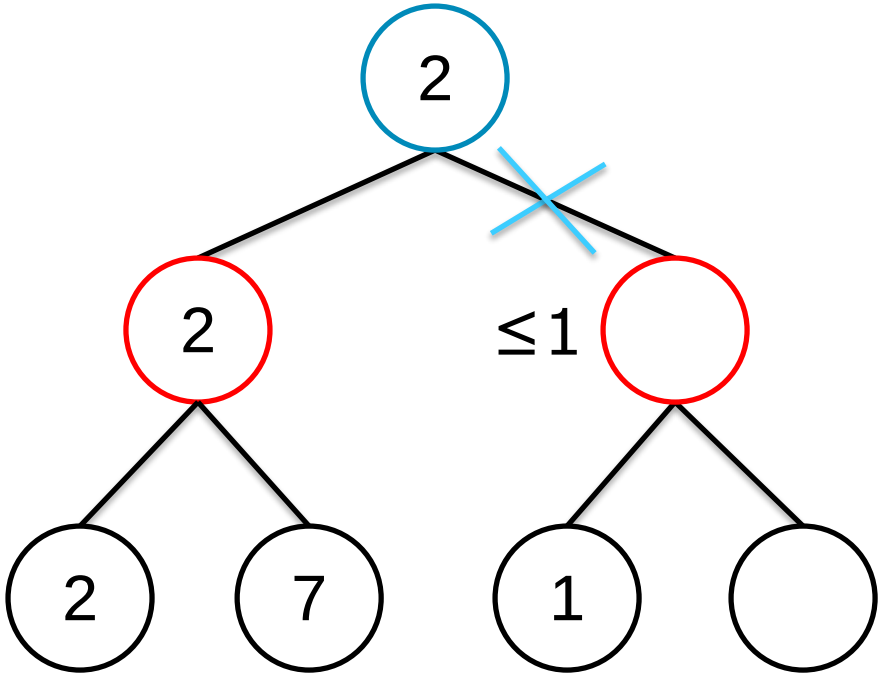
Min



Improving MiniMax

Max

Min

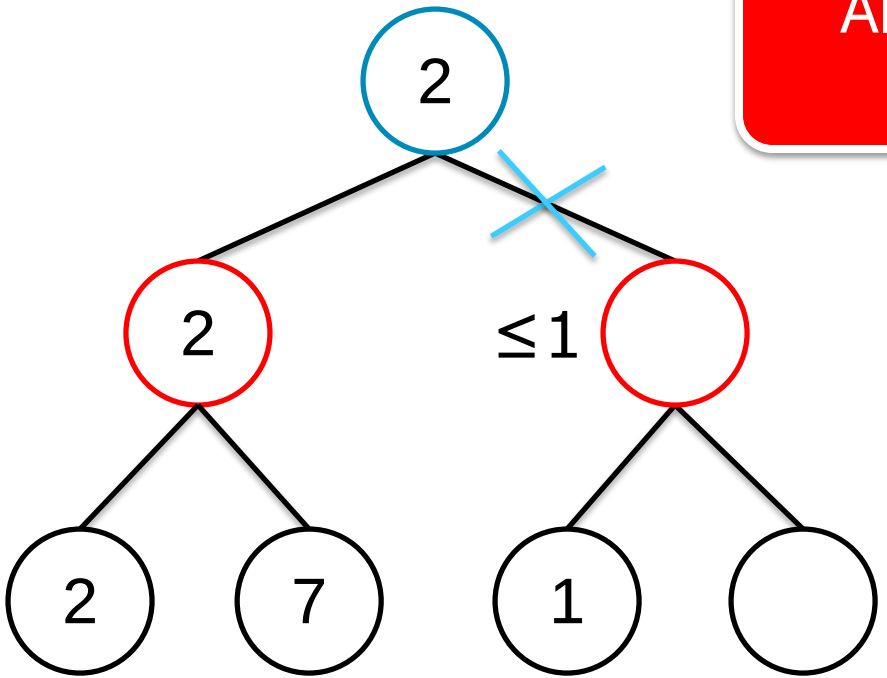


Improving MiniMax

Max

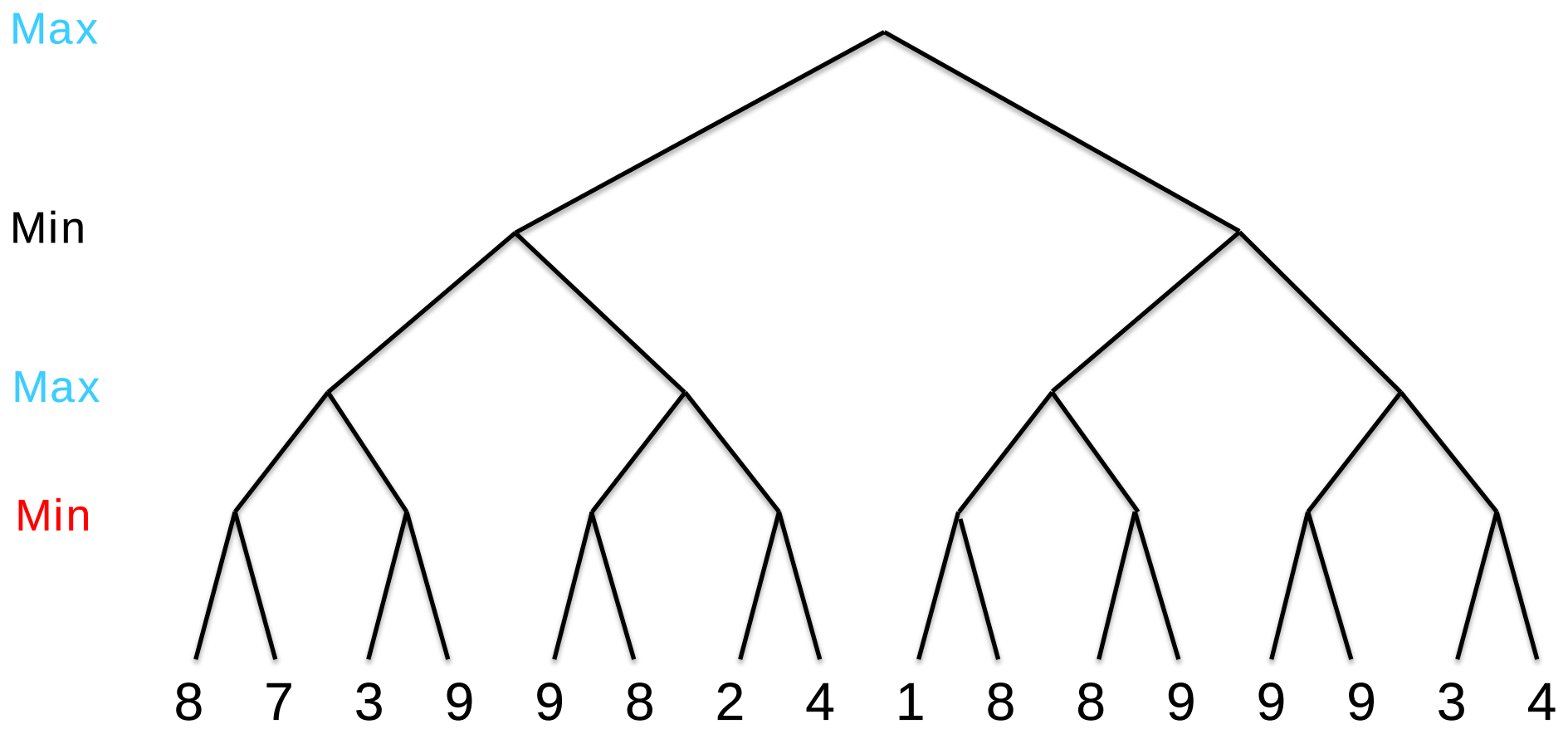
Min

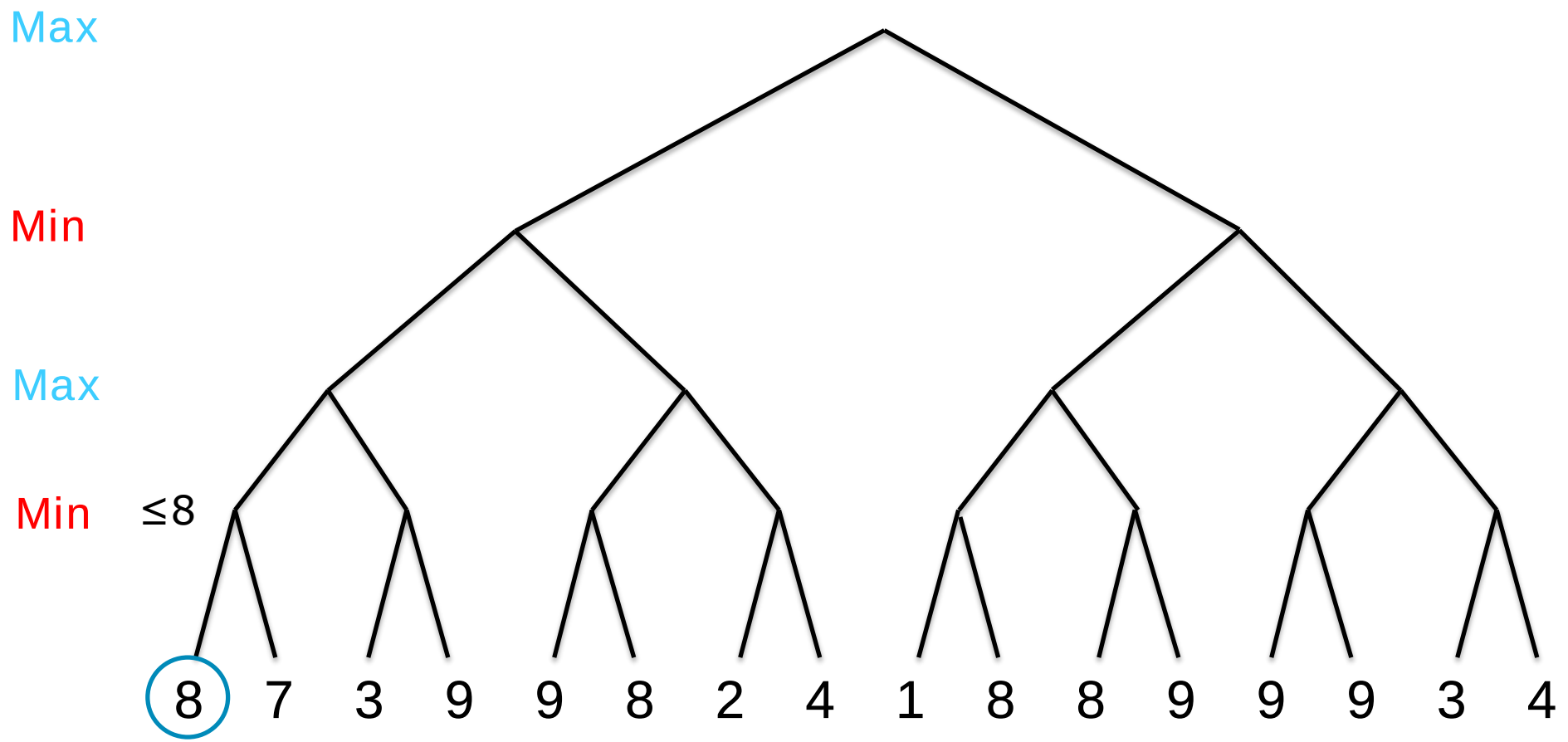
Alpha-Beta Pruning

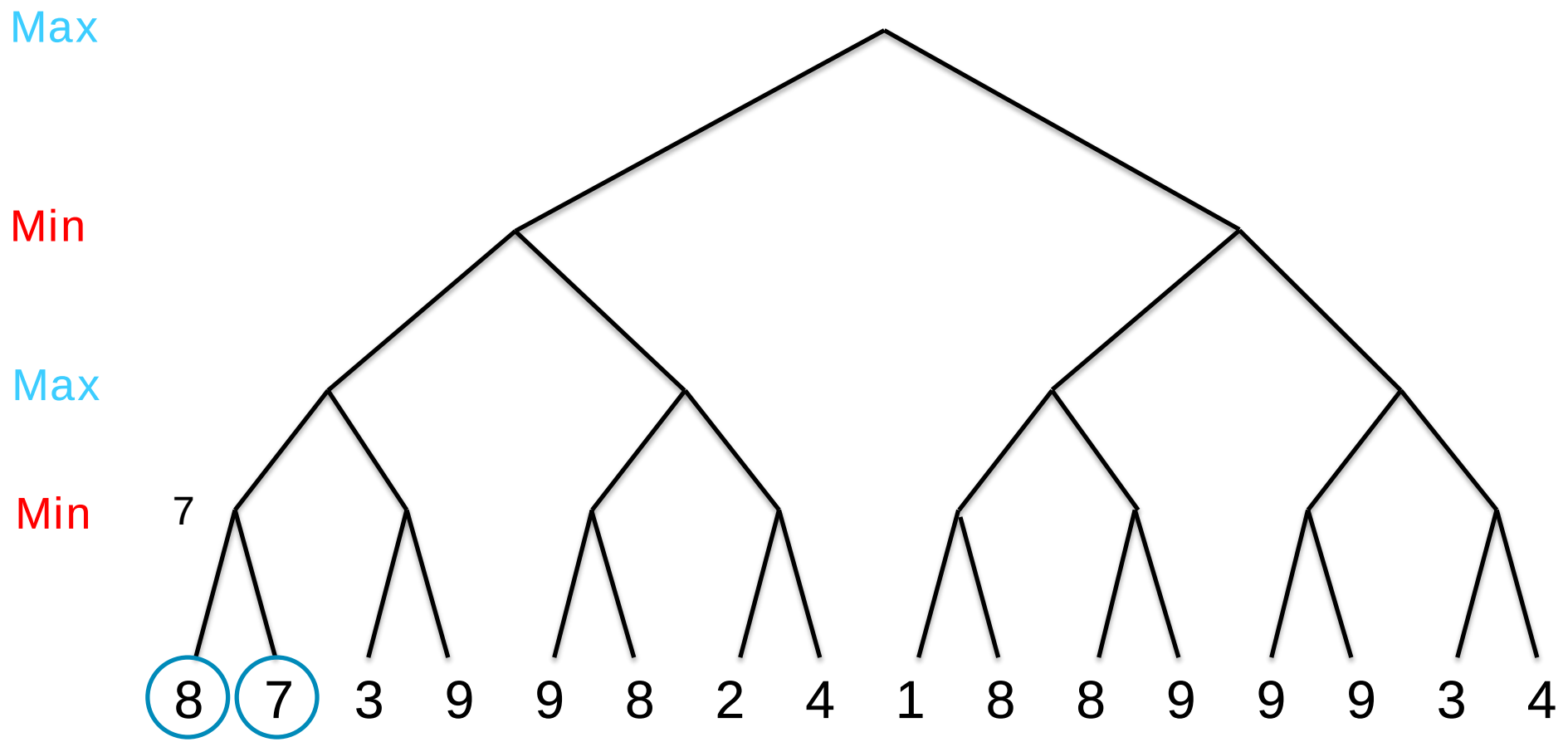


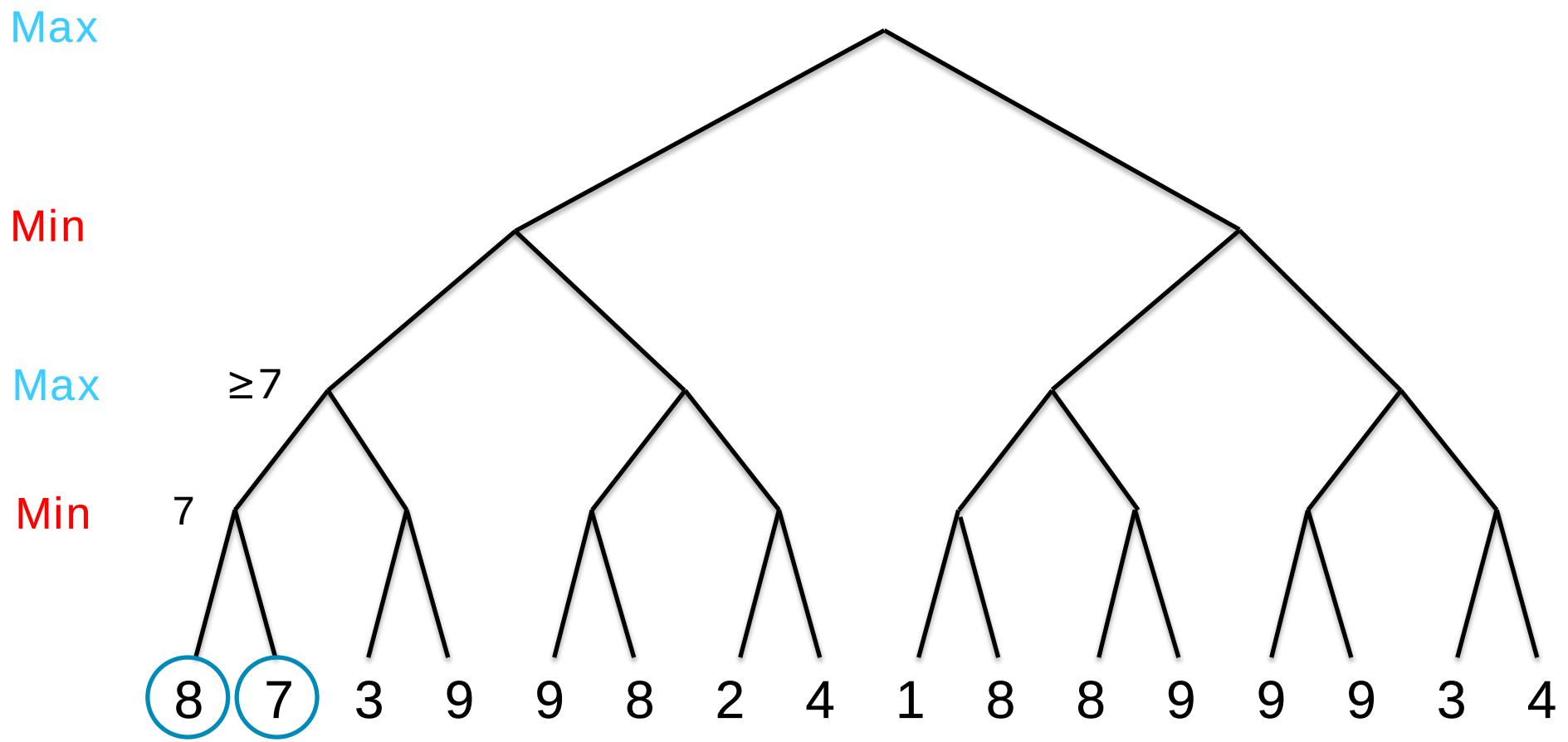
Alpha-Beta Pruning

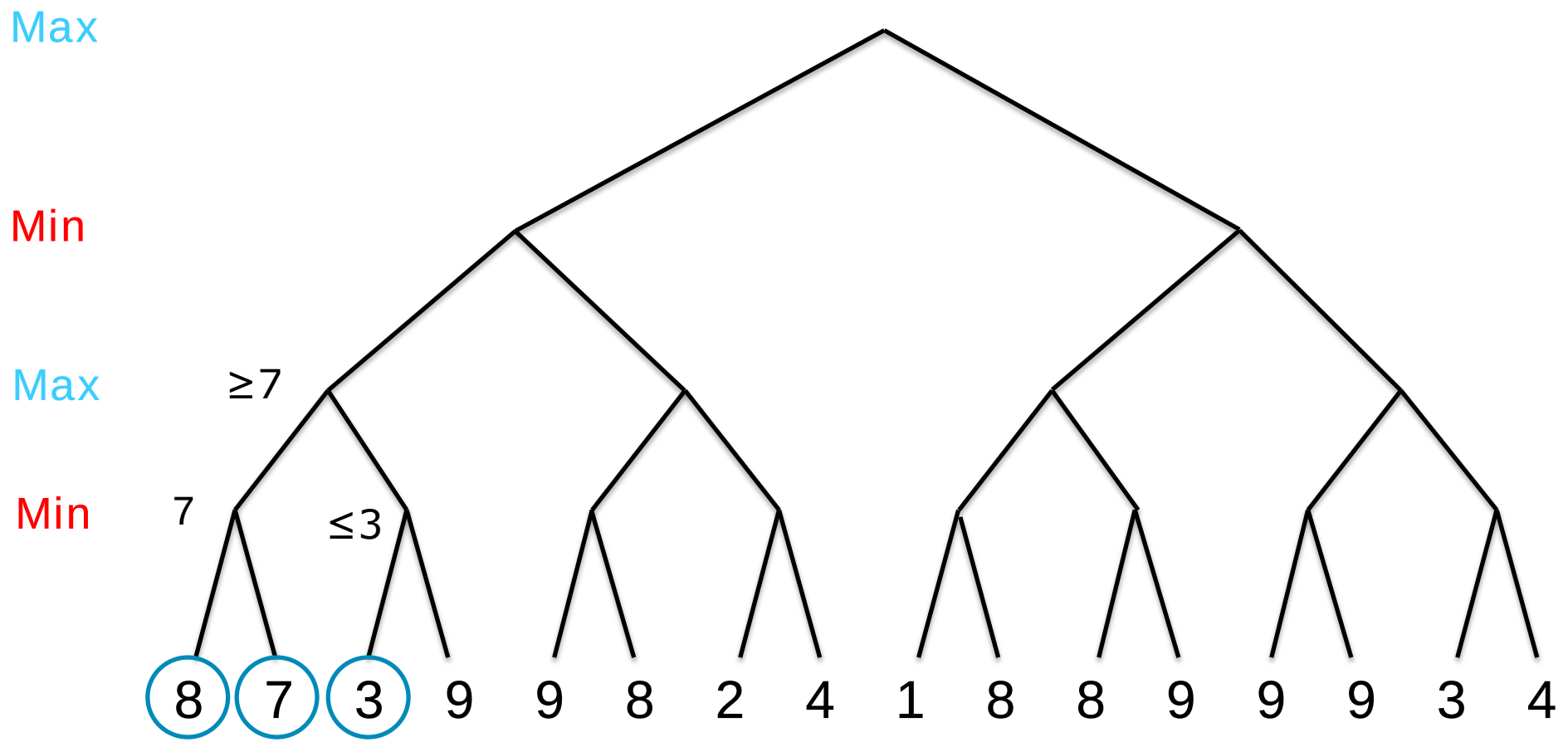
- Alpha-Beta is not a different algorithm than MiniMax, it is an optimization
- Maintains all of the properties of minimax but is strictly better
- Cutting off branches of the search tree yields exponential savings

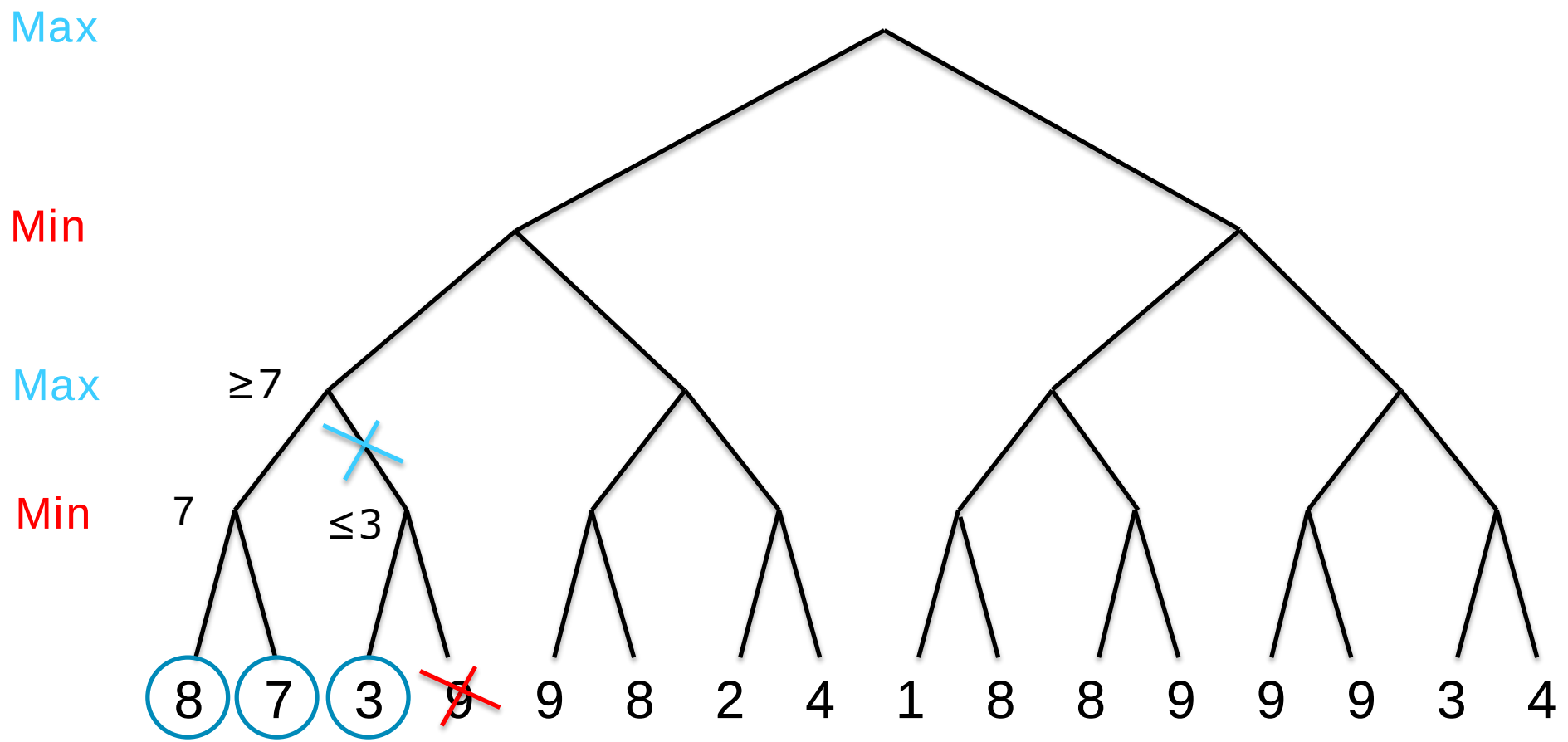

Max
Min
Max
Min
8 7 3 9 9 8 2 4 1 8 8 9 9 9 3 4

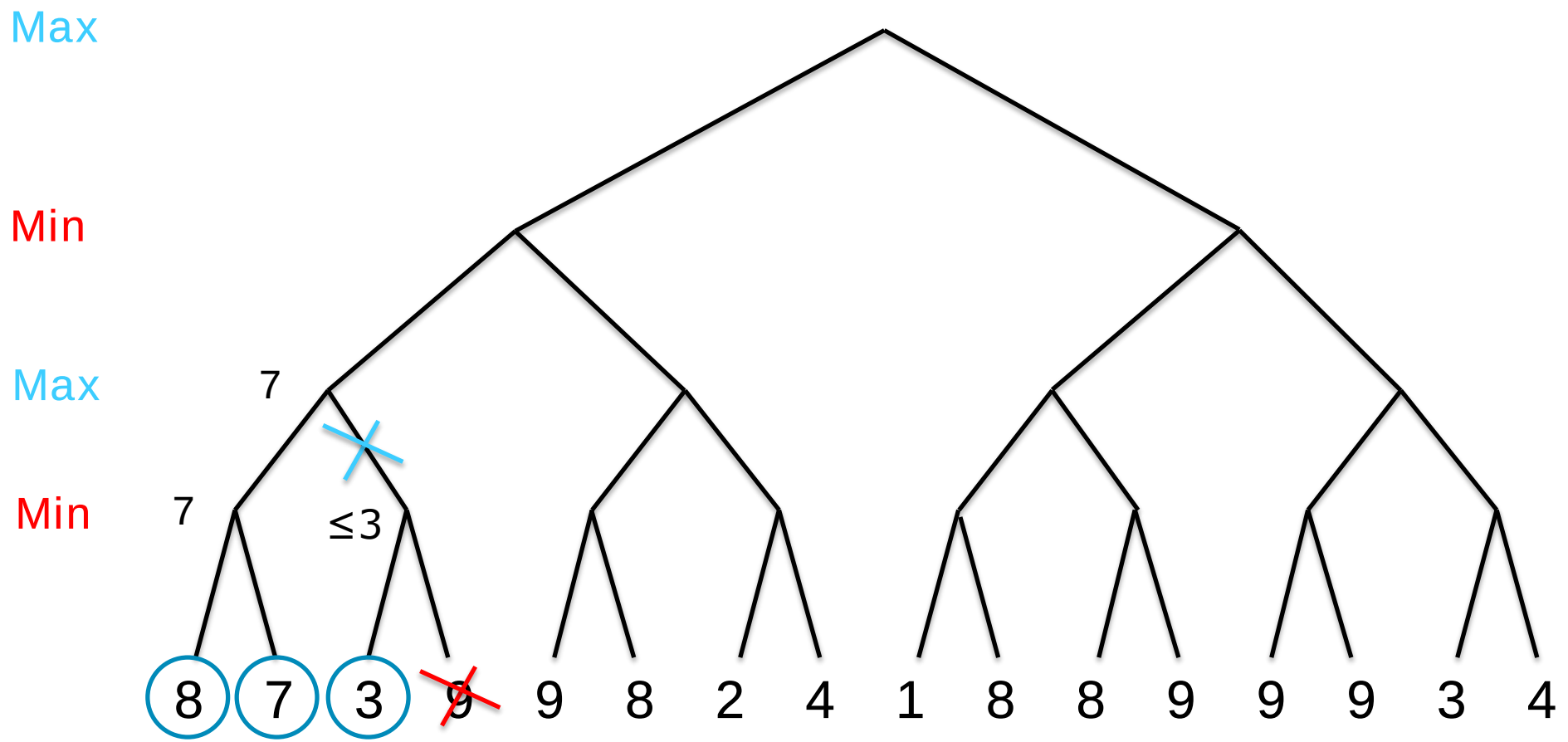

Max
Min
Max
Min
≤8
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4

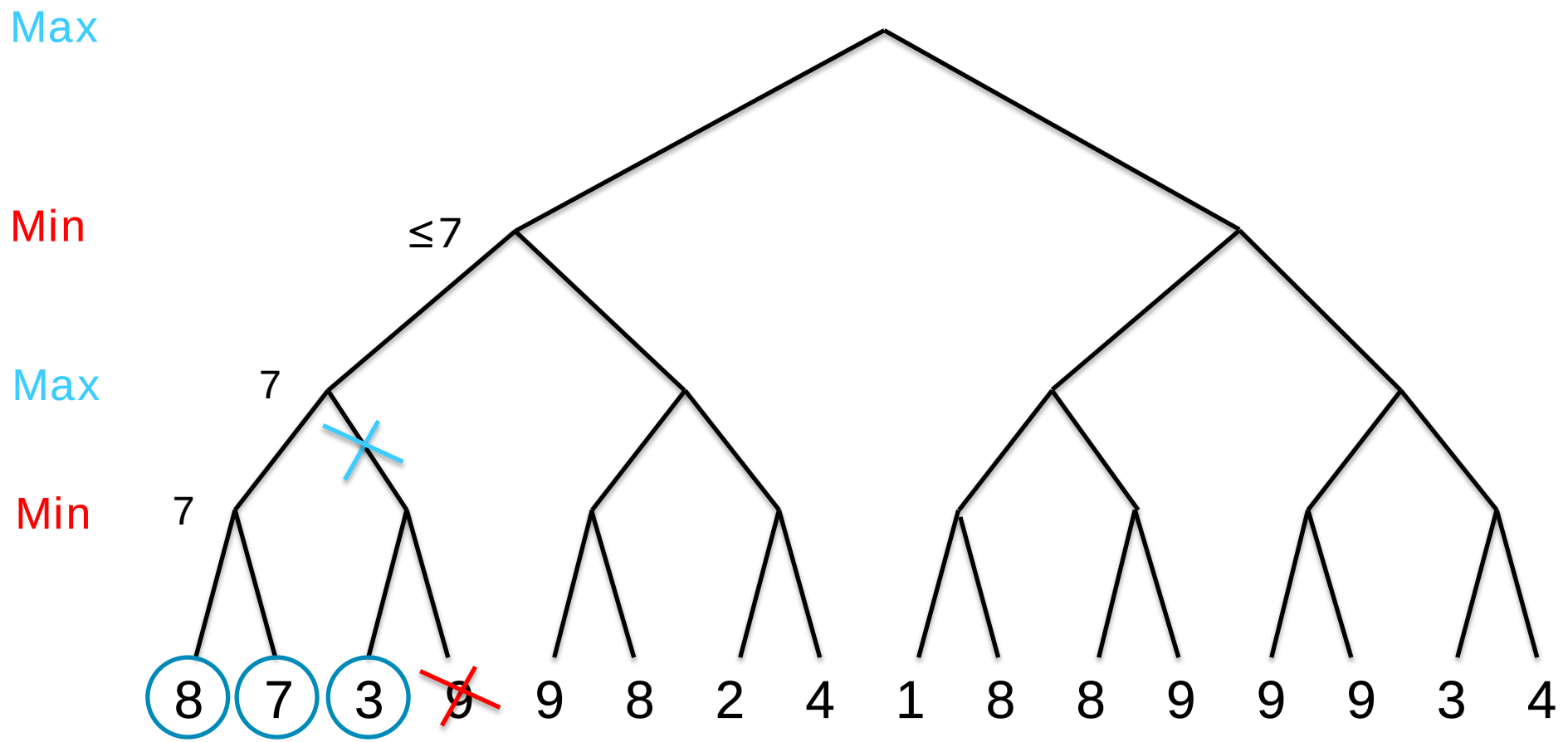

Max
Min
Max
Min
7
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4

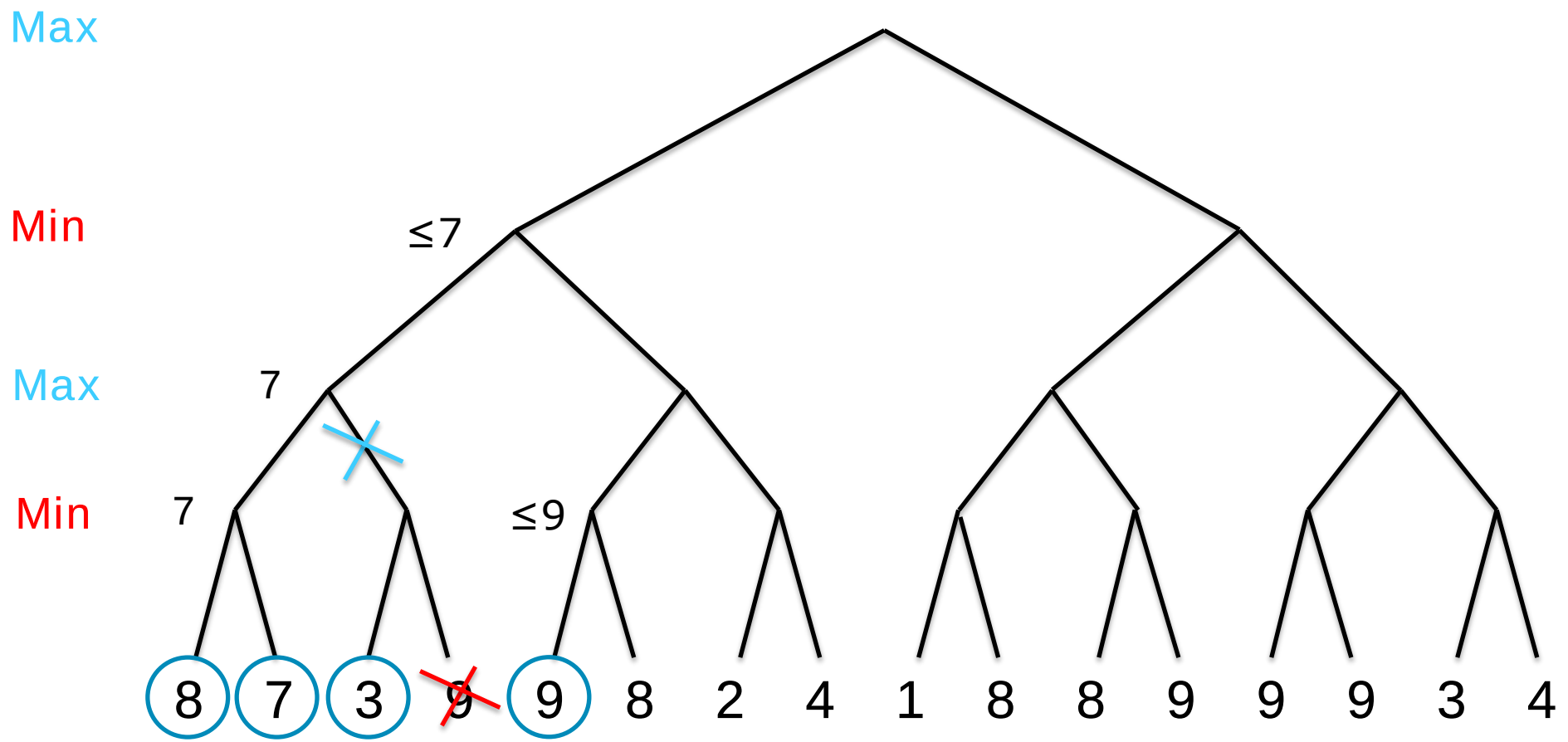

Max
Min
Max
Min
≥7
7
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4

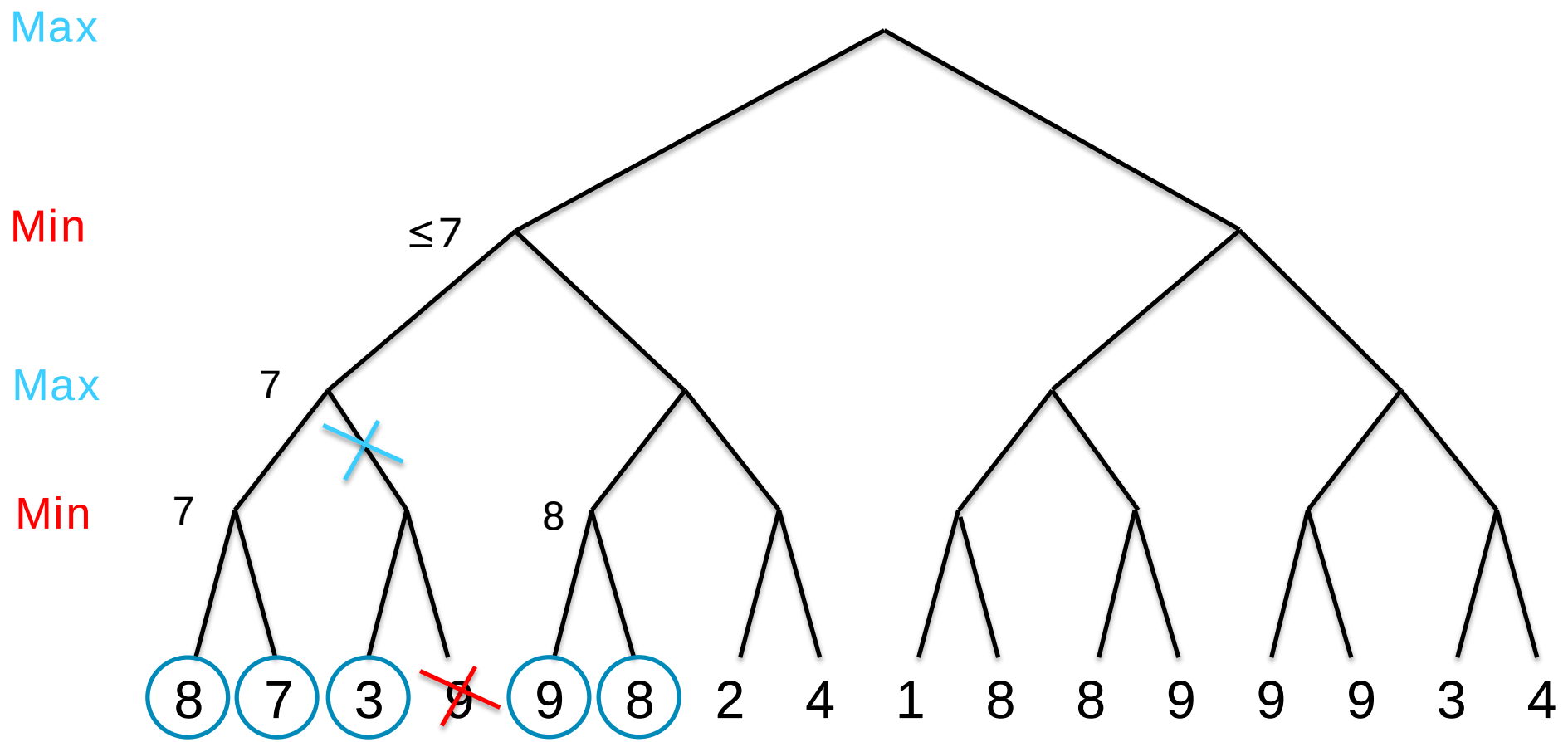

Max
Min
Max
Min
≥7
7
≤3
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4

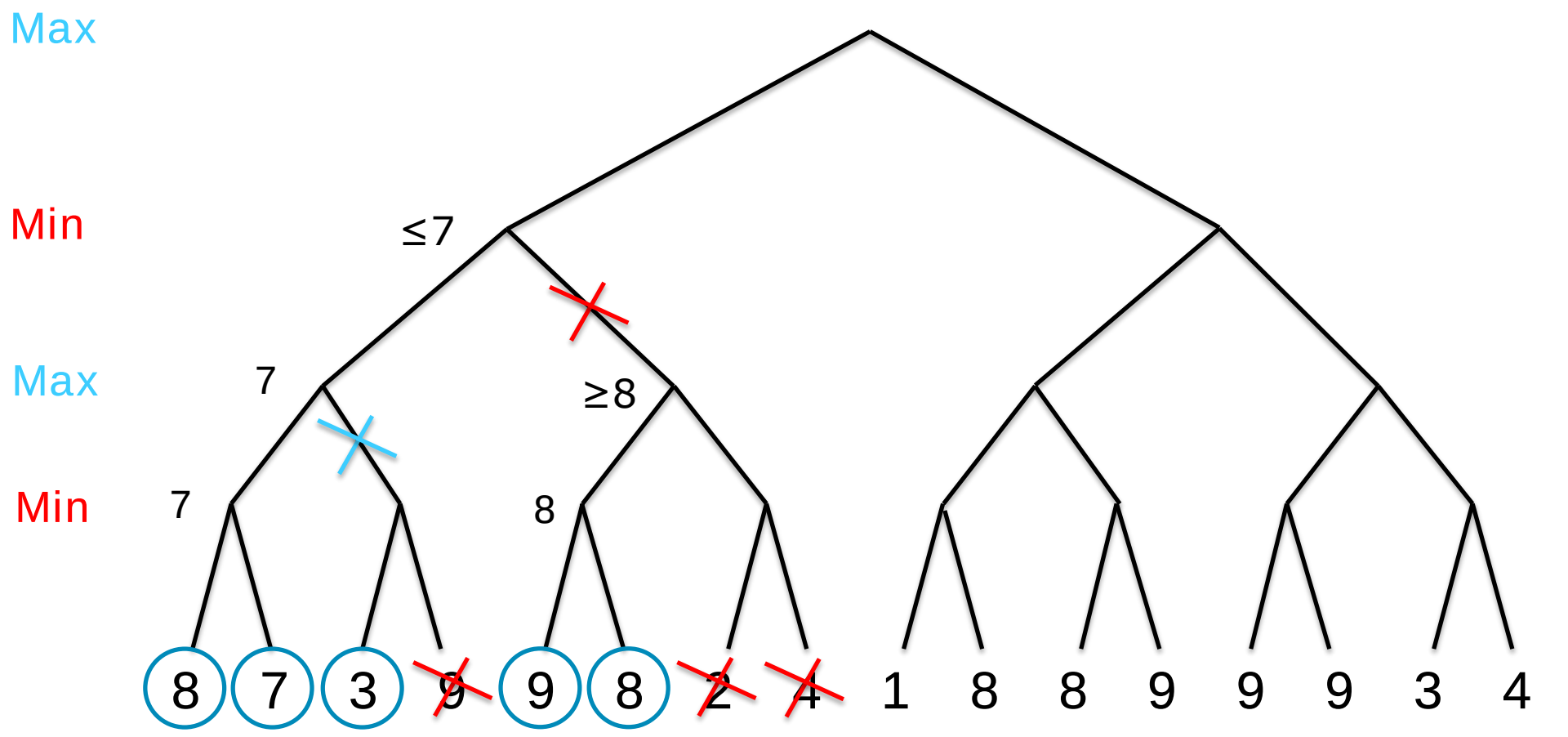

Max
Min
Max
Min
≥7
7
≤3
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4

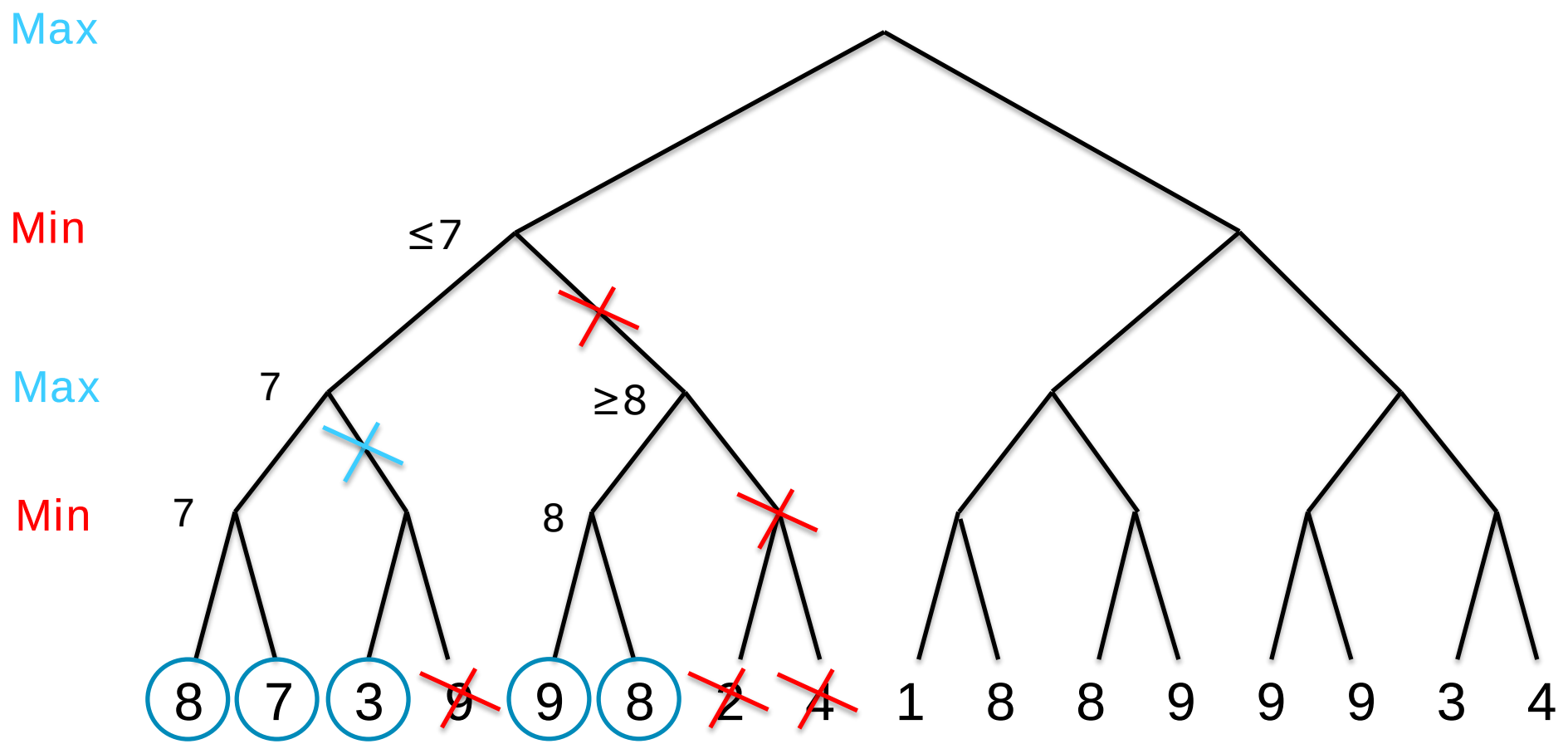

Max
Min
Max
Min
7
7
≤3
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4

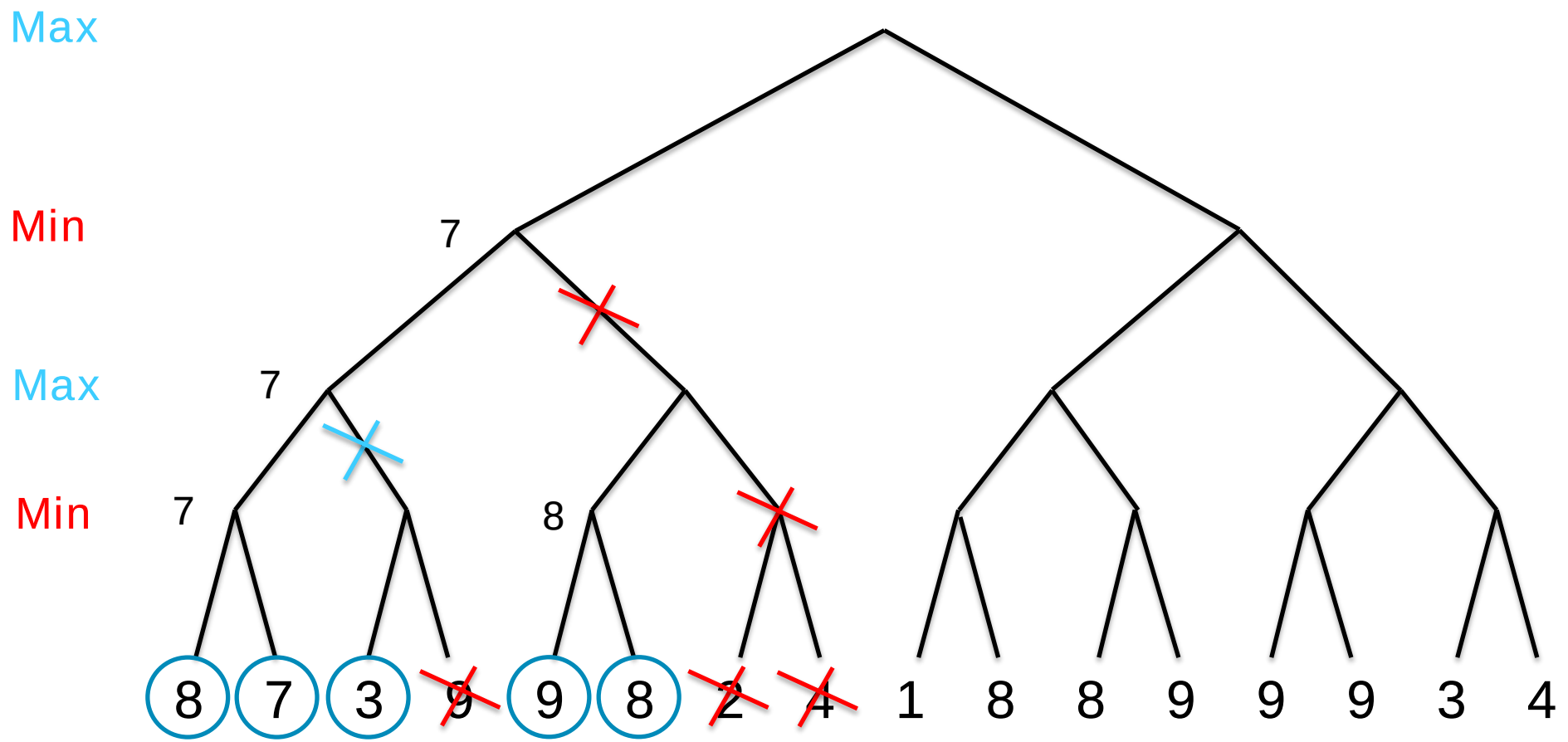

Max
Min
Max
Min
≤7
7
7
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4

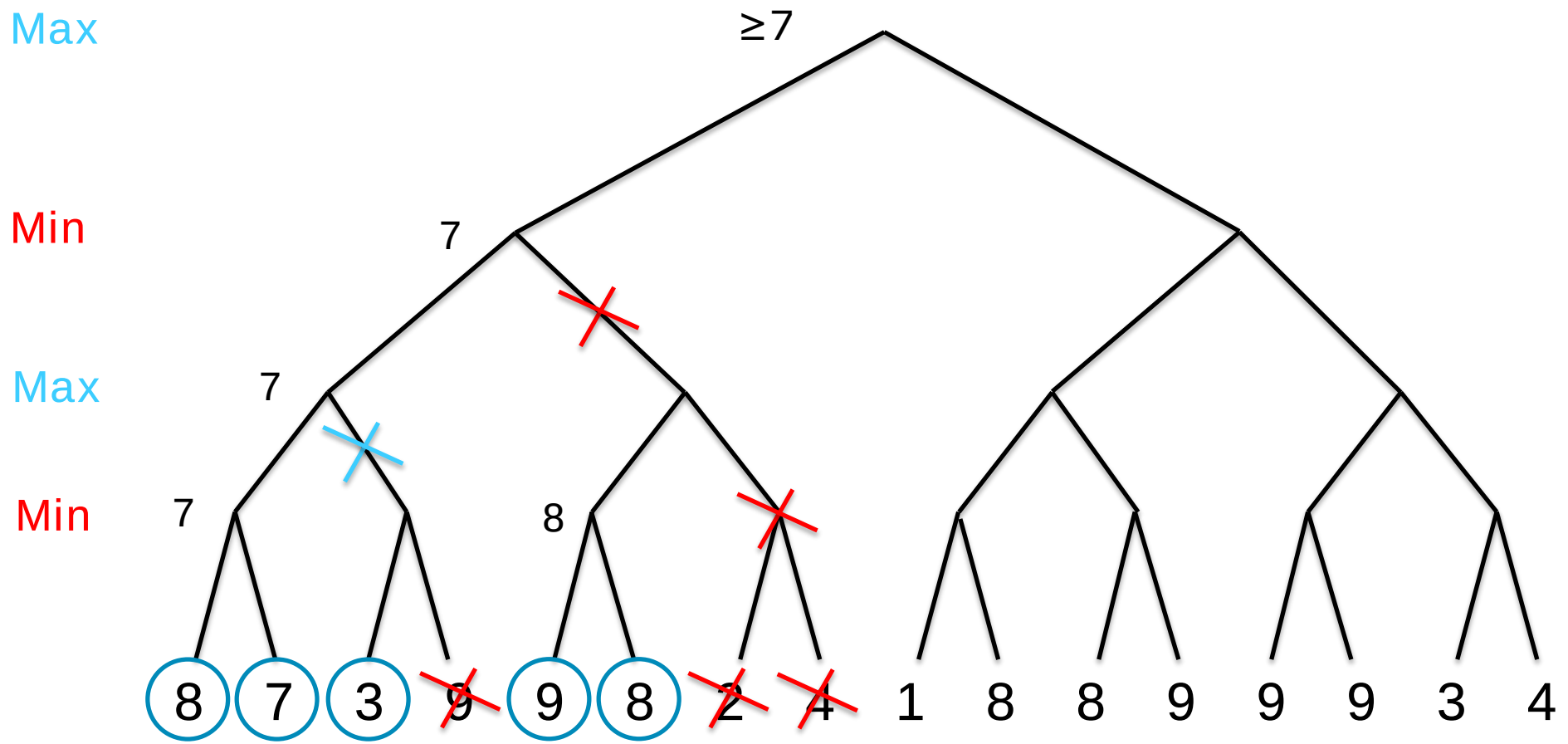

Max
Min
Max
Min
≤7
7
7
≤9
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4

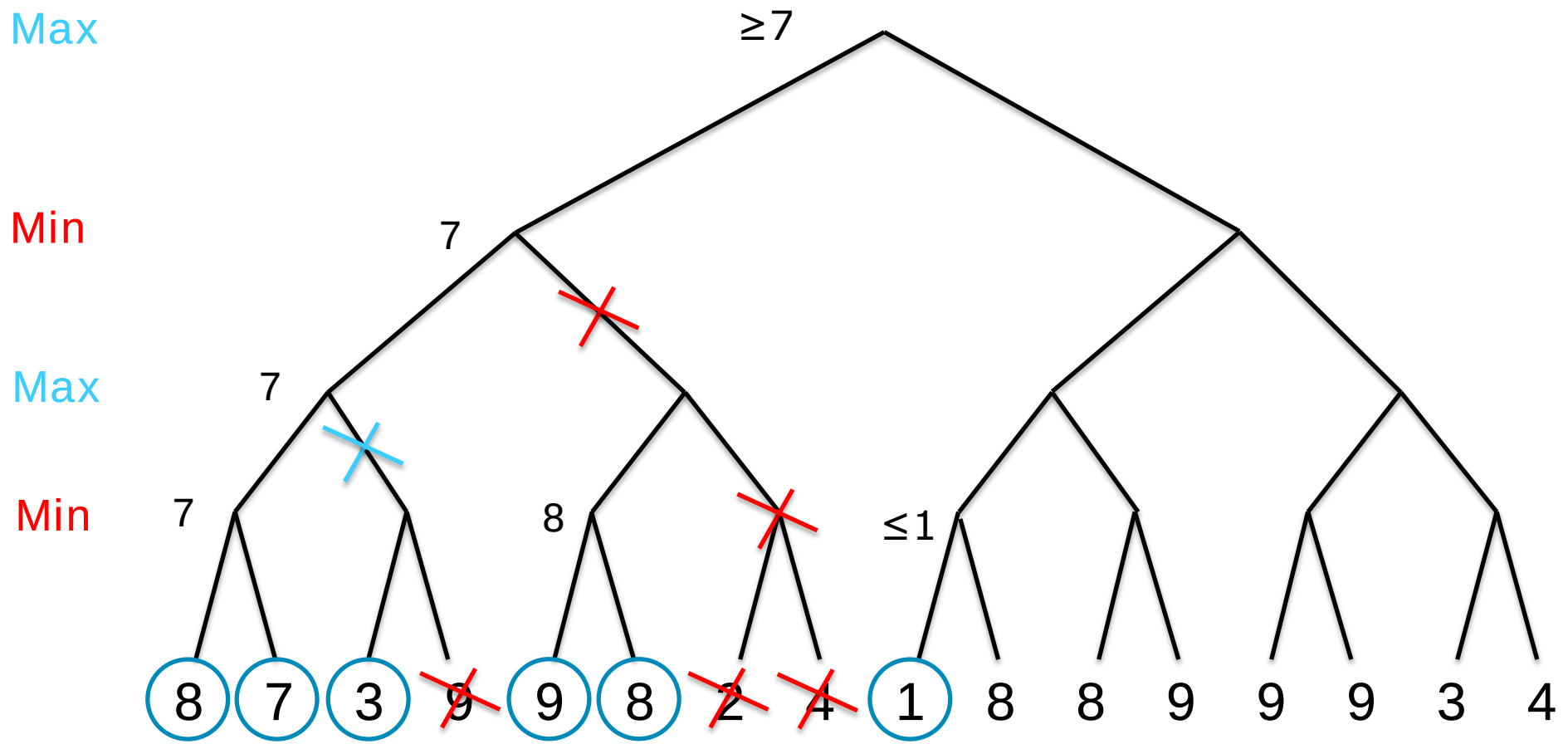

Max
Min
Max
Min
≤7
7
7
8
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4

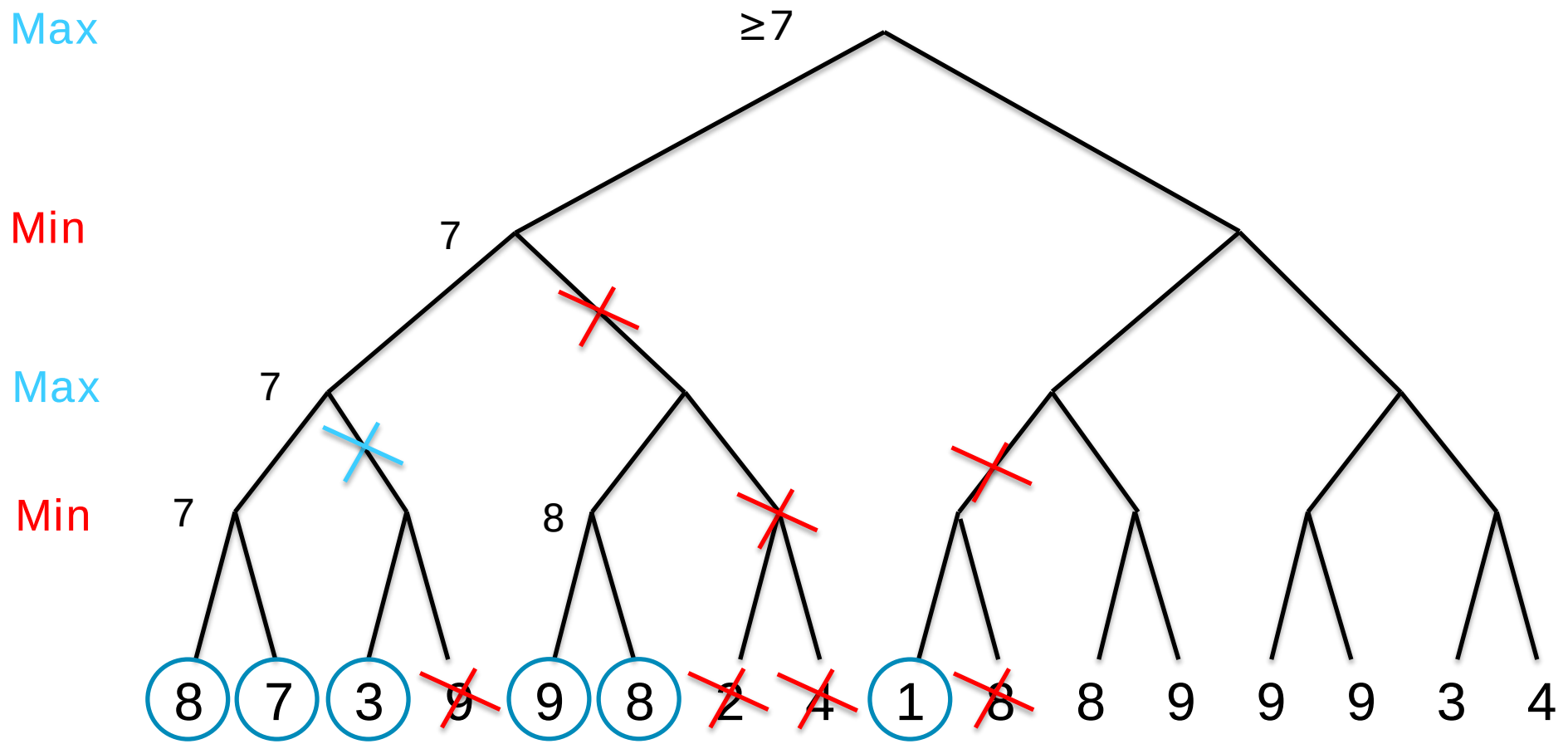

Max
Min
Max
Min
≤7
7
≥8
7
8
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4

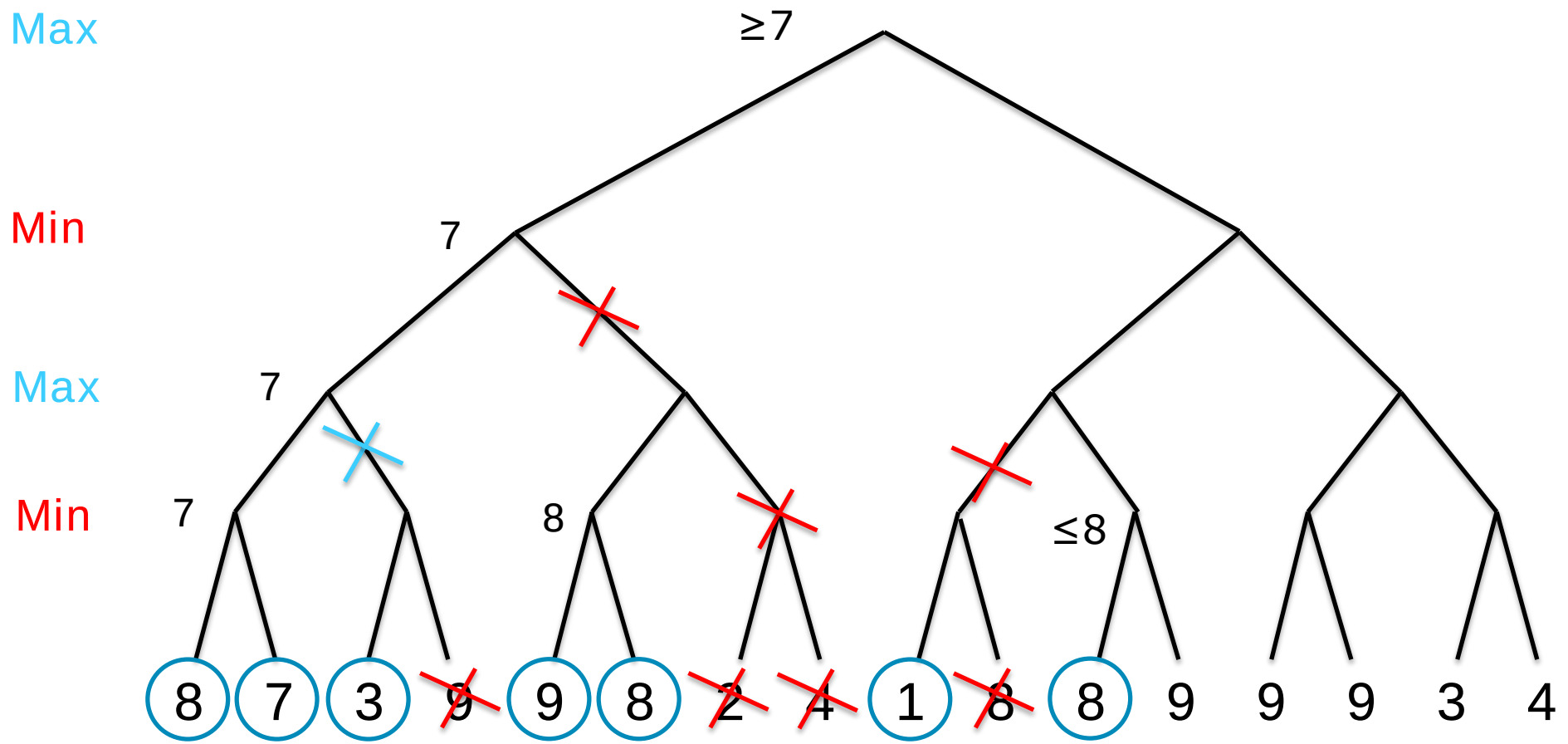

Max
Min
Max
Min
≤7
7
≥8
7
8
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4

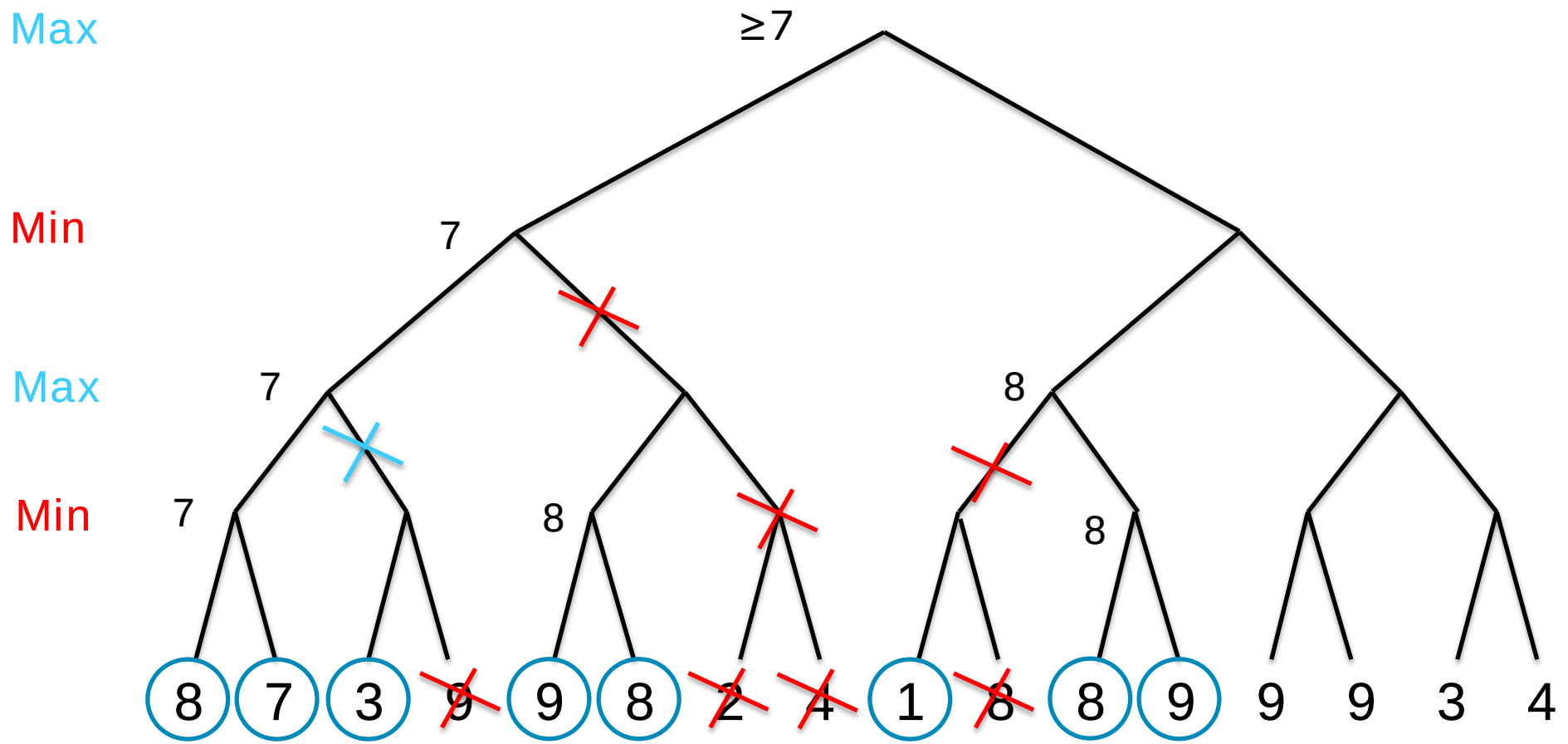

Max
Min
Max
Min
7
7
7
8
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4

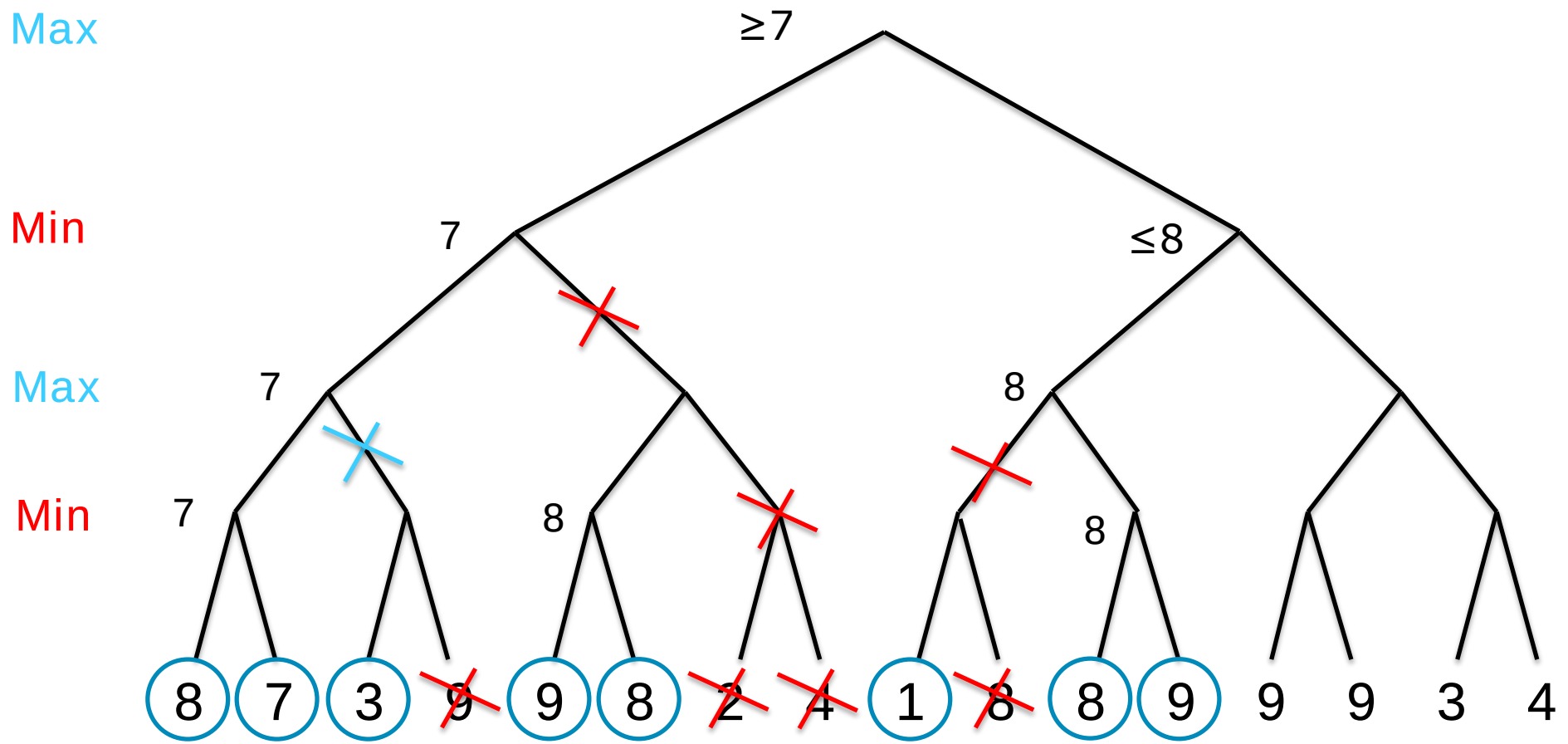

Max
Min
Max
Min
≥7
7
7
7
8
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4

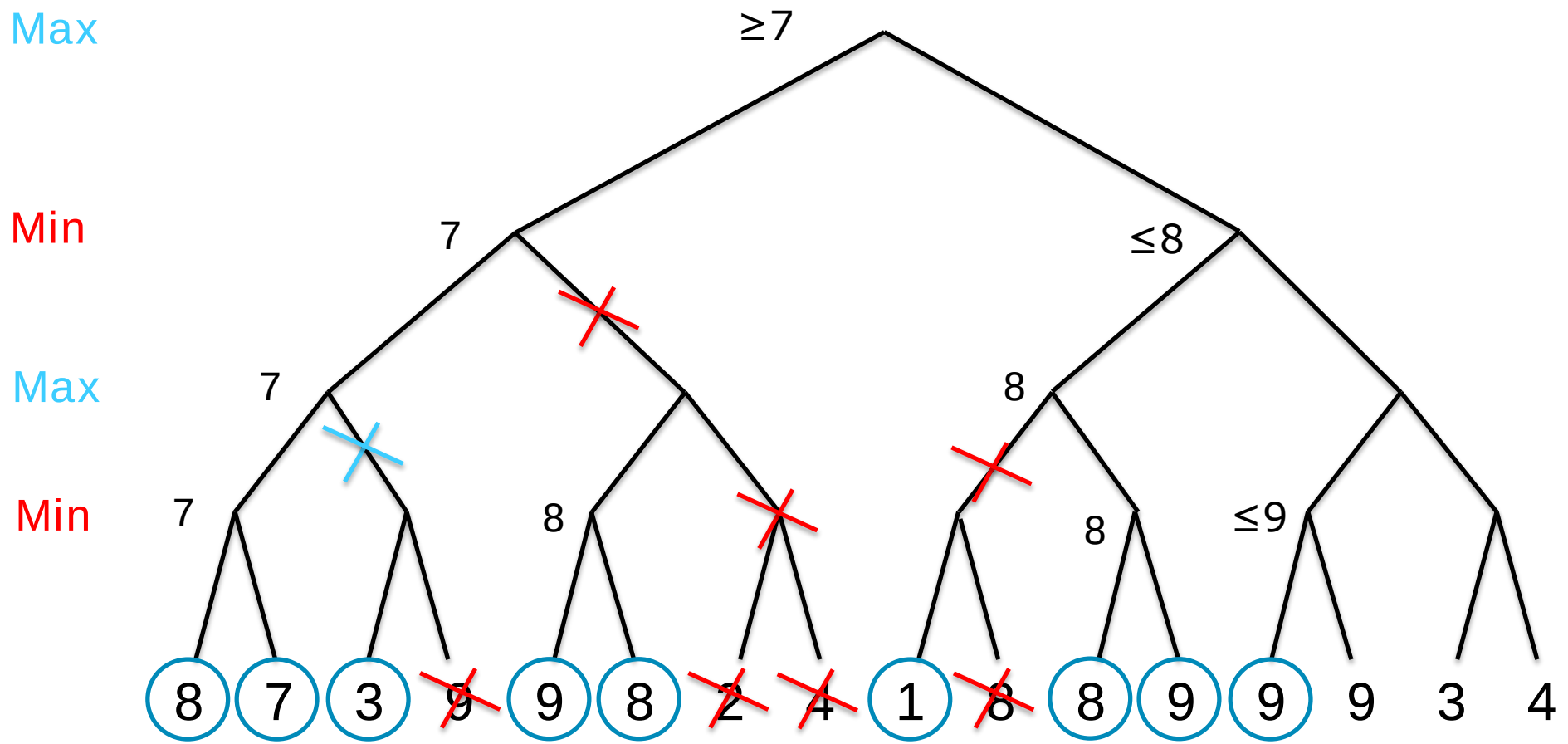

Max
Min
Max
Min
≥7
7
7
7
8
≤1
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4

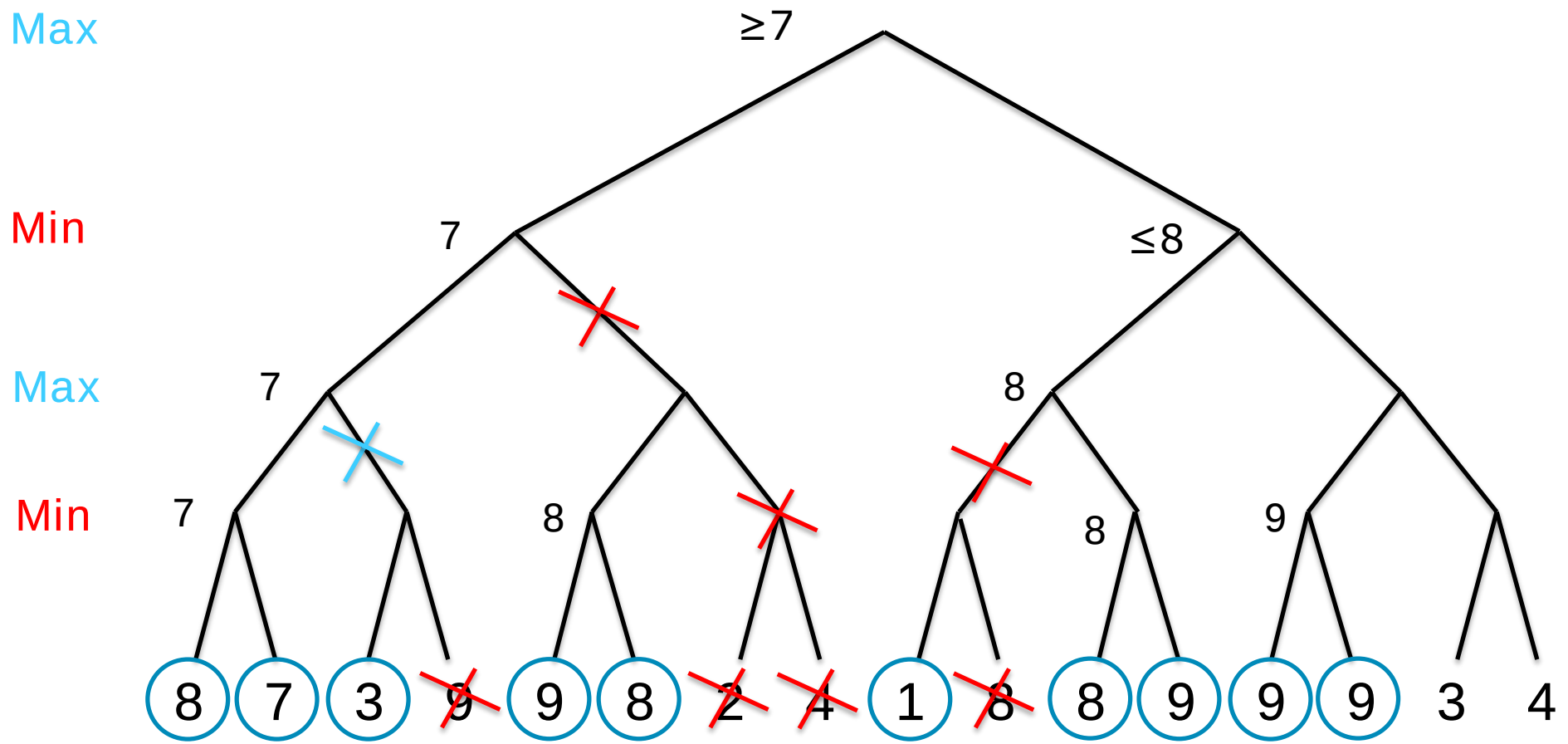

Max
Min
Max
Min
≥7
7
7
7
8
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4

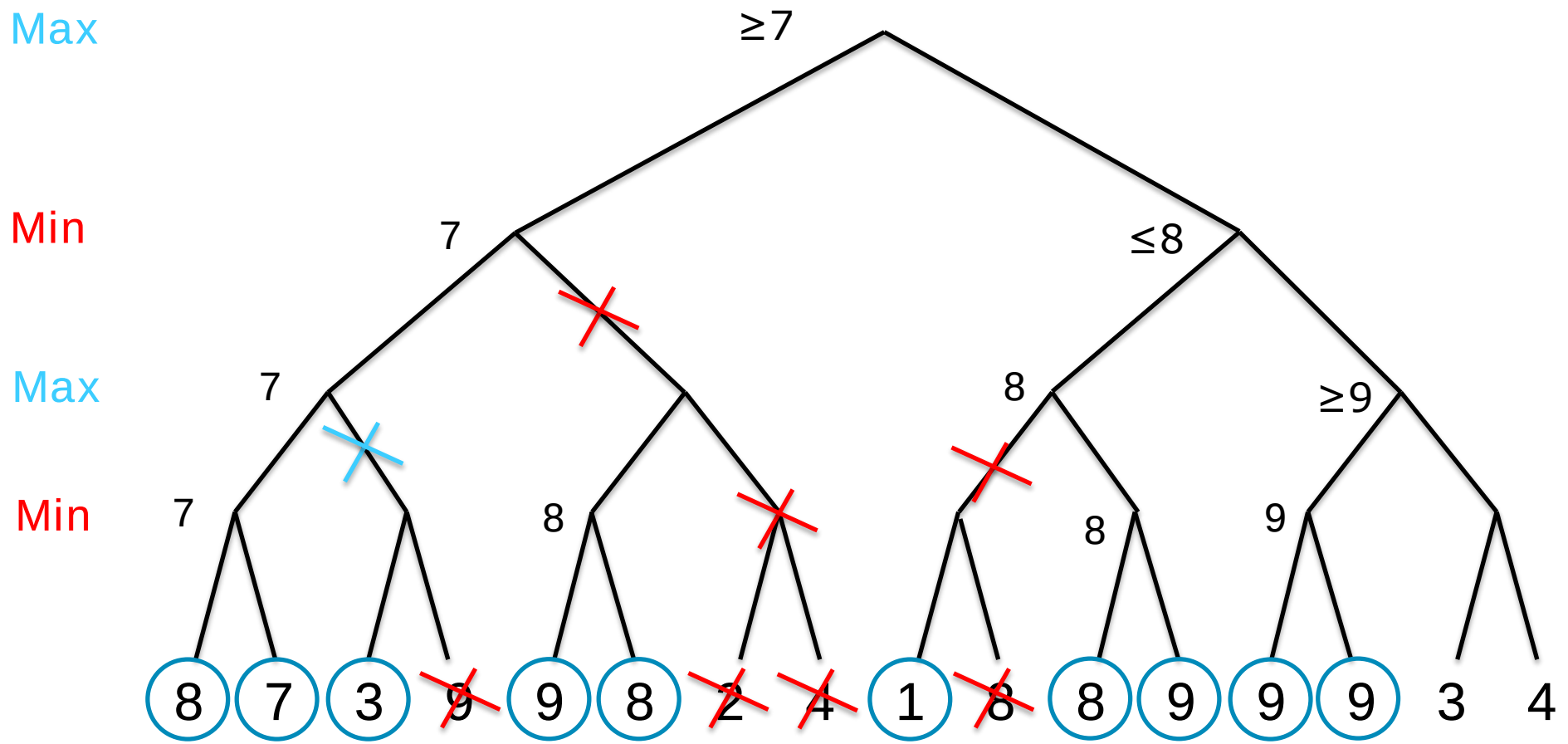

Max
Min
Max
Min
≥7
7
7
7
8
≤8
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4

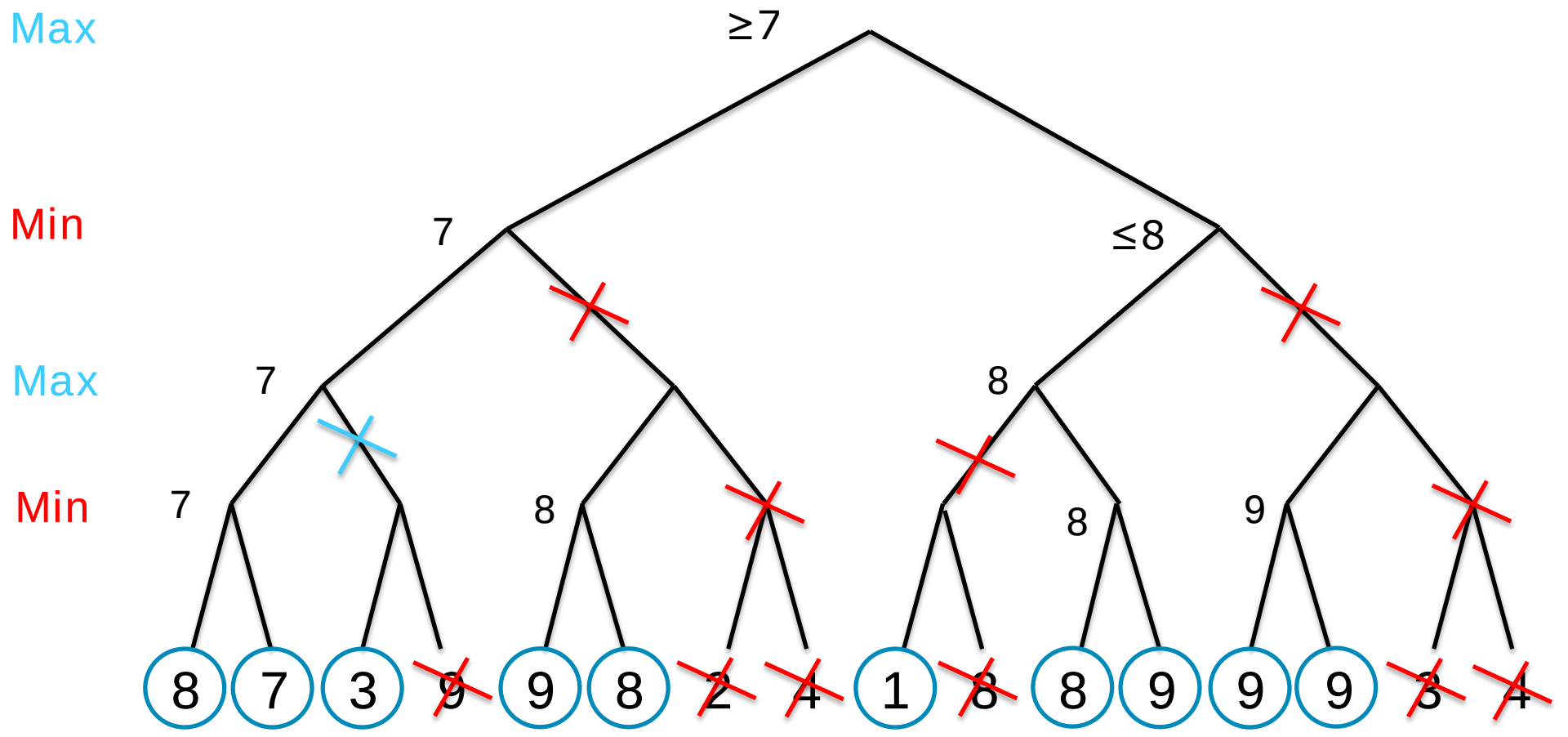

Max
Min
Max
Min
≥7
7
7
8
7
8
8
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4

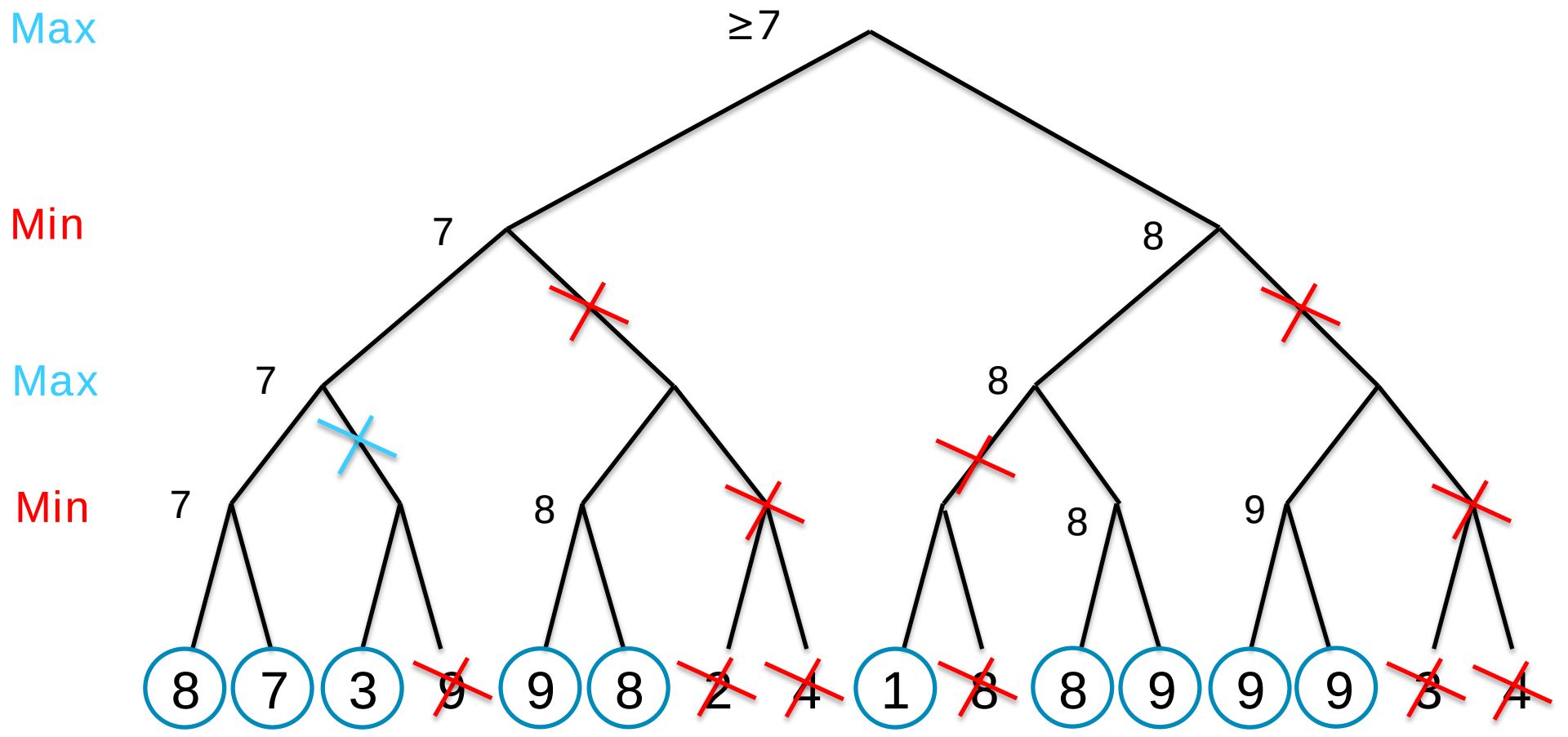

Max
Min
Max
Min
≥7
7
≤8
7
8
7
8
8
8 7 3 9 9 8 2 4 1 8 8 9 9 9 3 4

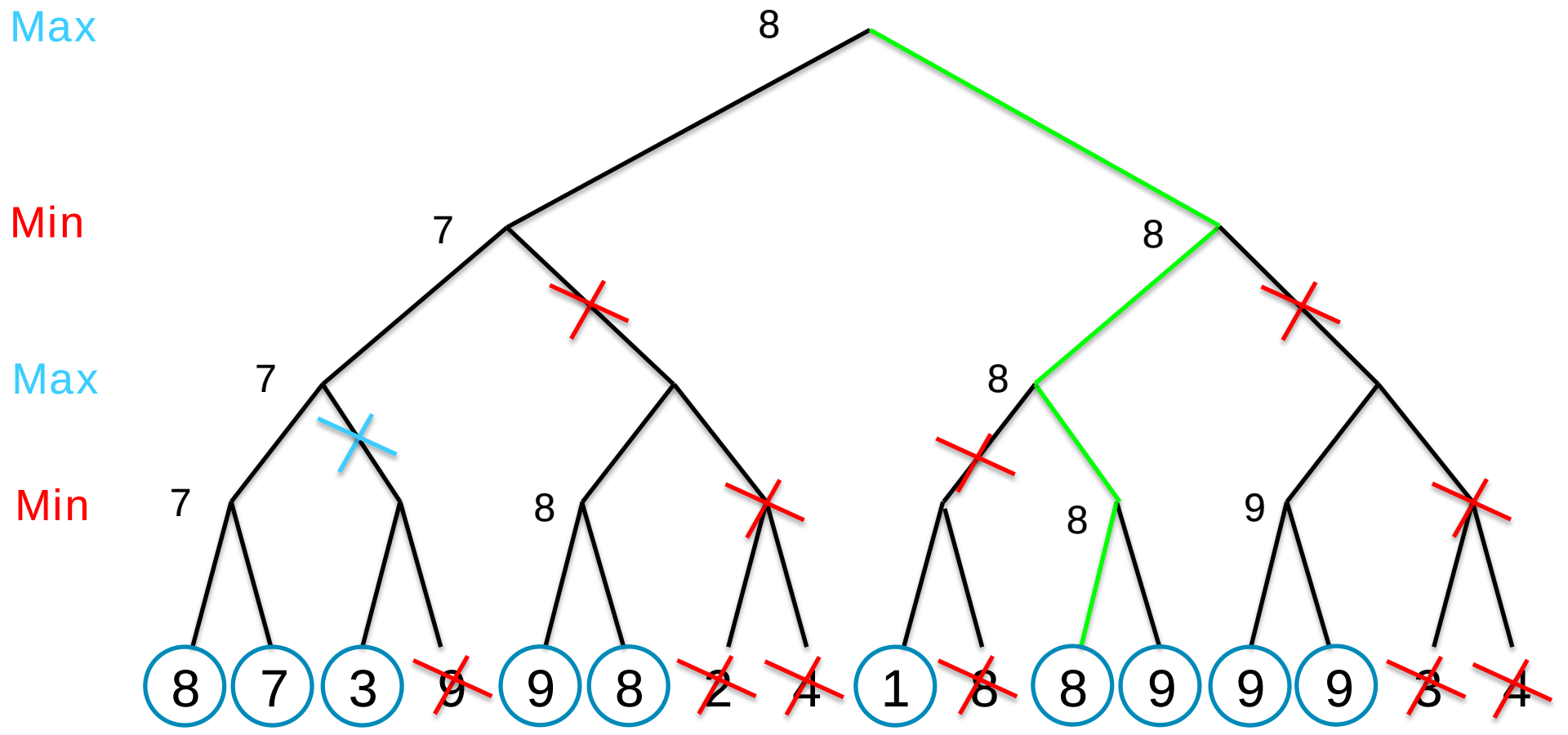

Max
Min
Max
Min
≥7
7
≤8
7
8
7
8
≤9
8
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4


Max
Min
Max
Min
≥7
7
≤8
7
8
7
8
8
9
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4


Max
Min
Max
Min
≥7
7
≤8
7
8
≥9
7
8
8
9
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4


Max
Min
Max
Min
≥7
7
≤8
7
8
7
8
9
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4


Max
Min
Max
Min
≥7
7
8
7
8
7
8
8
9
8
7
3
9
9
8
2
4
1
8
8
9
9
9
3
4


Max
Min
Max
Min
8
7
8
7
8
8
7
8
8
9
8
7
3
9
9
8
2
4
1
8
8
9
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

Alpha-Beta Algorithm

```
Function AlphaBeta(node, depth, alpha, beta, maxPlayer)
        if (depth > maxDepth or node.terminal)
            return eval(node)
        if (maxPlayer)
            value = -infinity
            for each child of node
                value = max(value, AlphaBeta(child, depth+1, alpha, beta, false))
                if (depth == 0 and value > alpha) best_action = child.action
                alpha = max(alpha, value)
                if (alpha >= beta) break
            return value
        else
            value = infinity
            for each child of node
                value = min(value, AlphaBeta(child, depth+1, alpha, beta, true))
                beta = min(beta, value)
                if (beta <= alpha) break
            return value
Initial Call: AlphaBeta(root, 0, -infinity, +infinity, true)
```

Alpha-Beta Algorithm (shorter)

```
Function AlphaBeta(node, depth, alpha, beta, maxPlayer)
        if (depth > maxDepth or node.terminal)
            return eval(node)
        for each child of node
            value = AlphaBeta(child, depth+1, alpha, beta, !maxPlayer)
            if (maxPlayer and (value > alpha))
                if (depth == 0) best_action = child.action
                alpha = value
            if (!maxPlayer and (value < beta)) beta = value
            if (alpha >= beta) break
        return alpha if maxPlayer else beta
Initial Call: AlphaBeta(root, 0, -infinity, +infinity, true)
```