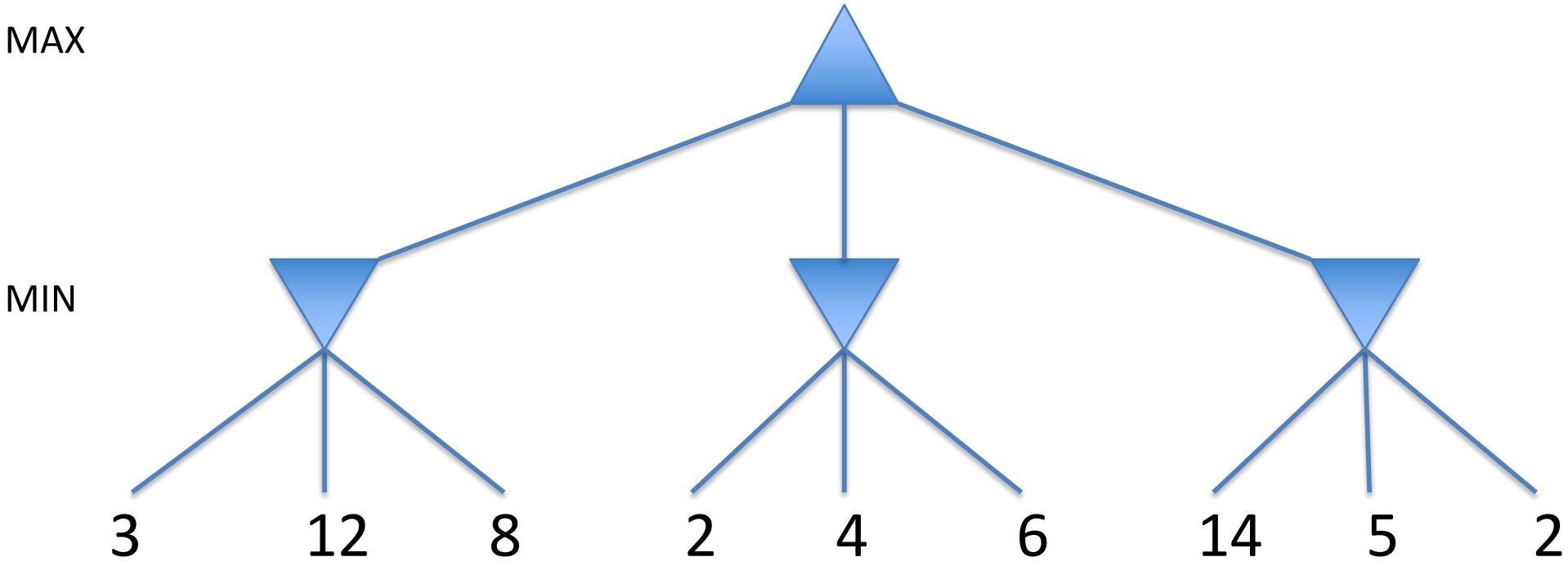
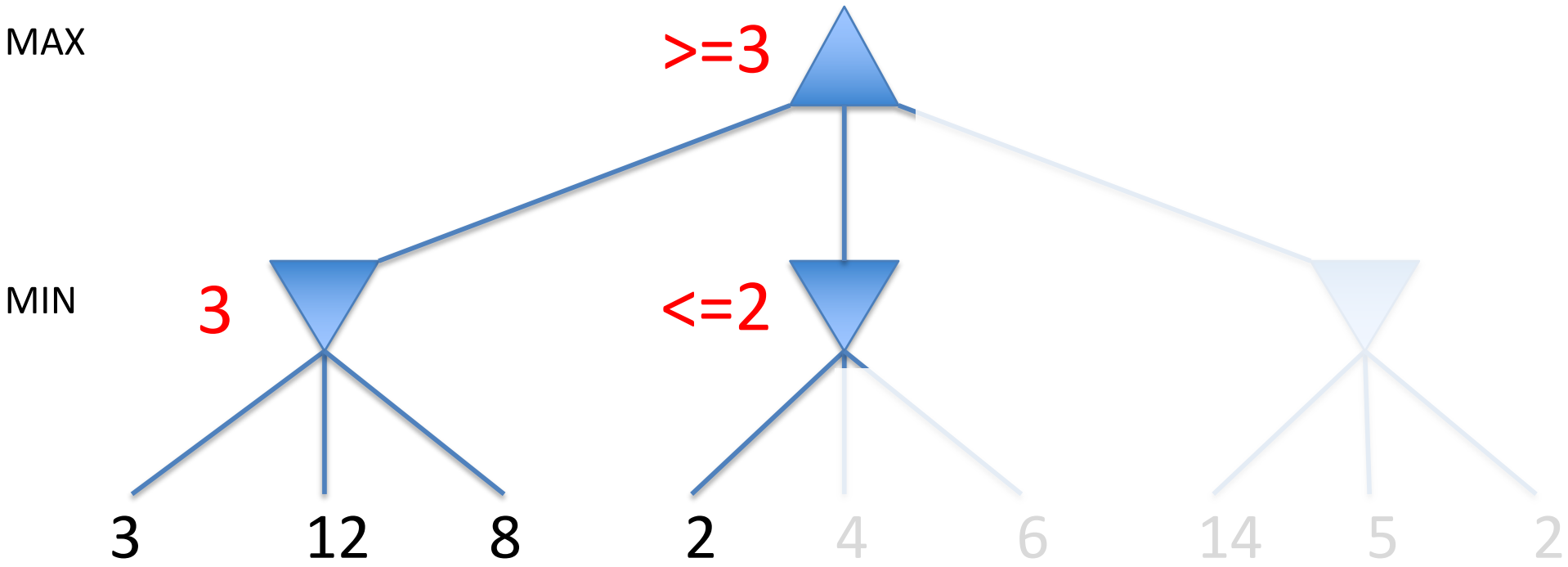
# Adversarial Search

- Problem: minimax takes too long.
- Solution: improve algorithm to ignore parts of the tree that will definitely not be used (assuming both players play optimally).


MAX
MIN
3
12
8
2
4
6
14
5
2


MAX
>=3
MIN
3
<=2
3
12
8
2
4
6
14
5
2

- Idea: for each node, keep track of the range of possible values that minimax could produce for that node.

- If we ever find ourselves at a node that we know will never be selected during (optimal) game play, we can "prune" it (end the recursion on this part of the tree).

- Enhanced version of minimax is called ***minimax with alpha-beta pruning.***

# Alpha-beta pruning

- Each node in the game tree needs two extra variables, called alpha and beta.
- Alpha and beta are inherited from parent nodes.
  - alpha = highest-value choice we've found so far (best move for MAX)
  - beta = lowest-value choice we've found so far (best choice for MIN)
- If at a MAX node, we see a child node that has a value >= than beta, **short-circuit**.
- If at a MIN node, we see a child node that has a value <= than alpha, **short-circuit**.

# Alpha-beta pruning

- Recall that minimax is a variant of depth-first search. During the algorithm, we will only consider nodes along the path from the root node to the current node.
- At each node in the search, we will maintain two variables:
  - alpha ($\alpha$) = highest numeric value we've found so far on this path (best move for MAX)
  - beta ($\beta$) = lowest numeric value we've found so far on this path (best choice for MIN)

# Alpha-beta pruning

- Alpha and beta are inherited from parent nodes as we recursively descend the tree.

- If at a MAX node, we see a child node that has a value >= than beta, **short-circuit**.
- If at a MIN node, we see a child node that has a value <= than alpha, **short-circuit**.

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

---

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

---

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

- The results of alpha-beta depend on the order in which moves are considered among the children of a node.
- If possible, consider better moves first!

# Real-world use of alpha-beta

- (Regular) minimax is normally run as a preprocessing step to find the optimal move from every possible situation.
- Minimax with alpha-beta can be run as a preprocessing step, but might have to re-run during play if a non-optimal move is chosen.
- Save states somewhere so if we re-encounter them, we don't have to recalculate everything.

# Real-world use of alpha-beta

- States get repeated in the game tree because of *transpositions*.
- When you discover a best move in minimax or alpha-beta, save it in a lookup table (probably a hash table).
  - Called a *transposition table*.

# Real-world use of alpha-beta

- In the real-world, alpha-beta does not "pre-generate" the game tree.
  - The whole point of alpha-beta is to not have to generate all the nodes.
- The DFS part of minimax/alpha-beta is what generates the tree.