Adversarial Search and Game Playing

(Where making good decisions requires respecting your opponent)

R&N: Chap. 5

- Games like Chess or Go are compact settings that mimic the uncertainty of interacting with the natural world
- For centuries humans have used them to exert their intelligence
- Recently, there has been great success in building game programs that challenge human supremacy

Games

- In this chapter we cover **competitive environments**
  - The agents' goals are in conflict
  - Giving rise to adversarial search problems
- Mathematical **game theory**, a branch of economics, views any multiagent environment as a game, provided that the impact of each agent on the others is "significant," regardless of whether the agents are **cooperative** or **competitive**.

Games

- In AI, the most common games are of a rather specialized kind what **game theorists call** deterministic, turn-taking, two-player, zero-sum games of perfect information (such as chess).
- In **our terminology**, this means deterministic, fully observable environments in which two agents act alternately and in which the utility values at the end of the game are always equal and opposite.
  - If one player wins a game of chess, the other player necessarily loses.

Specific Setting

Two-player, turn-taking, deterministic, fully observable, zero-sum, time-constrained game

- State space
- Initial state
- Successor function (RESULT(s, a)): it tells which actions can be executed in each state and gives the successor state for each action
- MAX's and MIN's actions alternate, with MAX playing first in the initial state
- Terminal test: it tells if a state is terminal and, if yes, if it's a win or a loss for MAX, or a draw
- All states are fully observable

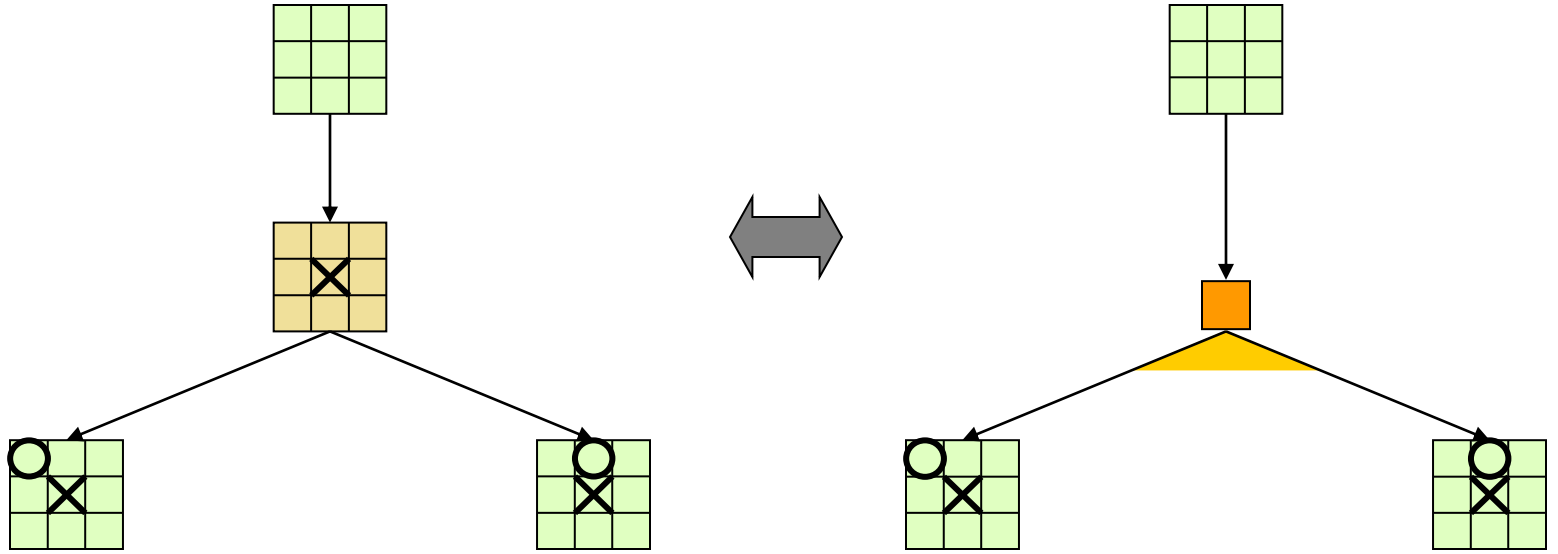
Specific Setting

Two-player, turn-taking, deterministic, fully observable, zero-sum, time-constrained game

- PLAYER(s)
  - Defines which player has the move in a state
- ACTIONS(s)
  - Returns the set of legal moves in a state.
- UTILITY(s, p)
  - A utility function, defines the final numeric value for a game that ends in terminal state s for a player p.
- A zero-sum game is (confusingly) 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 1/2 + 1/2

Relation to Previous Lecture

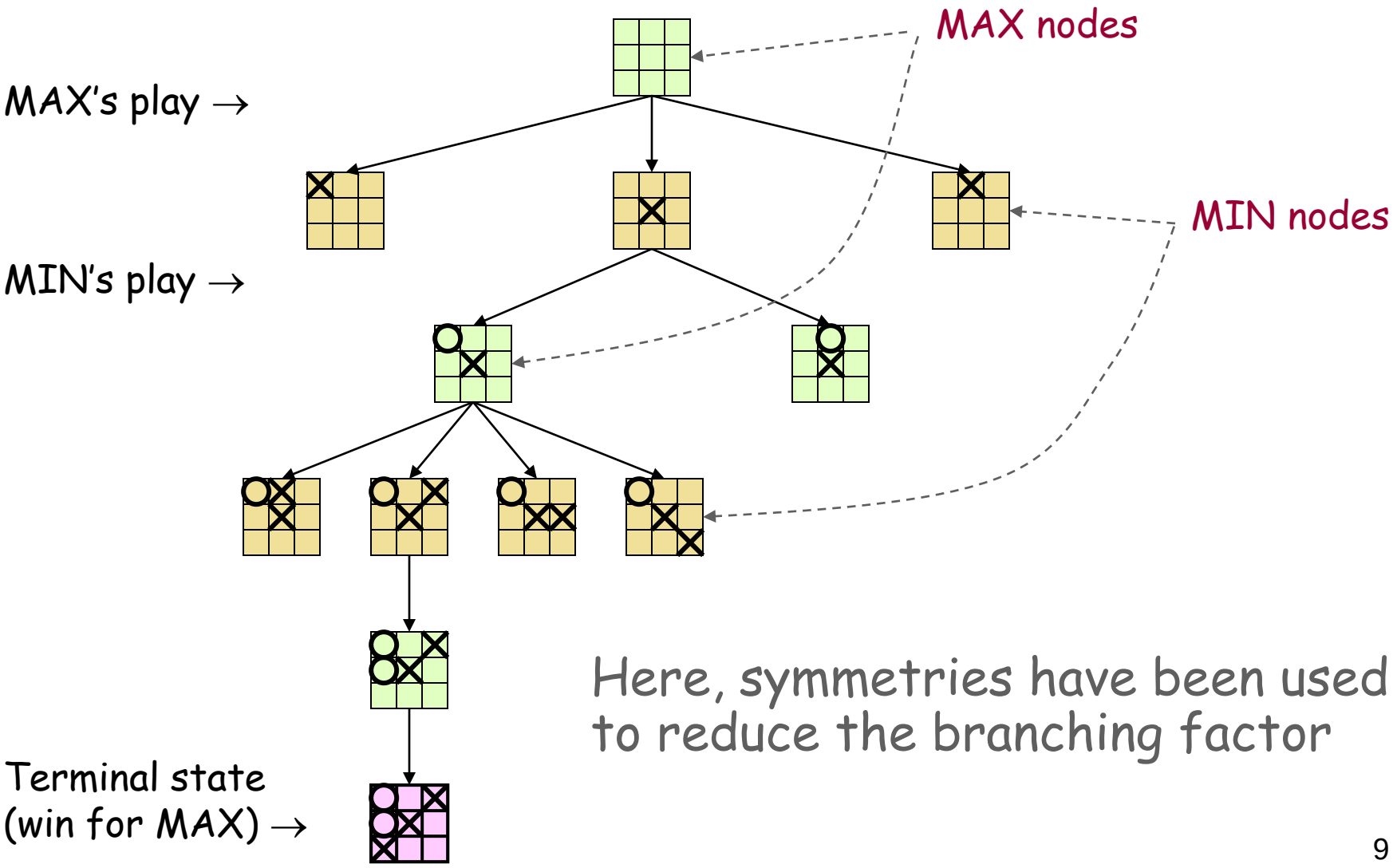
- Here, uncertainty is caused by the actions of another agent (MIN), who competes with our agent (MAX)



Relation to Previous Lecture

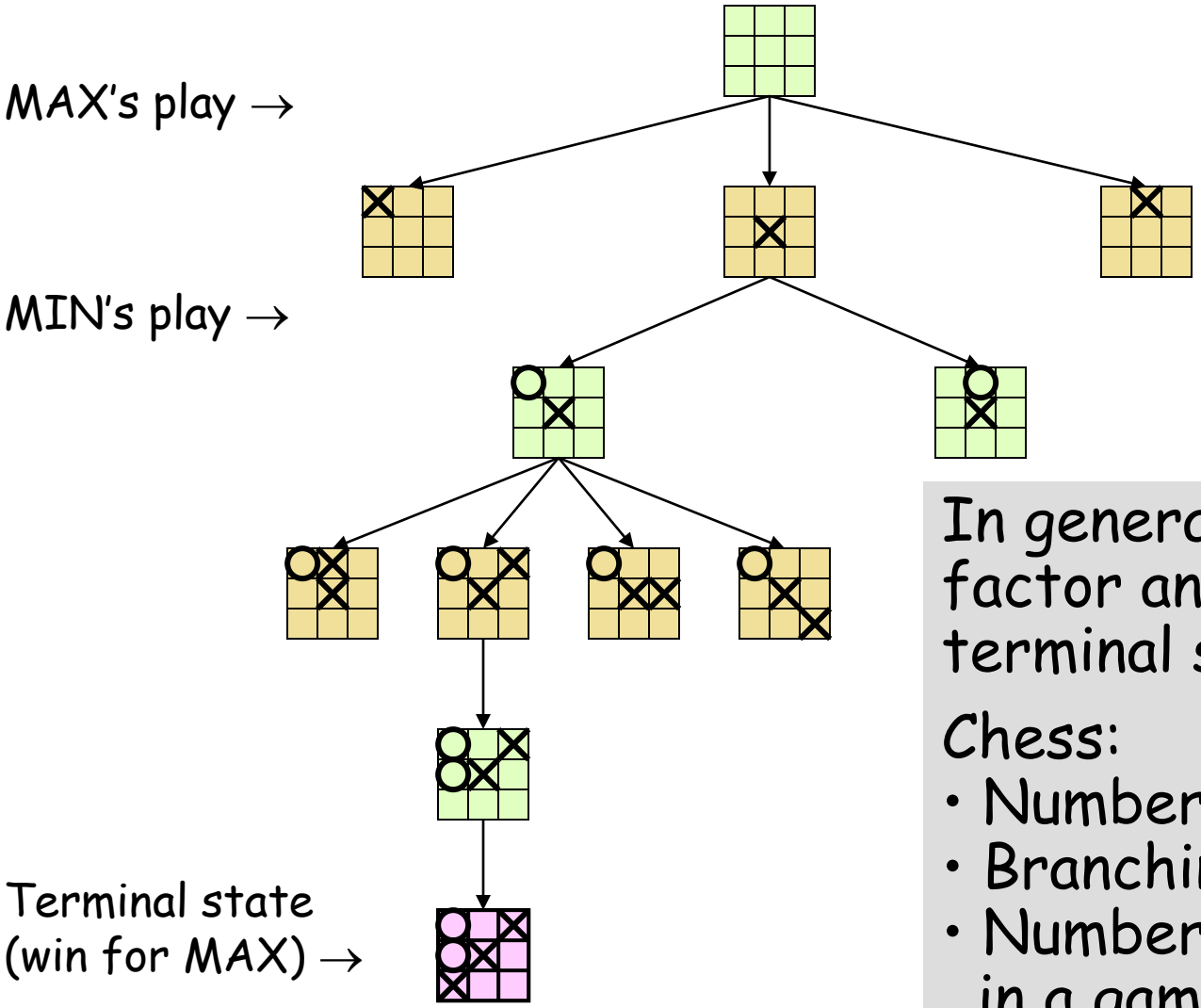
- Here, uncertainty is caused by the actions of another agent (MIN), who competes with our agent (MAX)
- MIN wants MAX to lose (and vice versa)
- No plan exists that guarantees MAX's success regardless of which actions MIN executes (the same is true for MIN)
- At each turn, the choice of which action to perform must be made within a specified time limit
- The state space is enormous: only a tiny fraction of this space can be explored within the time limit

Game Tree



Here, symmetries have been used to reduce the branching factor

Game Tree



In general, the branching factor and the depth of terminal states are large

Chess:

- Number of states: ~10^{40}
- Branching factor: ~35
- Number of total moves in a game: ~100

Choosing an Action: Basic Idea

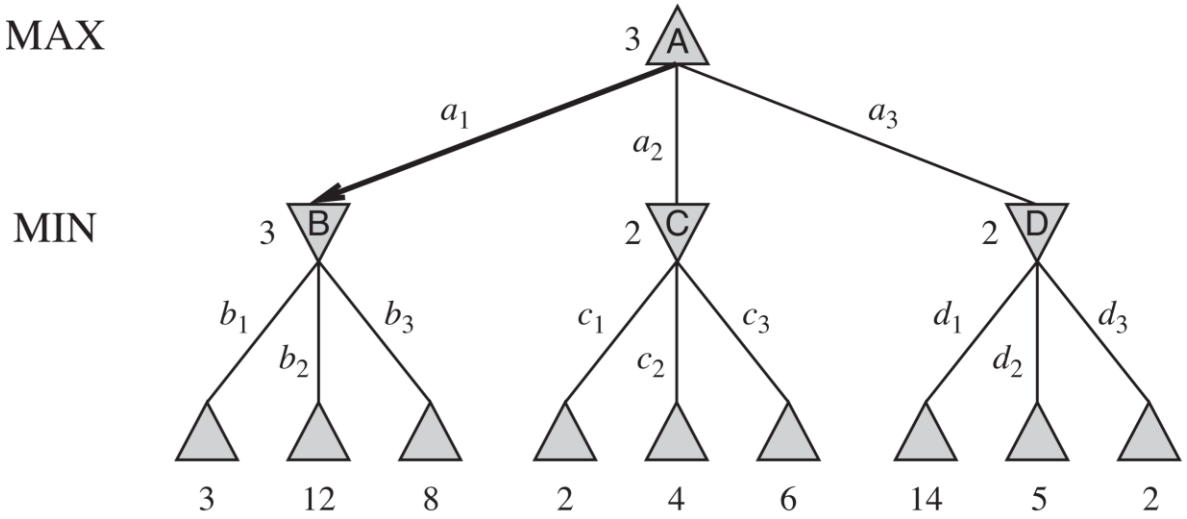
1) Using the current state as the initial state, build the game tree uniformly to the maximal depth h (called horizon) feasible within the time limit
2) Evaluate the states of the leaf nodes
3) Back up the results from the leaves to the root and pick the best action assuming the worst from MIN

→ Minimax algorithm

Minimax Algorithm

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

- MINIMAX(s) shows the best achievable outcome of being in state (assumption: optimal opponent)
  - Maximizes the worst-case outcome for MAX



Minimax Algorithm

```
function MINIMAX-DECISION(state) returns an action
    return argmax_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))
```

```
function MAX-VALUE(state) 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)))
    return v
```

```
function MIN-VALUE(state) 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)))
    return v
```

Computational time limit (example)

- 100 secs is allowed for each move (game rule)
- 10^4 nodes/sec (processor speed)
- We can explore just 10^6 nodes for each move
  - $b^m = 10^6$, $b=35 \Rightarrow m=4$
- We must make a decision even when finding the optimal move is infeasible.

Computational time limit (Solution)

- Cut off the search and apply a heuristic evaluation function
  - cutoff test: turns non-terminal nodes into terminal leaves
    - Cut off test instead of terminal test (e.g., depth limit)
  - evaluation function: estimated desirability of a state
    - Heuristic function evaluation instead of utility function
- This approach does not guarantee optimality.

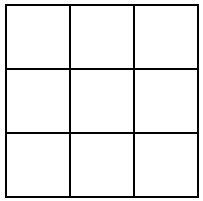
$$\text{H-MINIMAX}(s,d) = \begin{cases} \text{EVAL}(s) & \text{if CUTOFF-TEST}(s,d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s,a), d+1) & \text{if PLAYER}(s) = \text{MIN}. \end{cases}$$

Evaluation Function

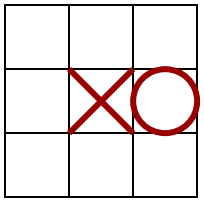
- Function e: state $s \rightarrow$ number $e(s)$
- $e(s)$ is a heuristics that estimates how favorable s is for MAX
- $e(s) > 0$ means that s is favorable to MAX (the larger the better)
- $e(s) < 0$ means that s is favorable to MIN
- $e(s) = 0$ means that s is neutral

Example: Tic-tac-Toe

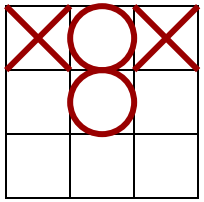
e(s) = number of rows, columns, and diagonals open for MAX – number of rows, columns, and diagonals open for MIN



8–8 = 0



6–4 = 2



3–3 = 0

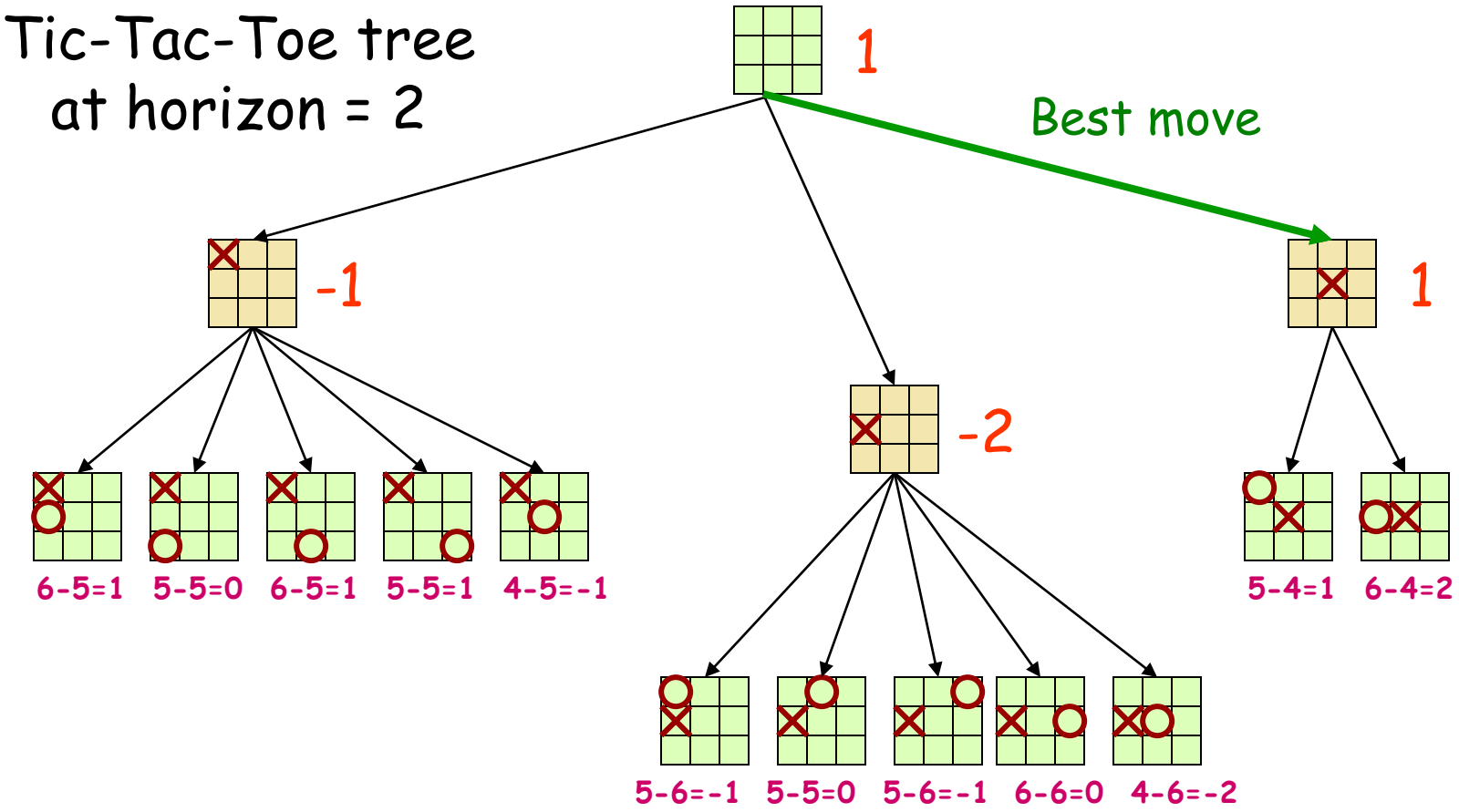
Construction of an Evaluation Function

- Usually a weighted sum of "features":

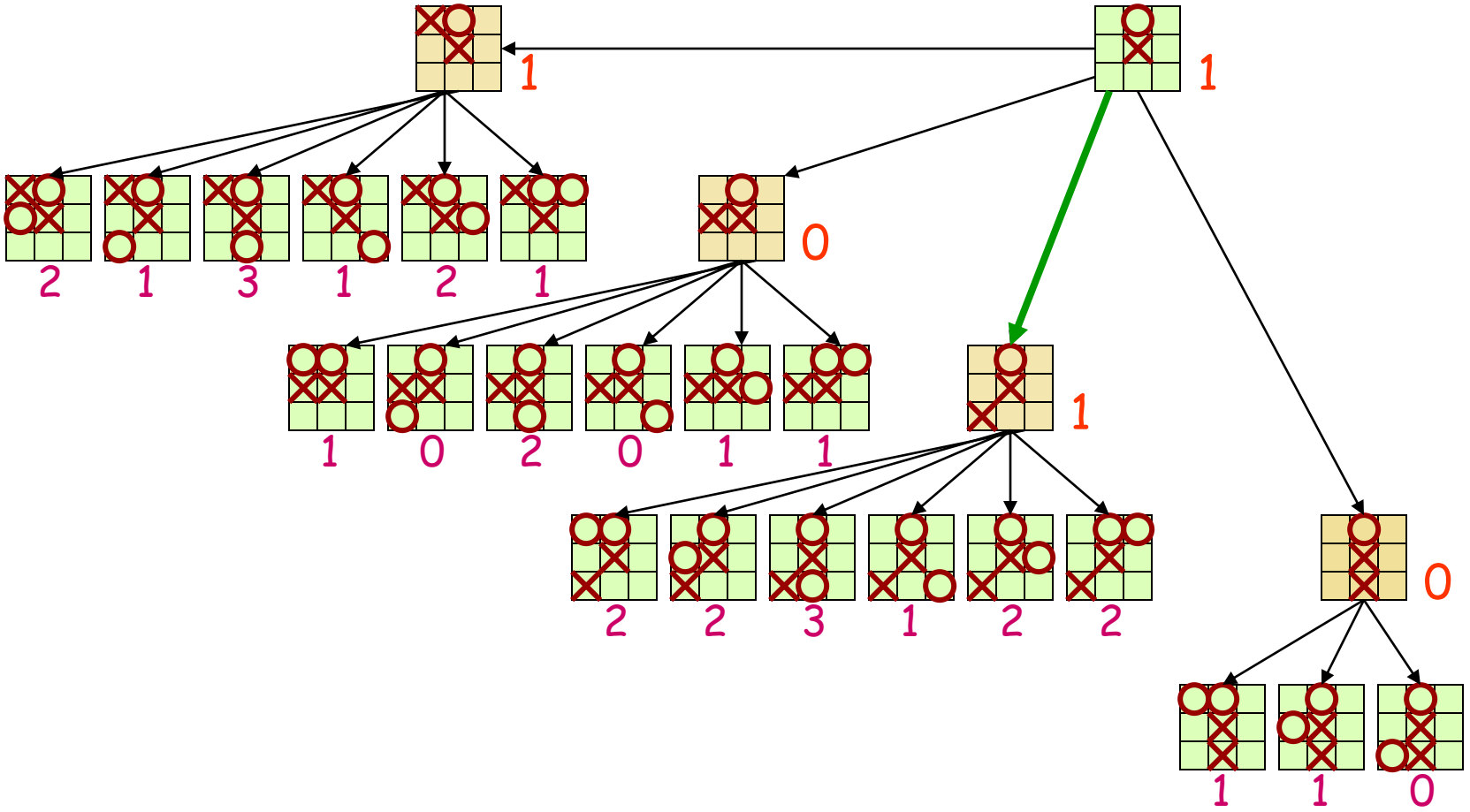
$$e(s)=\sum_{i=1}^{n} w_i f_i(s)$$

- Features may include
  - Number of pieces of each type
  - Number of possible moves
  - Number of squares controlled
- Weights can be assigned based on the human experience or machine learning methods.

Backing up Values



Continuation



Why using backed-up values?

- At each non-leaf node N, the backed-up value is the value of the best state that MAX can reach at depth h if MIN plays well (by the same criterion as MAX applies to itself)
- If e is to be trusted in the first place, then the backed-up value is a better estimate of how favorable STATE(N) is than e(STATE(N))

Minimax Algorithm

1. Expand the game tree uniformly from the current state (where it is MAX's turn to play) to depth h
2. Compute the evaluation function at every leaf of the tree
3. Back-up the values from the leaves to the root of the tree as follows:
   a. A MAX node gets the maximum of the evaluation of its successors
   b. A MIN node gets the minimum of the evaluation of its successors
4. Select the move toward a MIN node that has the largest backed-up value

Minimax Algorithm

1. Expand the game tree uniformly from the current state (where it is MAX's turn to play) to depth h
2. Compute the evaluation function at every leaf of the tree
3. Back-up the values from the leaves to the root of the tree as
   a. A MAX no successors
   b. A MIN node successors
4. Select the move toward a MIN node that has the largest backed-up value

Horizon: Needed to return a decision within allowed time

Repeated States

Left as an exercise

[Distinguish between states on the same path and states on different paths]

Game Playing (for MAX)

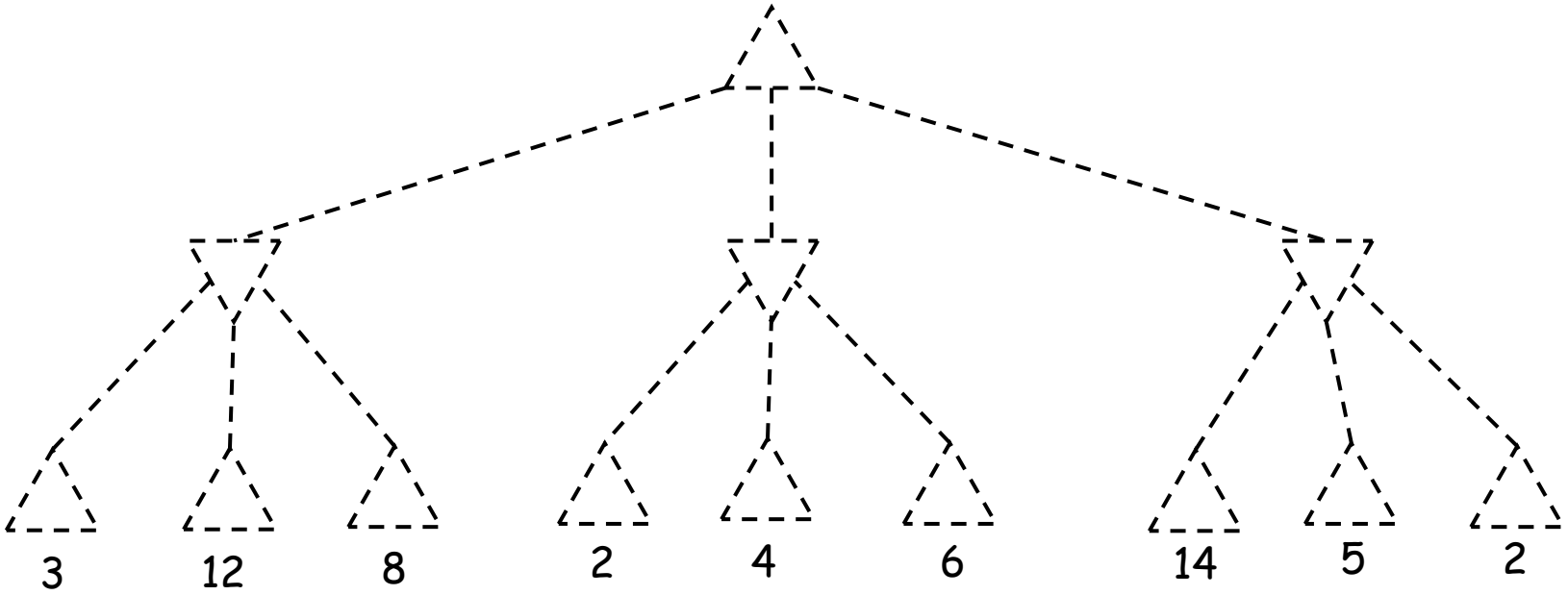
Repeat until a terminal state is reached

1. Select move using Minimax
2. Execute move
3. Observe MIN's move

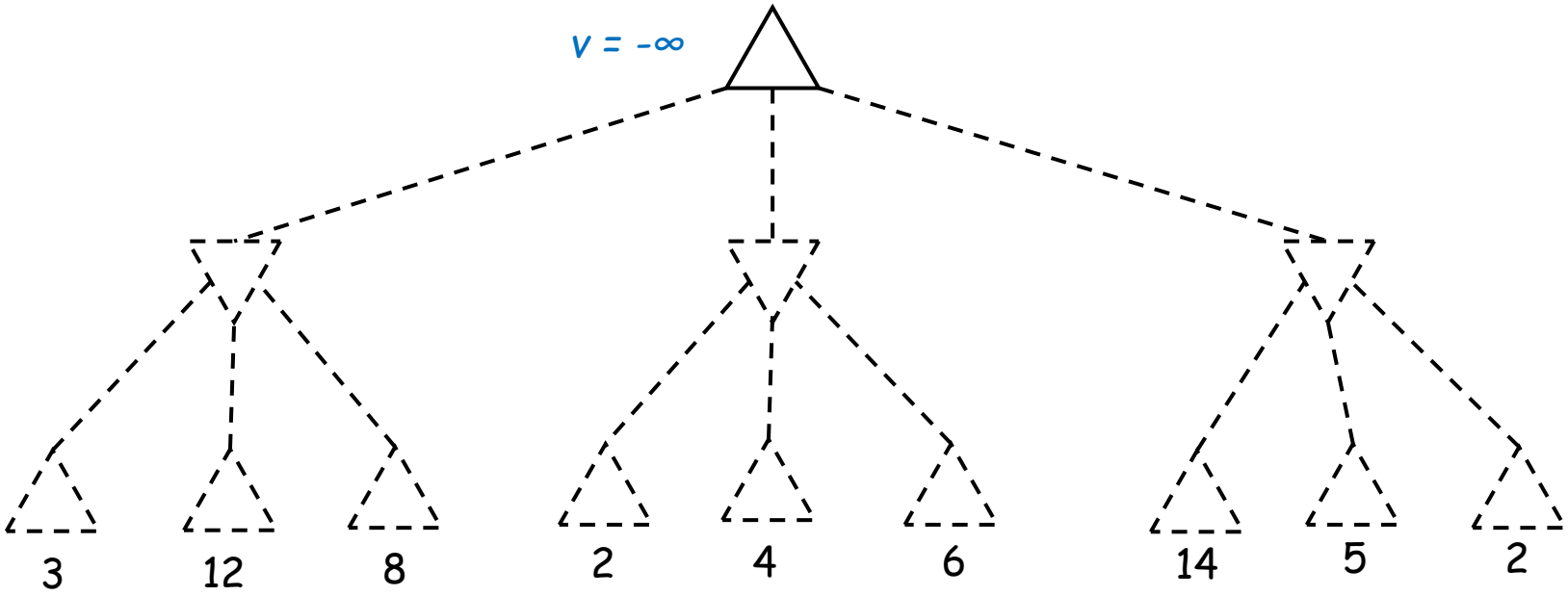
Note that at each cycle the large game tree built to horizon h is used to select only one move

All is repeated again at the next cycle (a sub-tree of depth h-2 can be re-used)

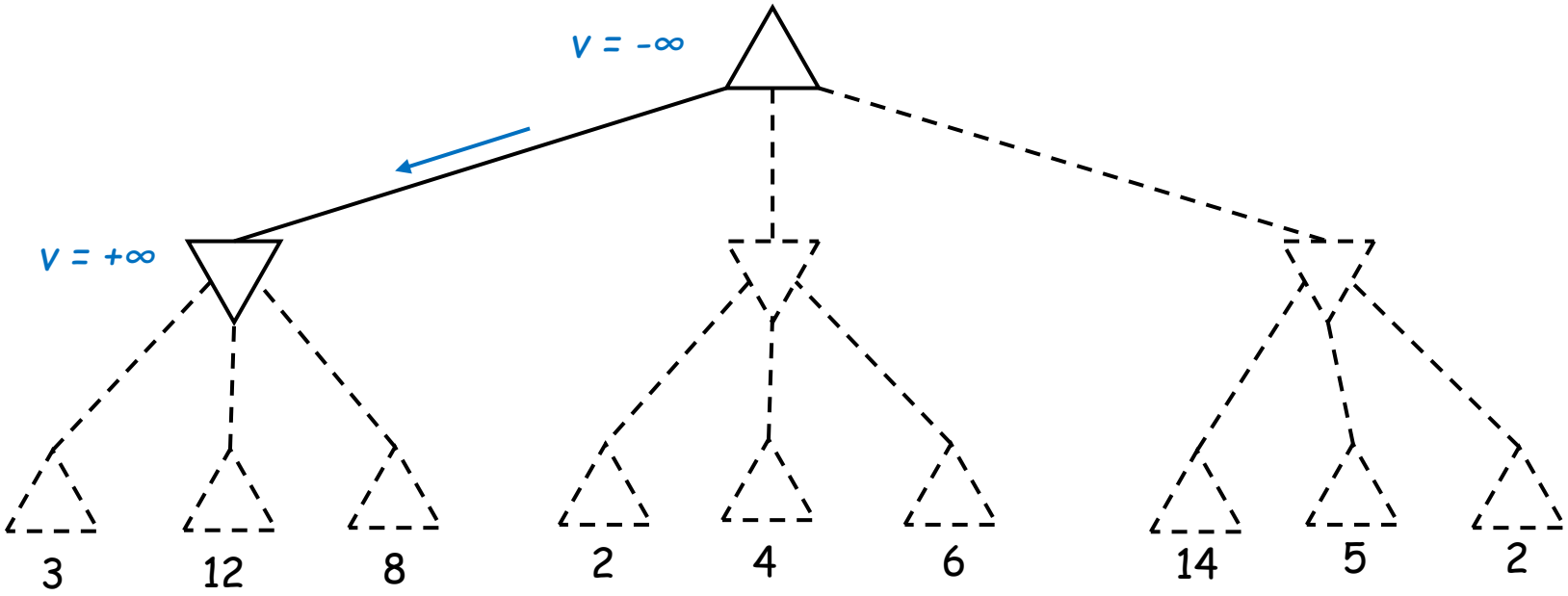
Example



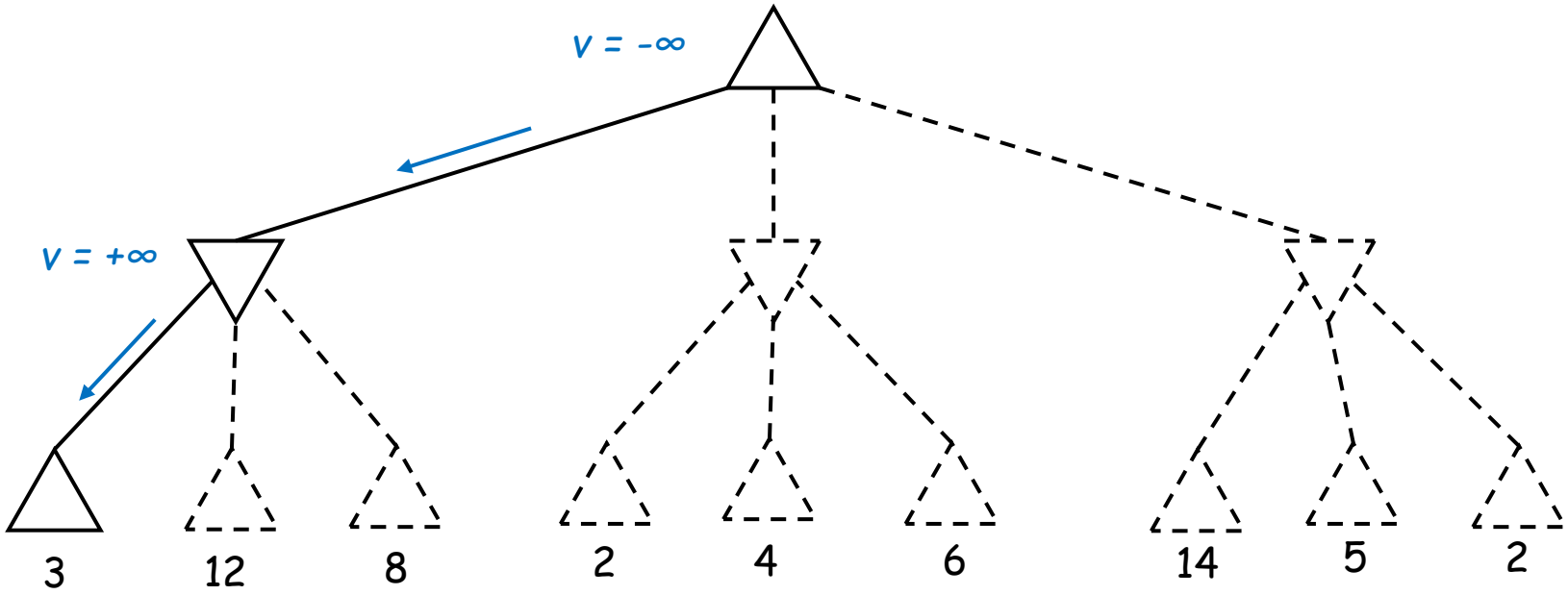
Example



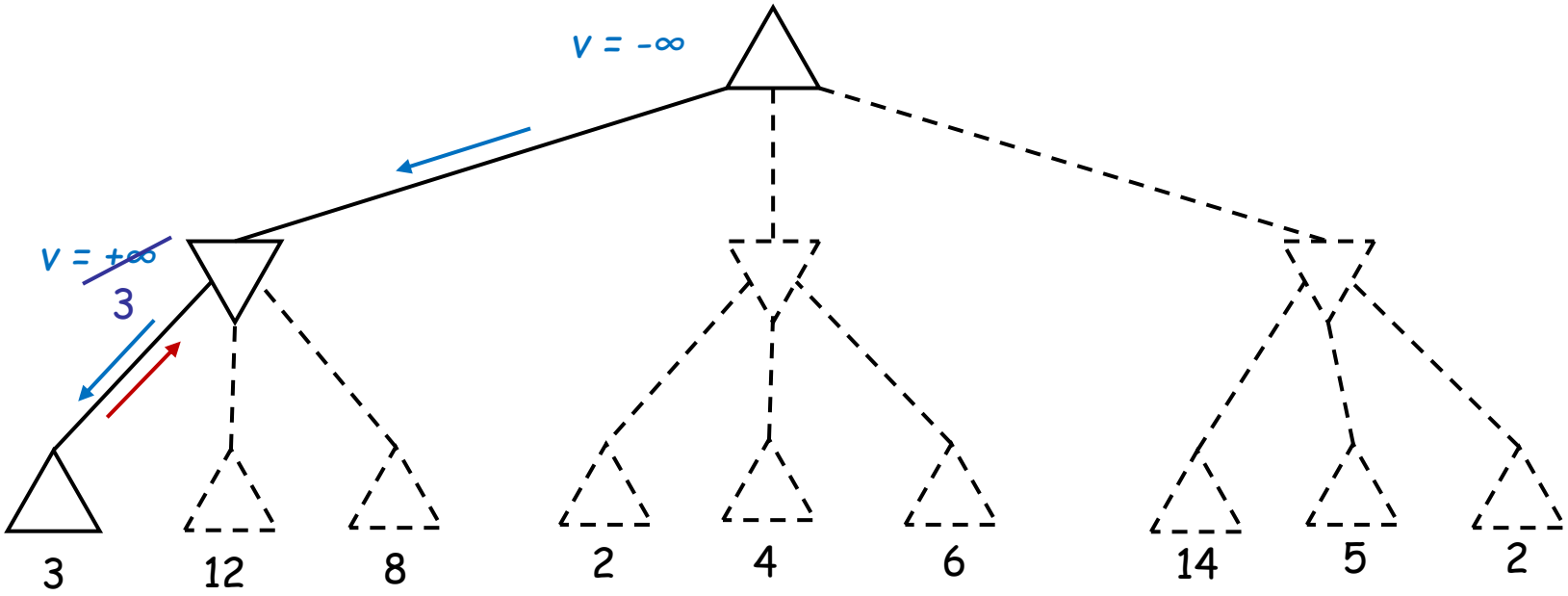
Example



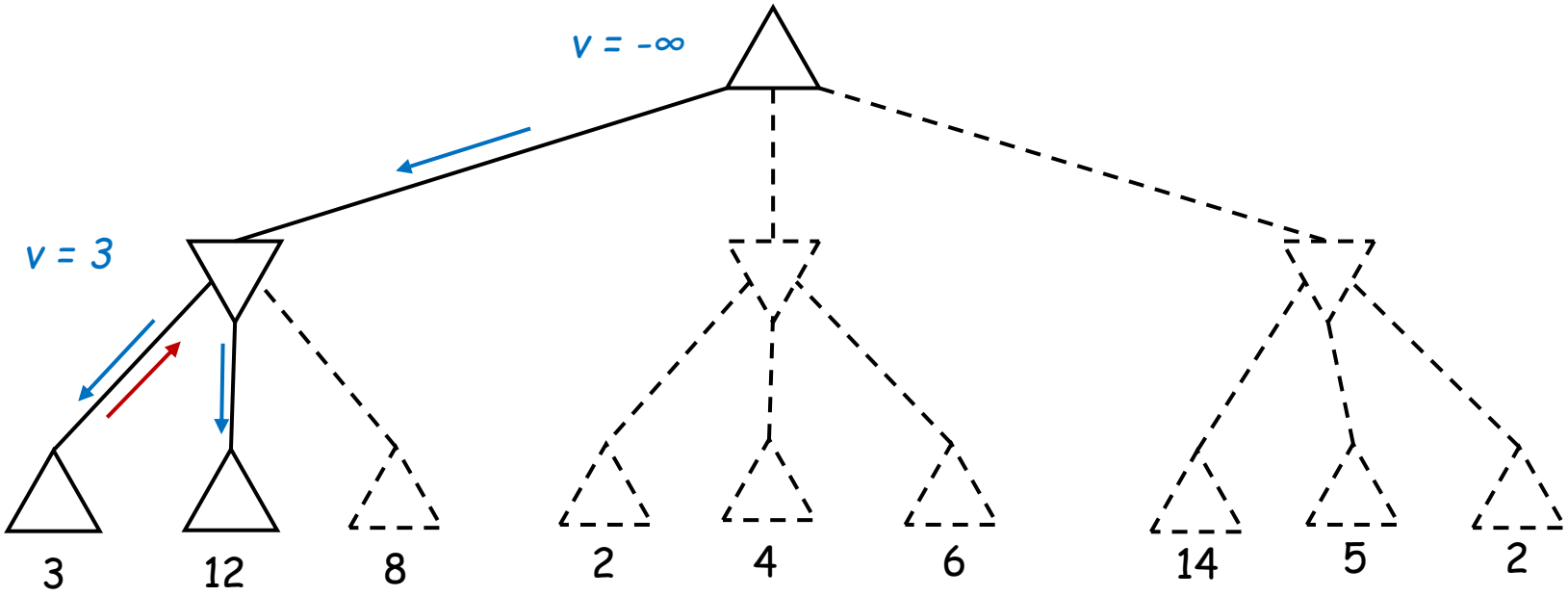
Example



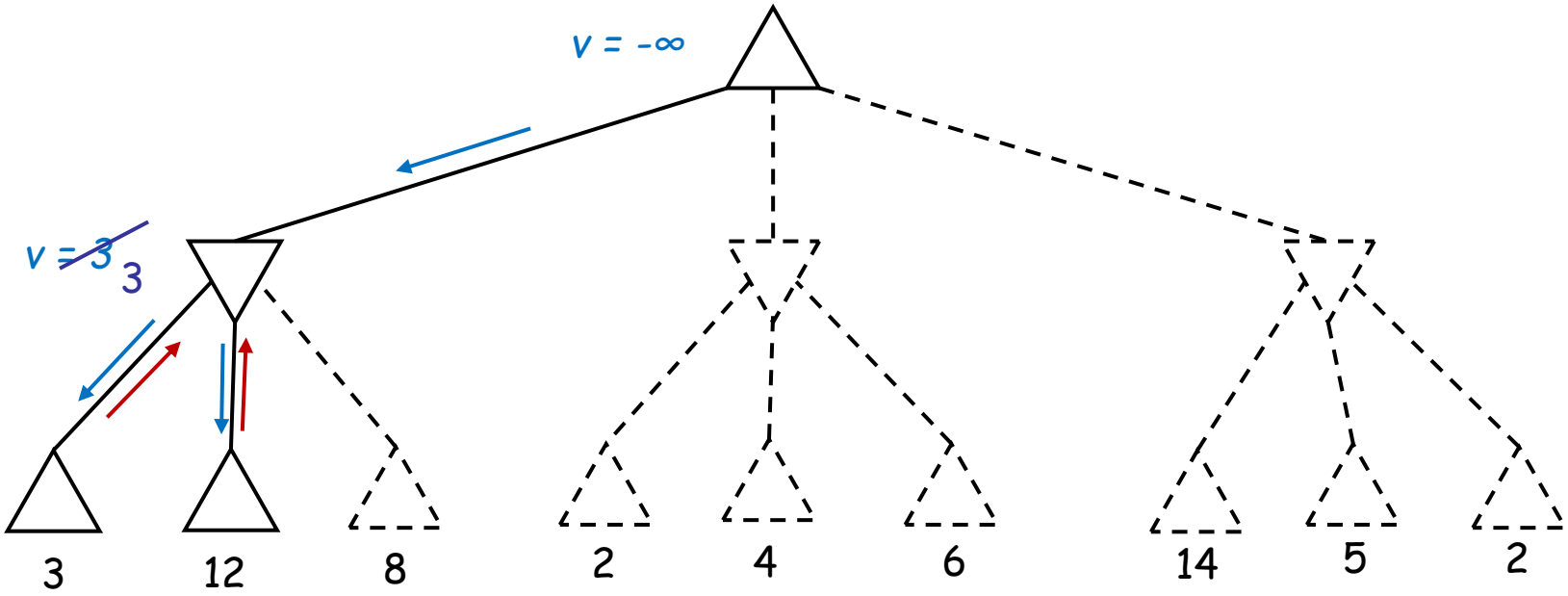
Example



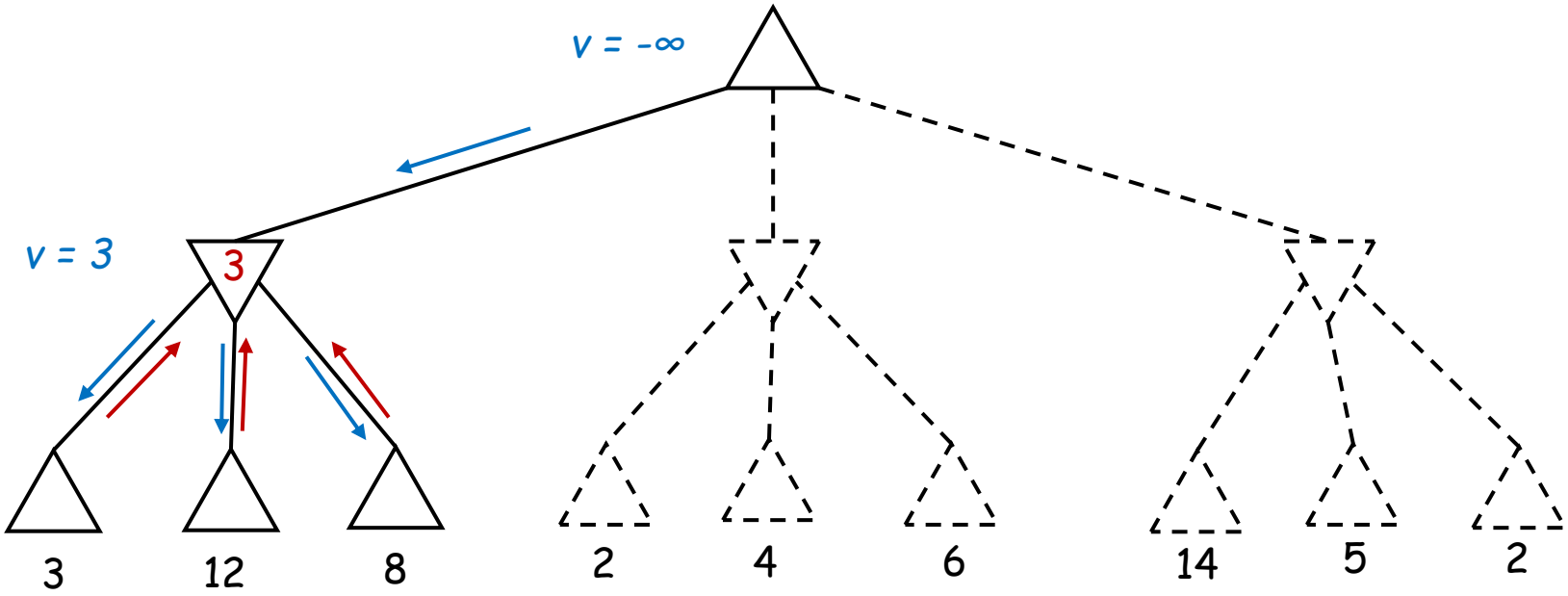
Example



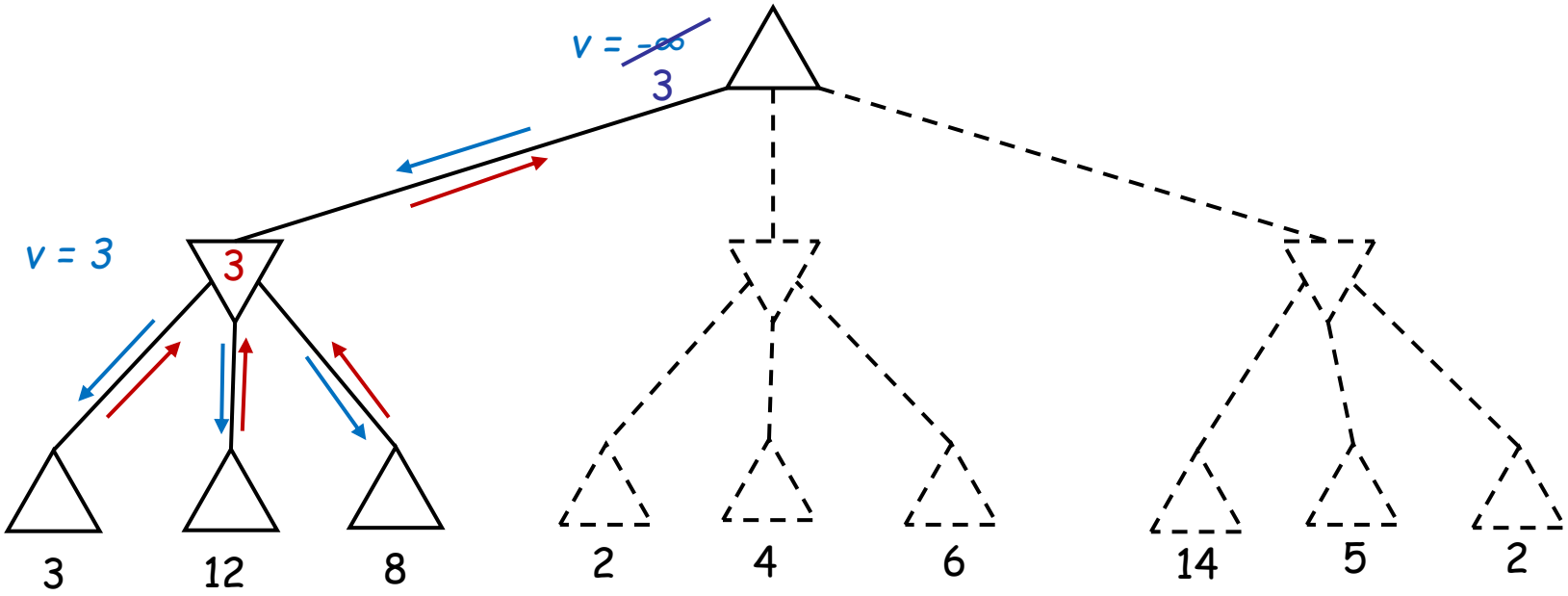
Example



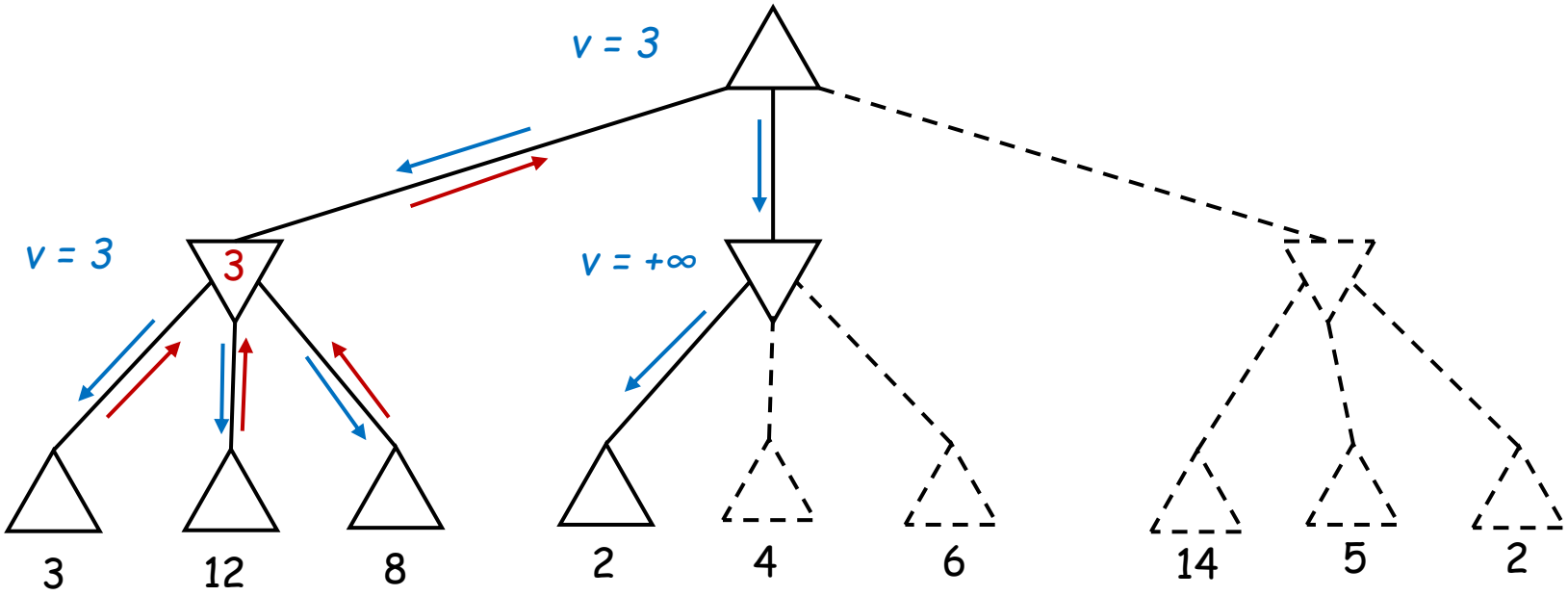
Example



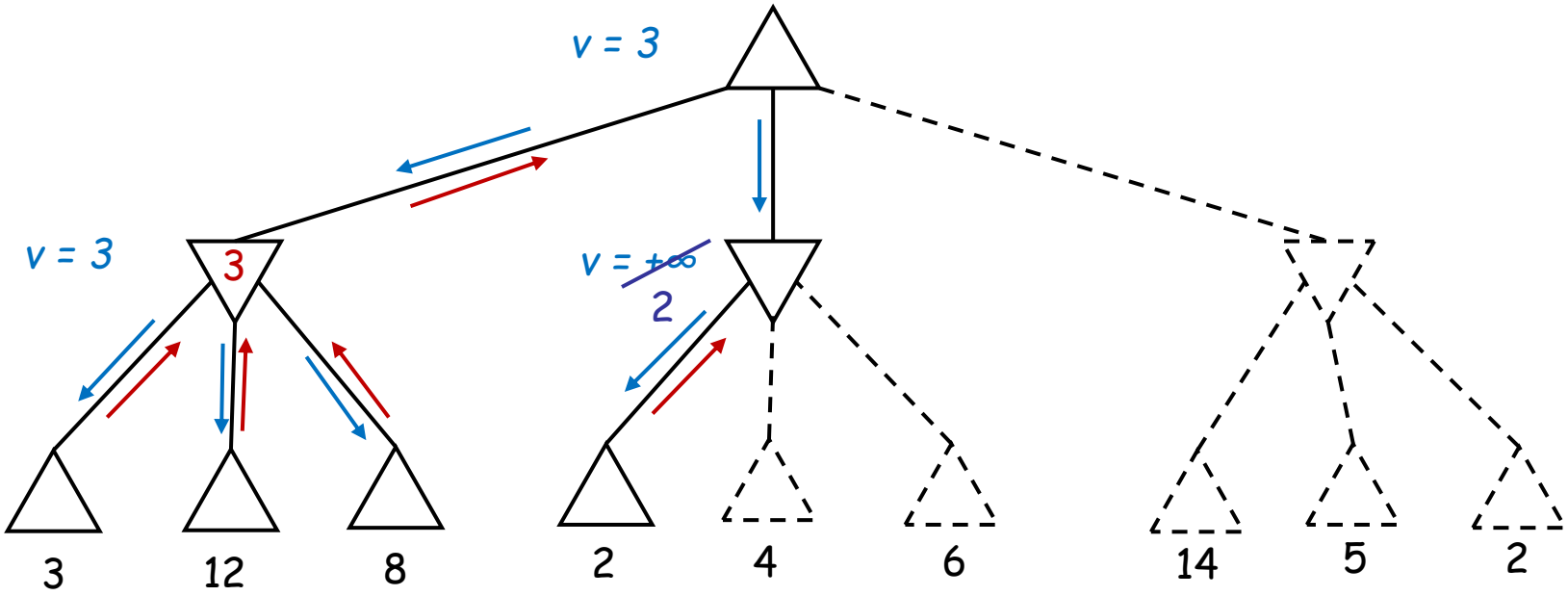
Example



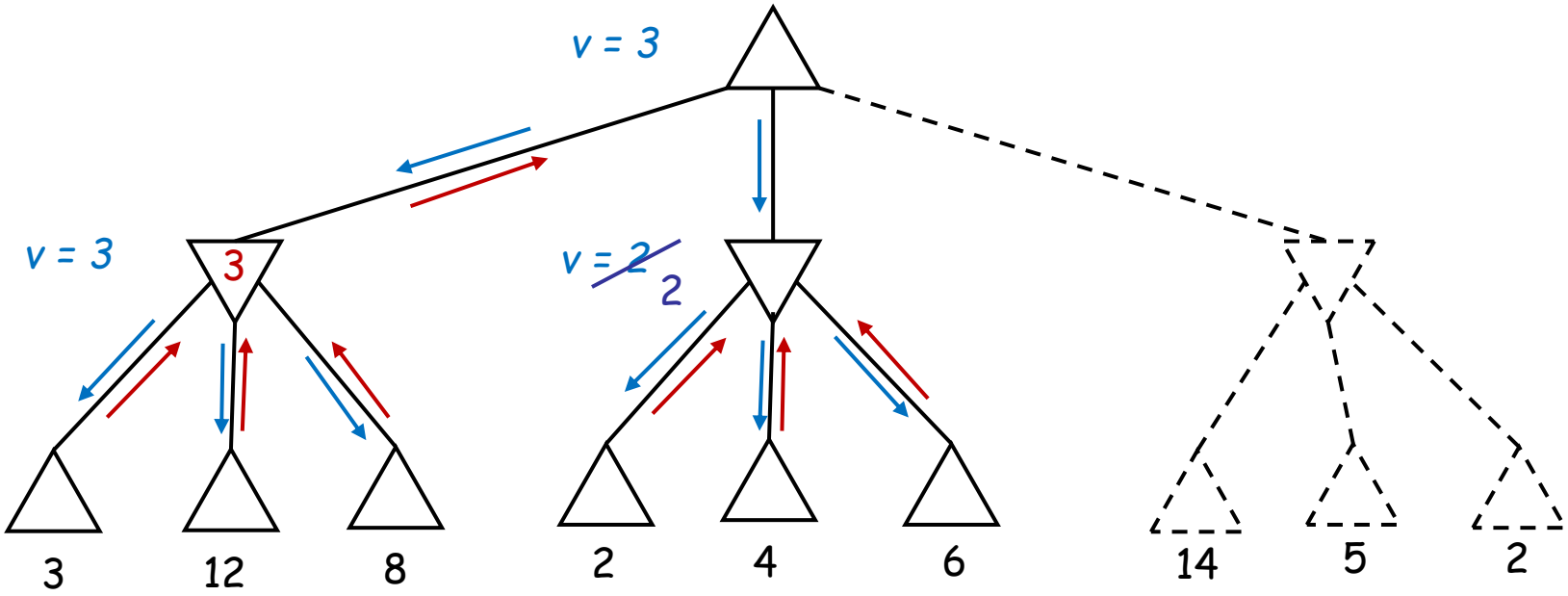
Example



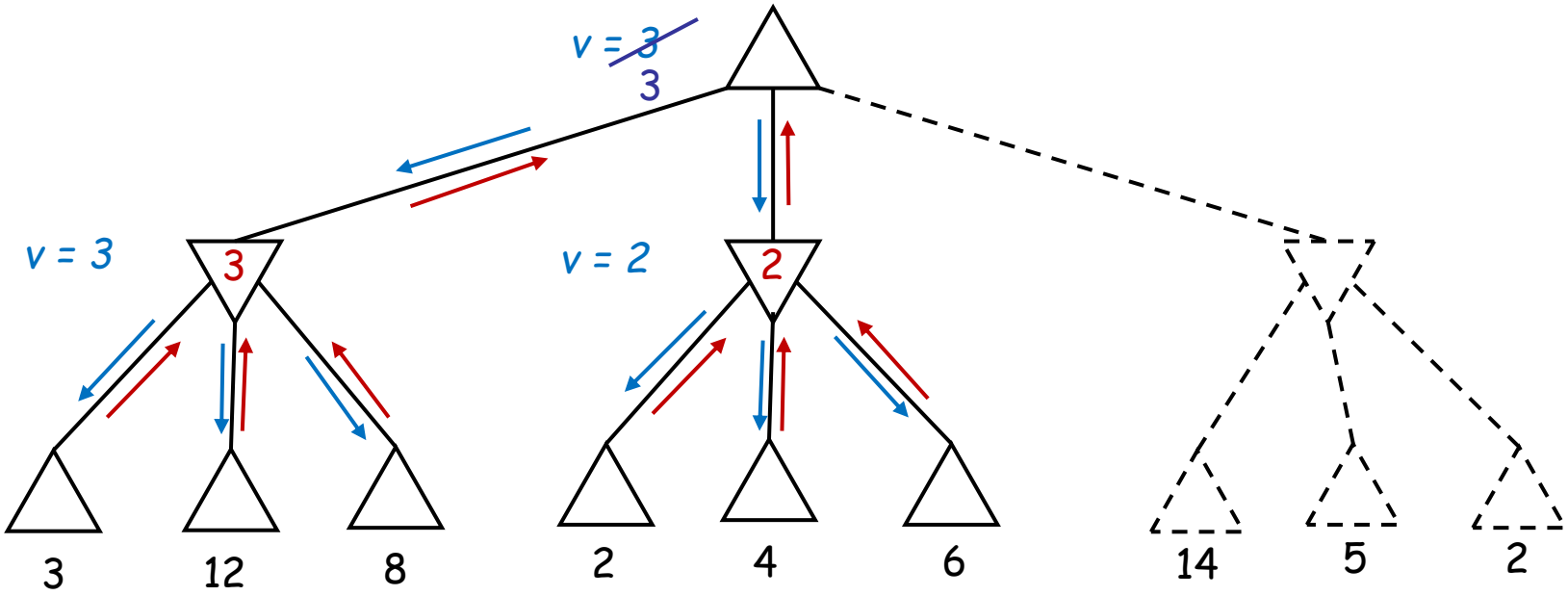
Example



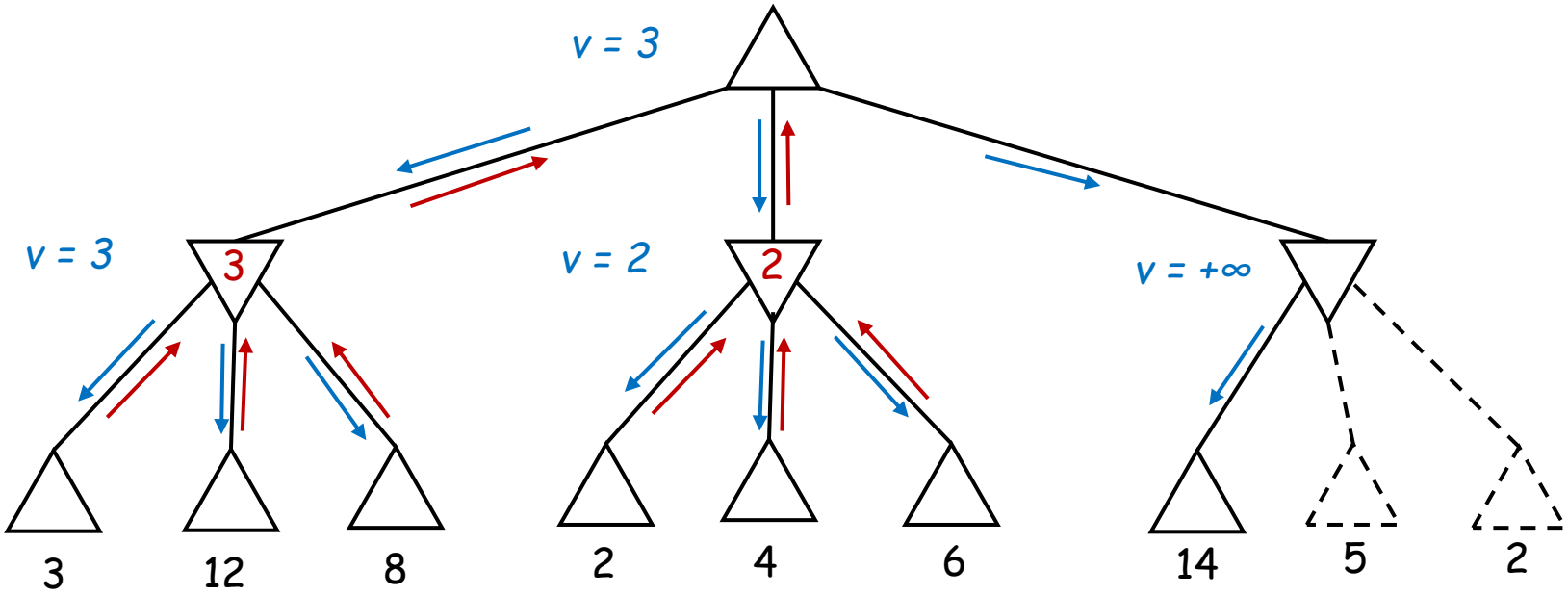
Example



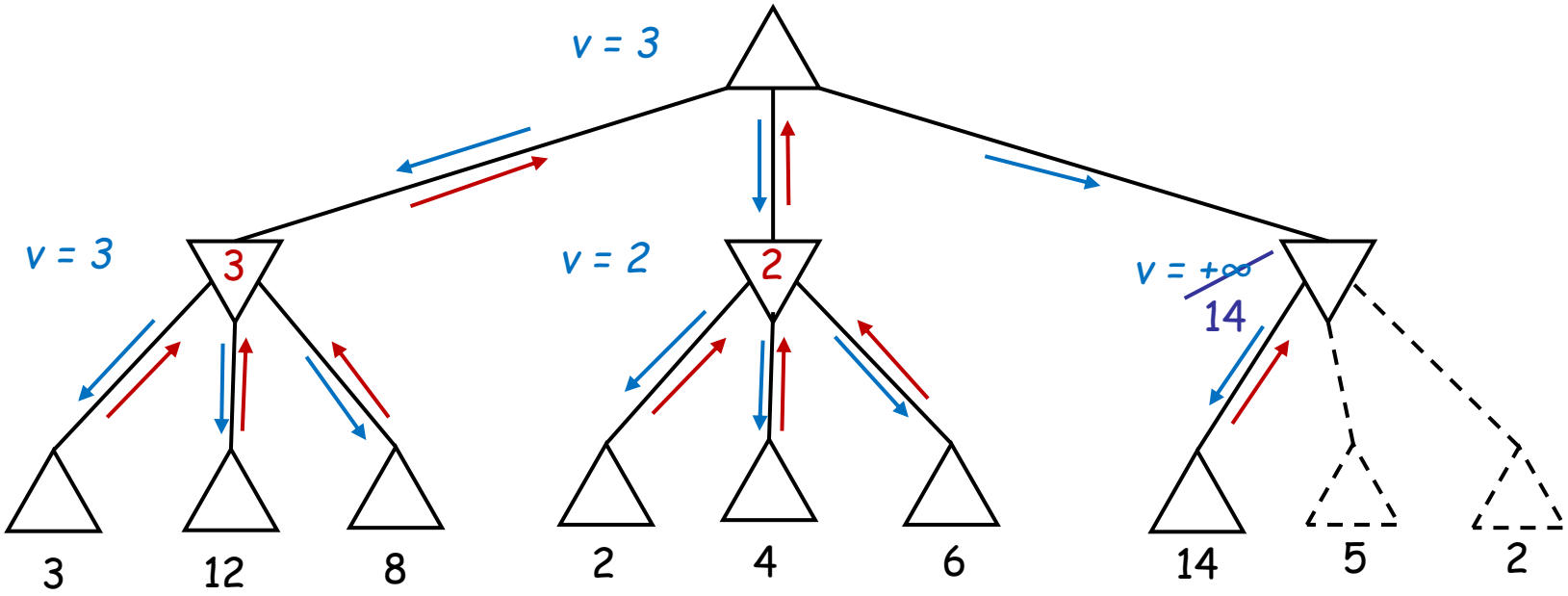
Example



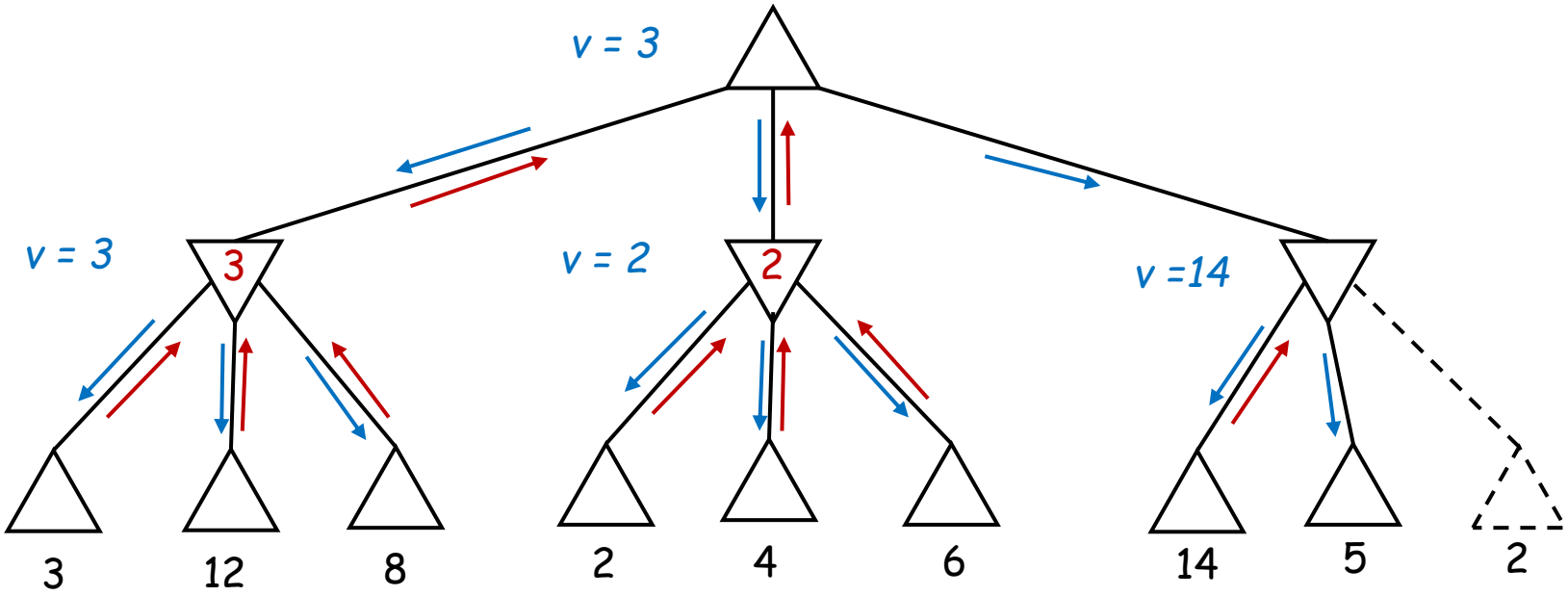
Example



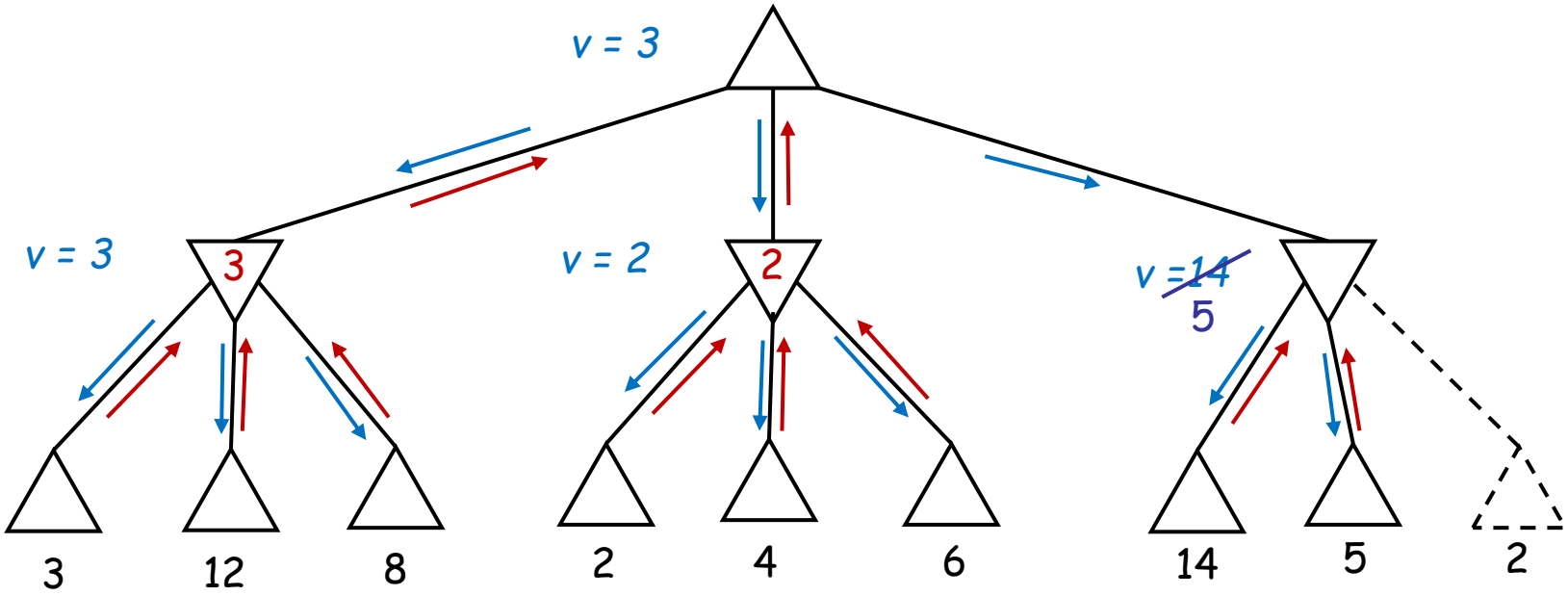
Example



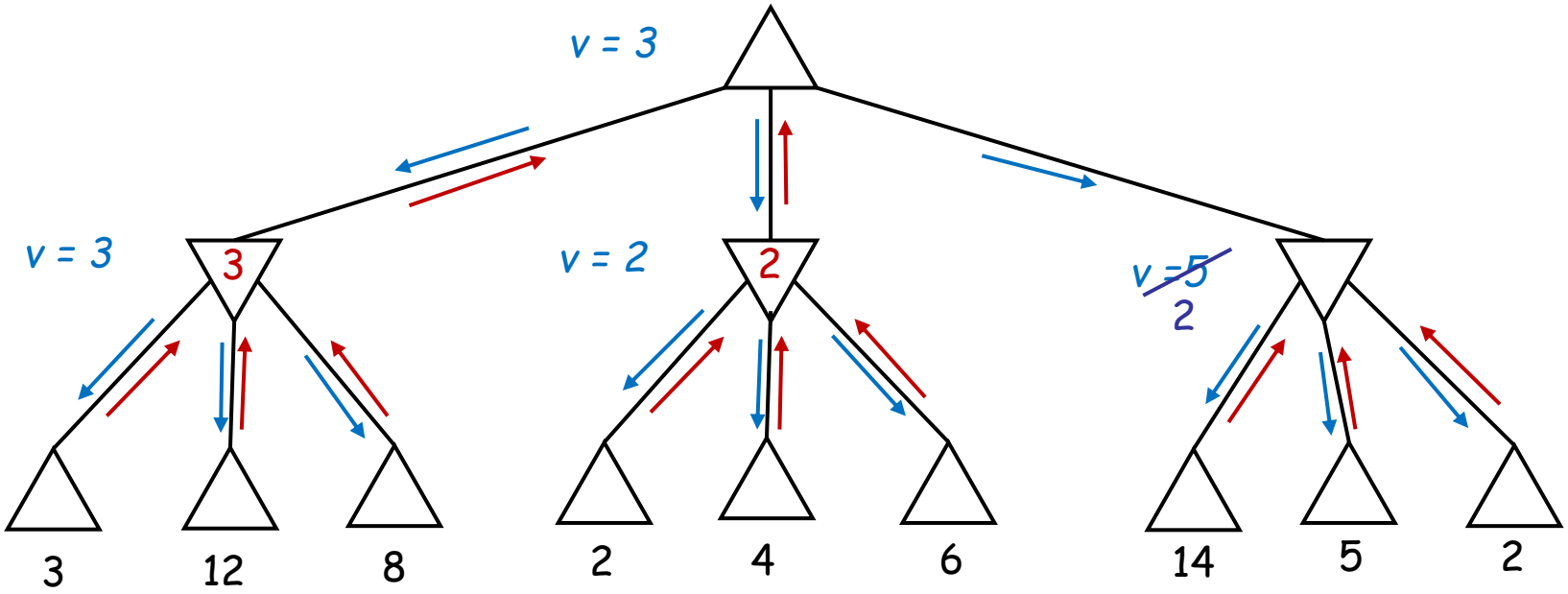
Example



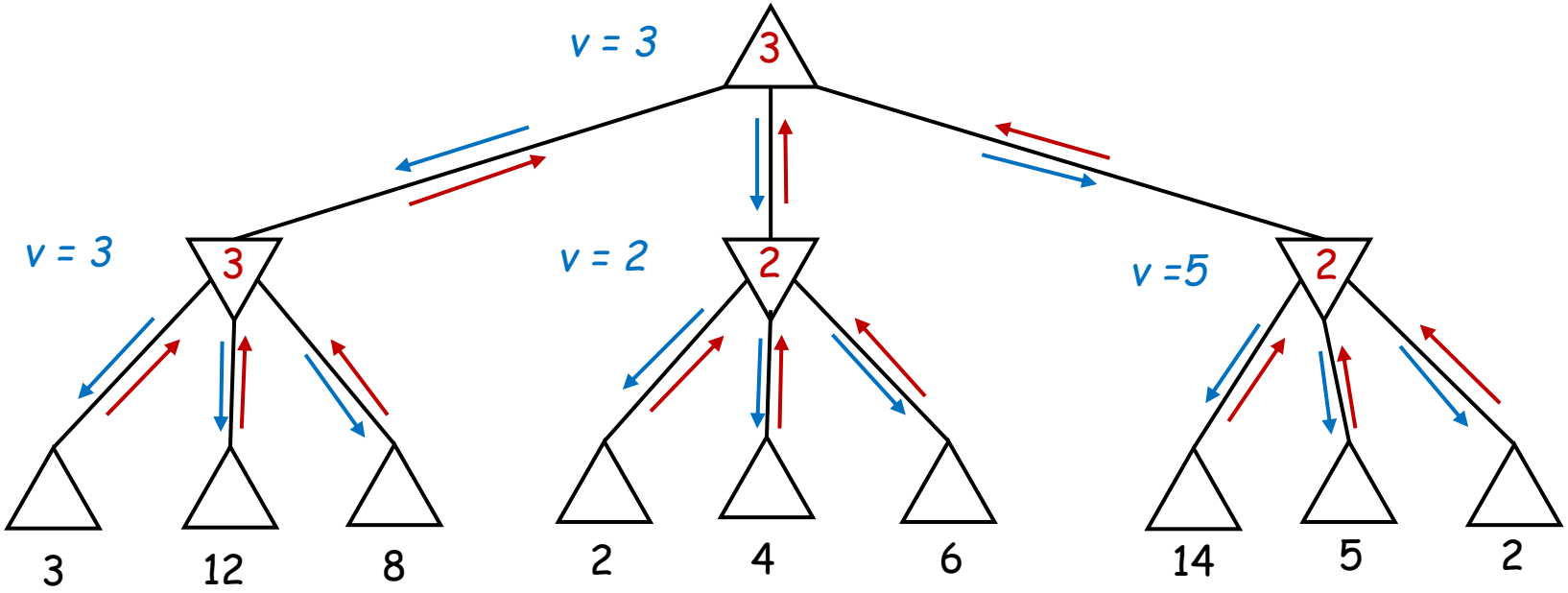
Example



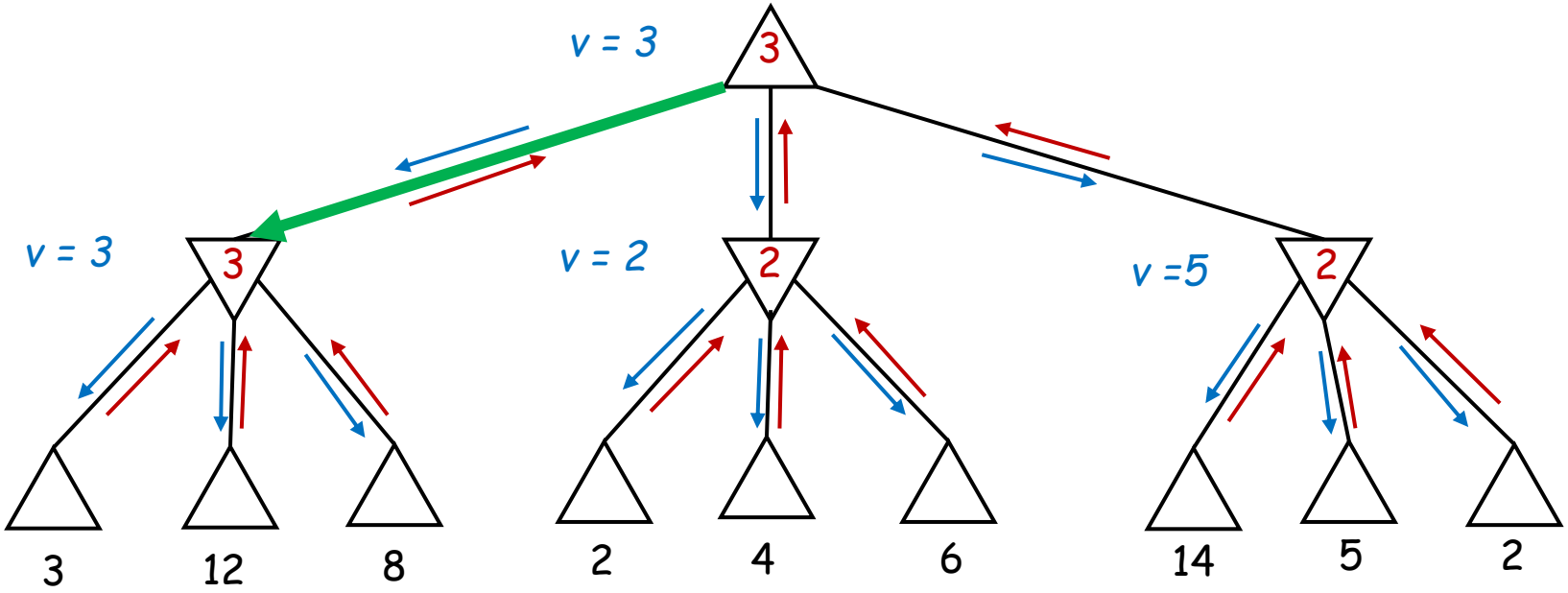
Example



Example



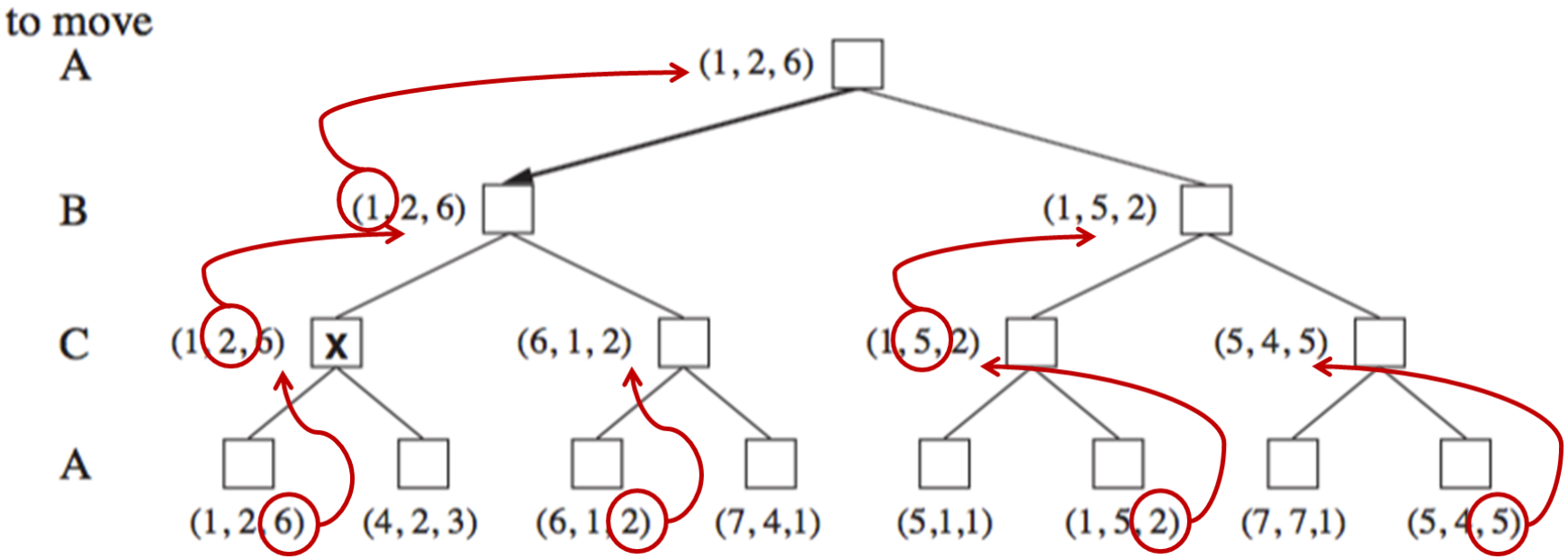
Example



Properties of minimax

- Complete?
  - Yes (when tree is finite)
- Optimal?
  - Yes (against an optimal opponent)
- Time complexity:
  - $O(b^m)$
- Space complexity:
  - O(bm) (depth-first exploration)

Multiplayer games

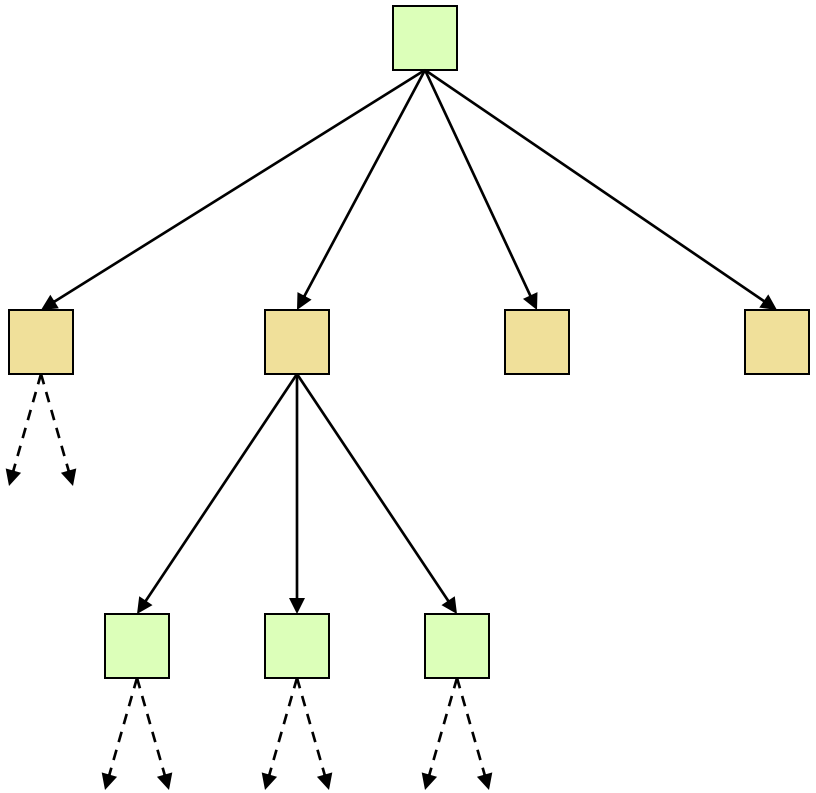


Can we do better?

Yes ! Much better !

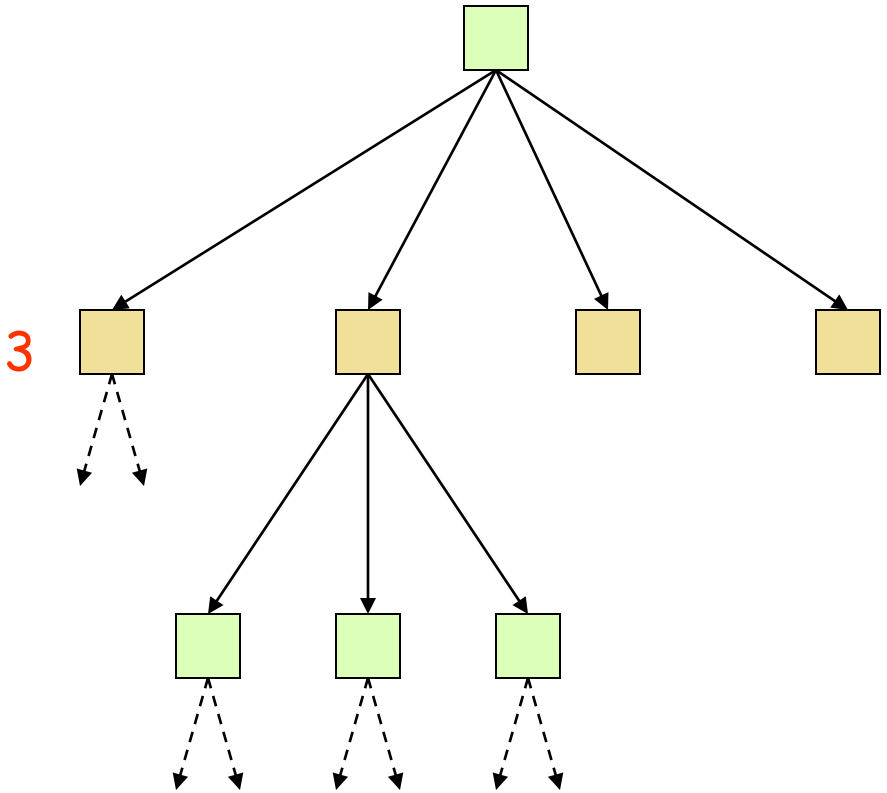
Can we do better?

Yes ! Much better !



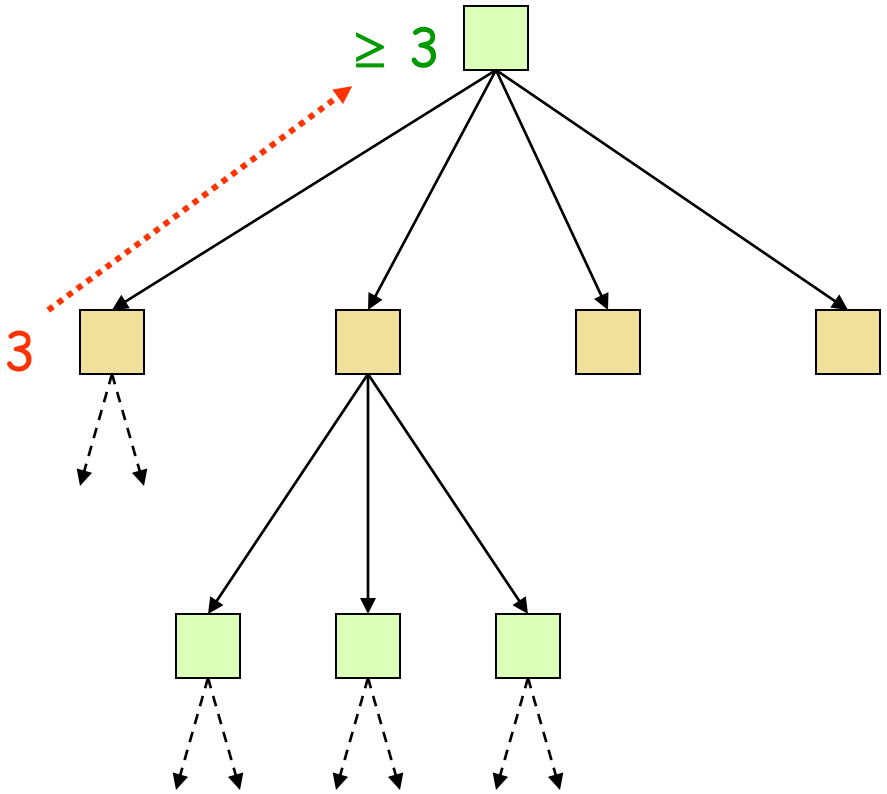
Can we do better?

Yes ! Much better !



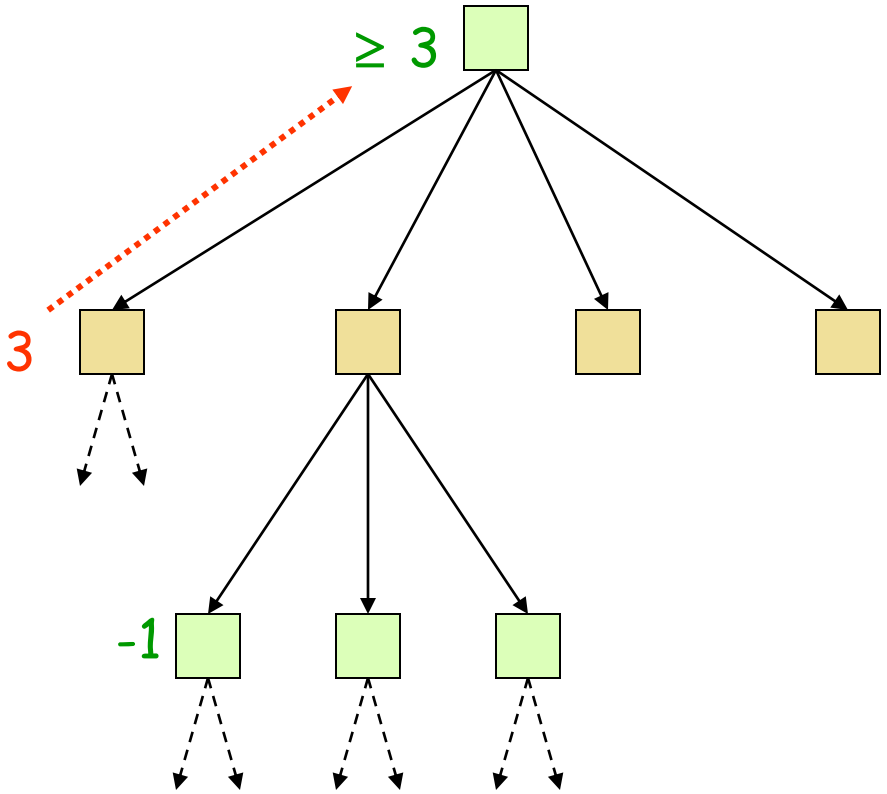
Can we do better?

Yes ! Much better !



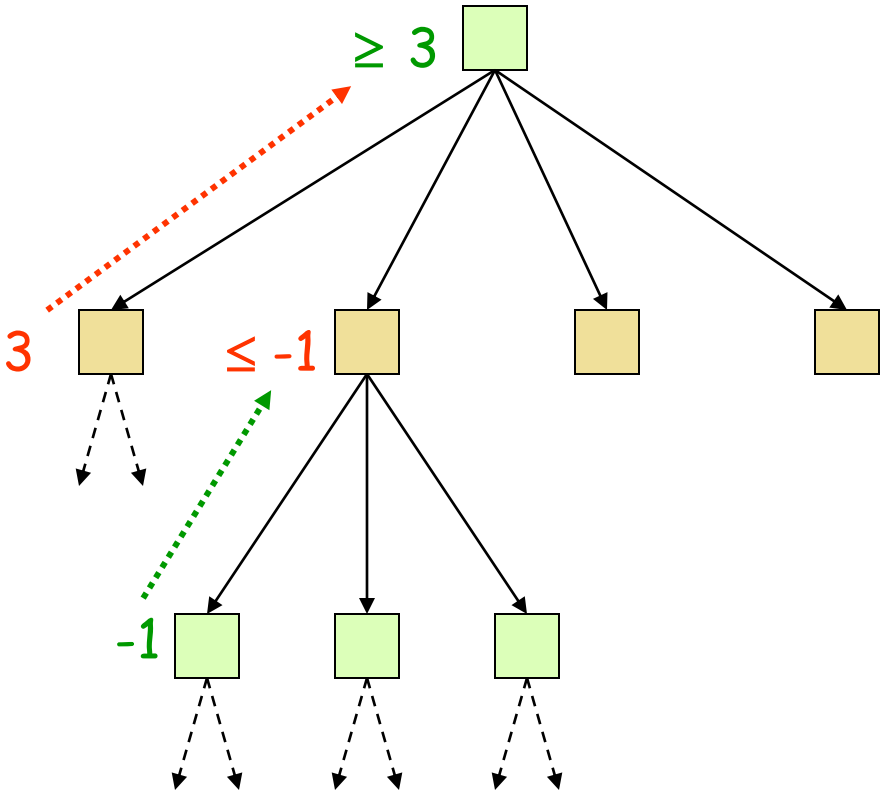
Can we do better?

Yes ! Much better !



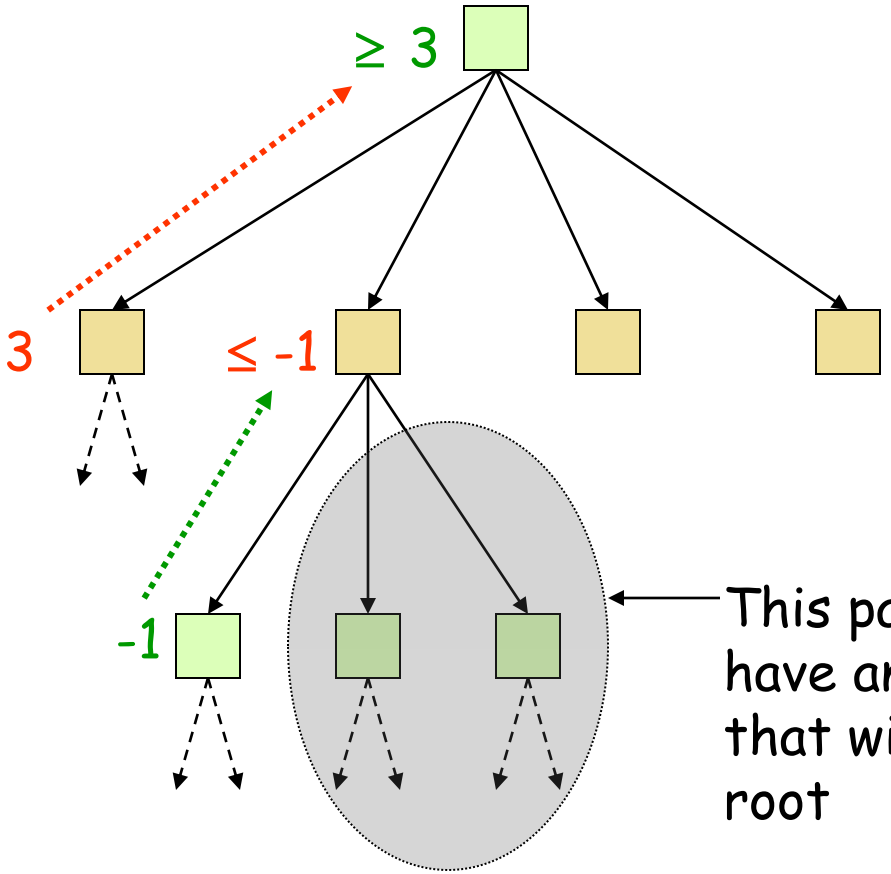
Can we do better?

Yes ! Much better !



Can we do better?

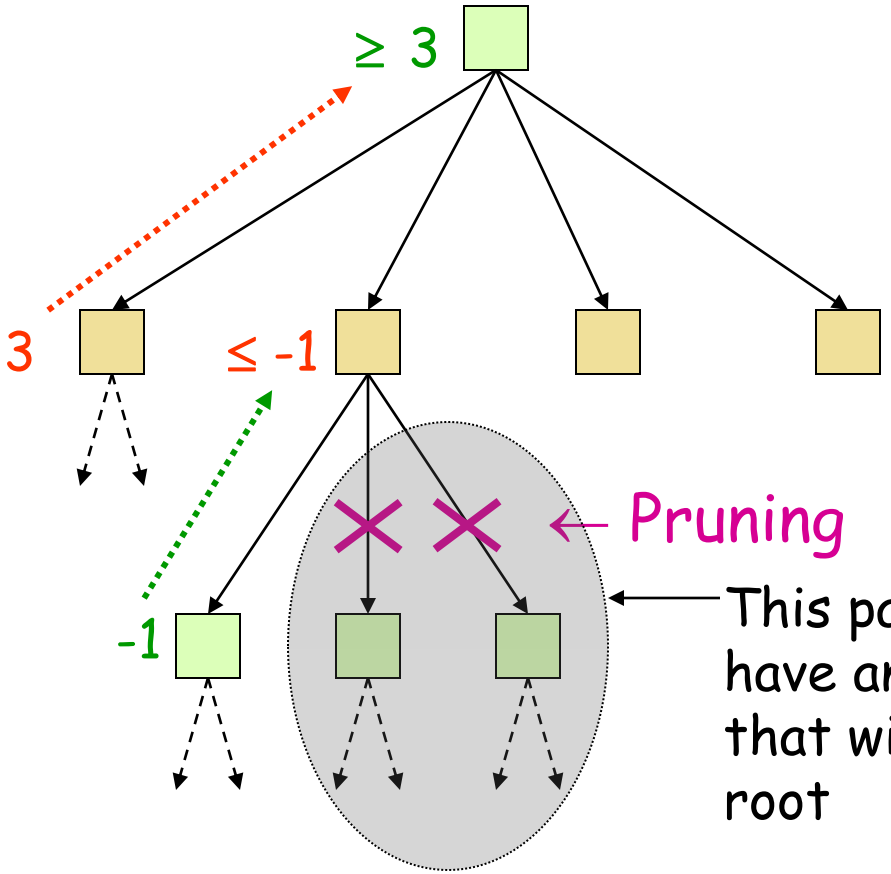
Yes ! Much better !



This part of the tree can't have any effect on the value that will be backed up to the root

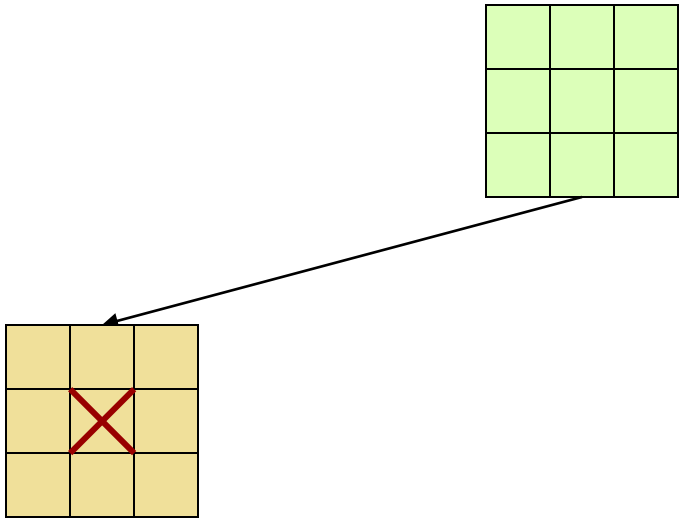
Can we do better?

Yes ! Much better !

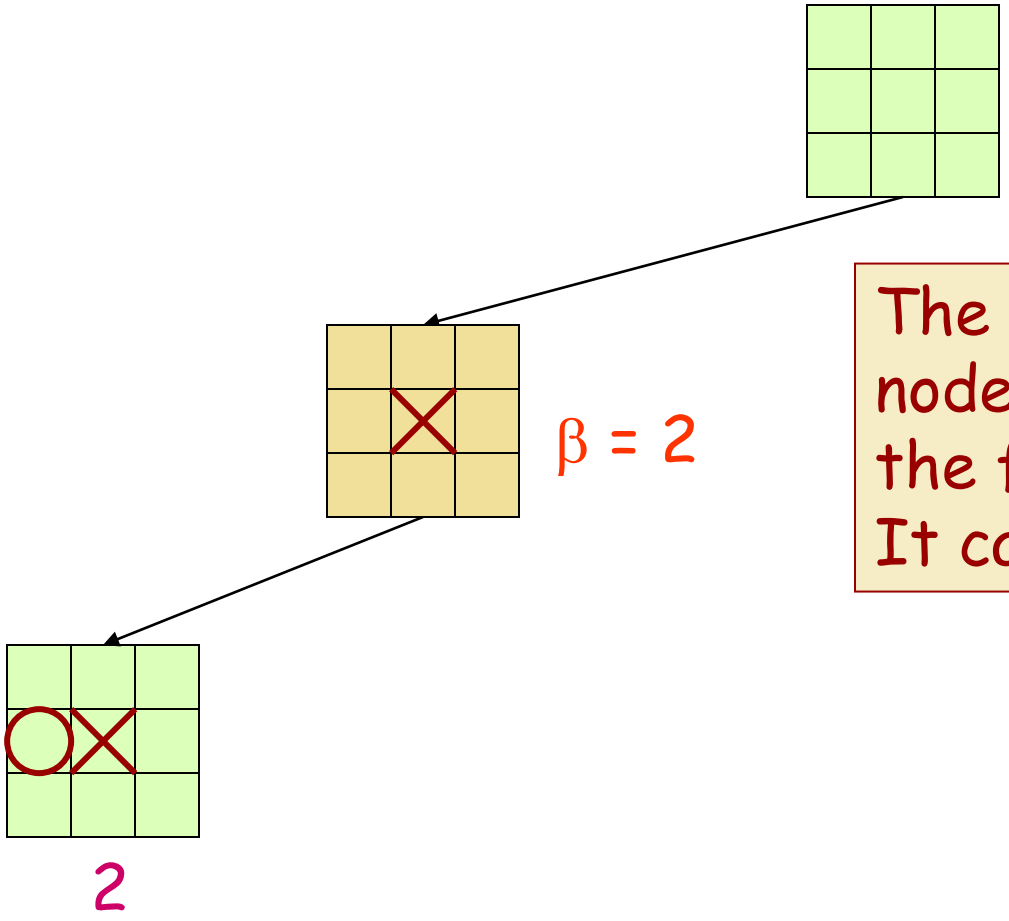


This part of the tree can't have any effect on the value that will be backed up to the root

Example

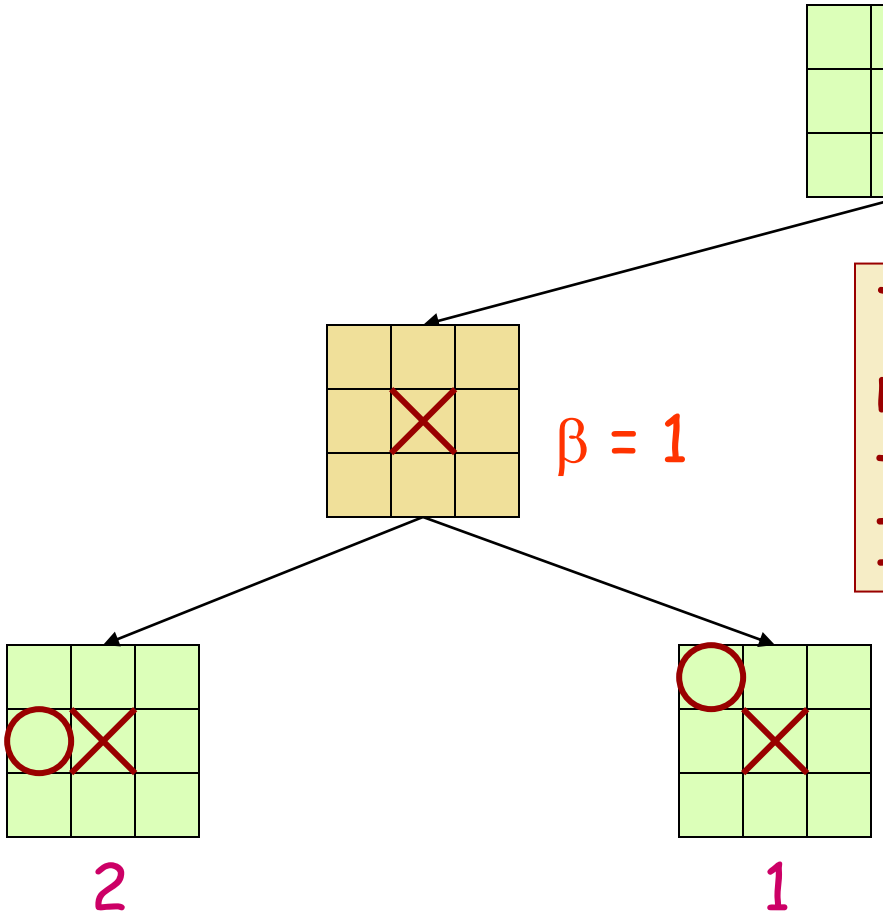


Example



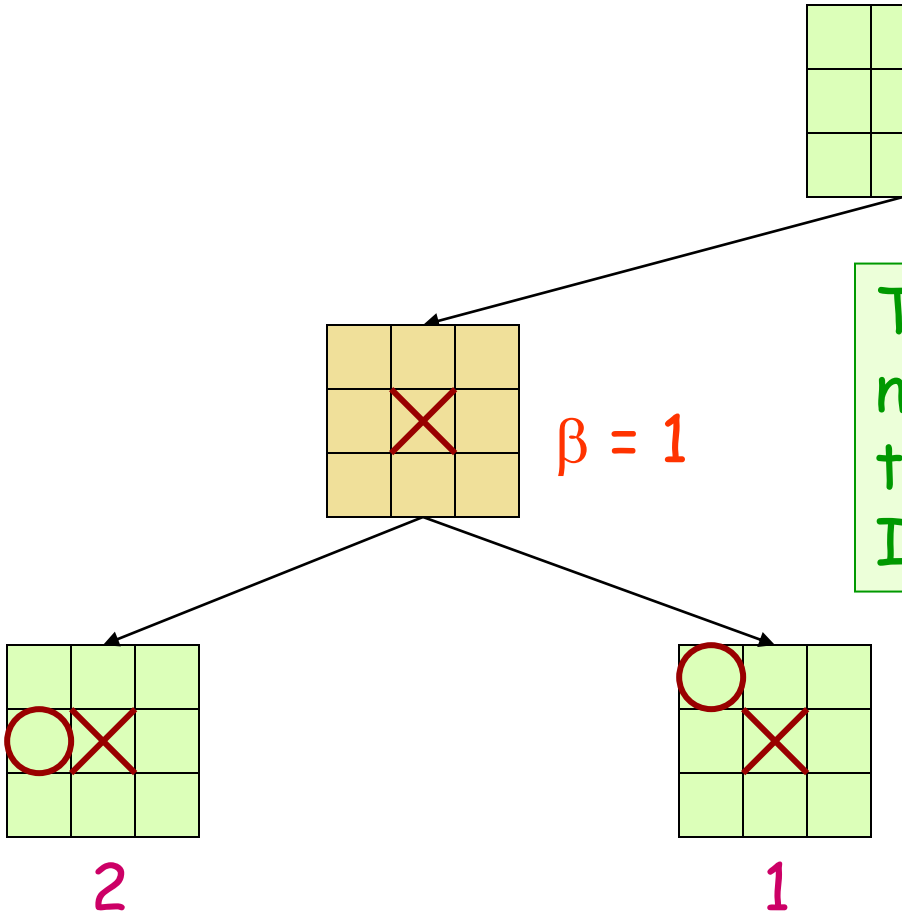
The beta value of a MIN node is an upper bound on the final backed-up value. It can never increase

Example



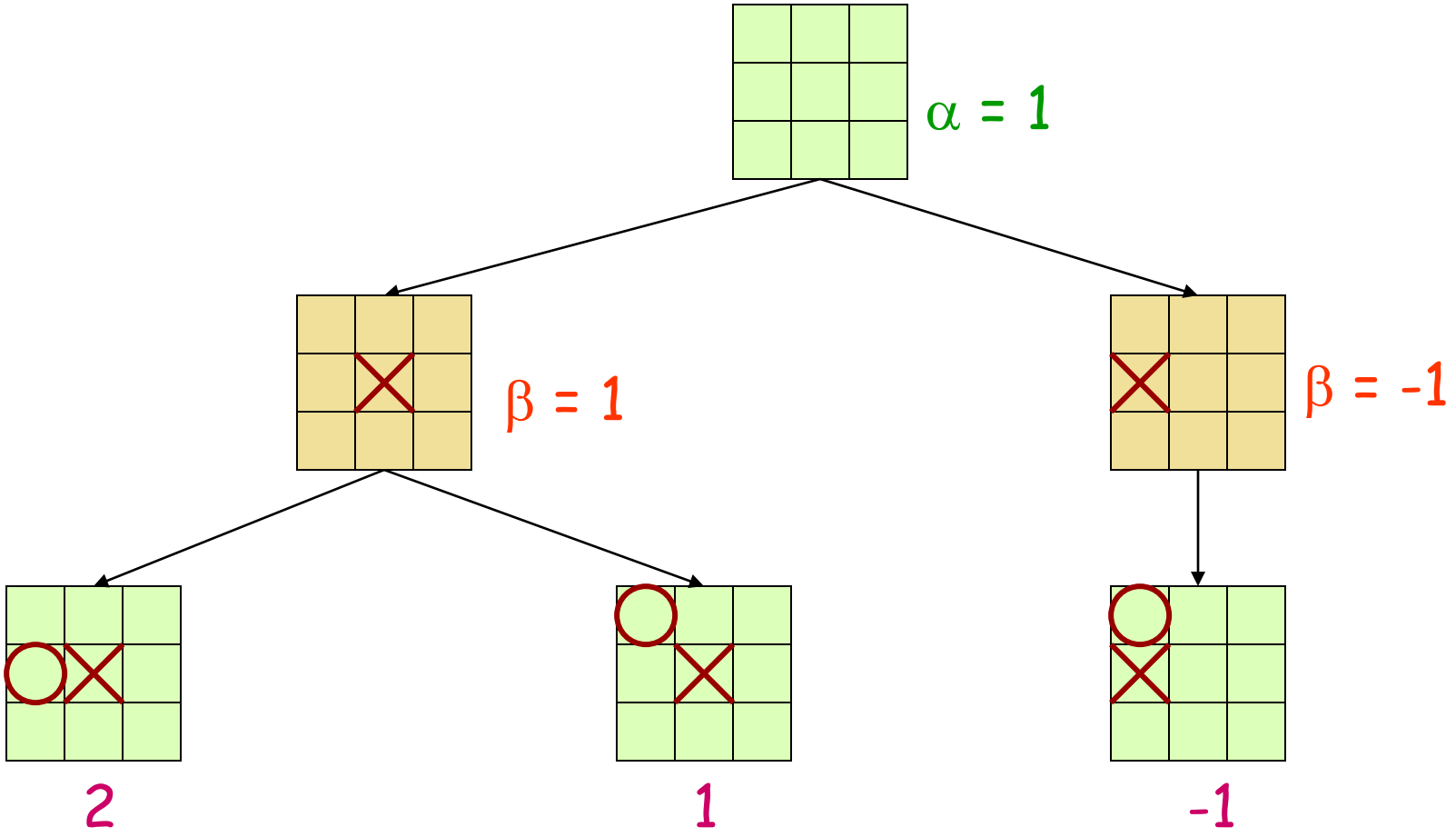
The beta value of a MIN node is an upper bound on the final backed-up value. It can never increase

Example

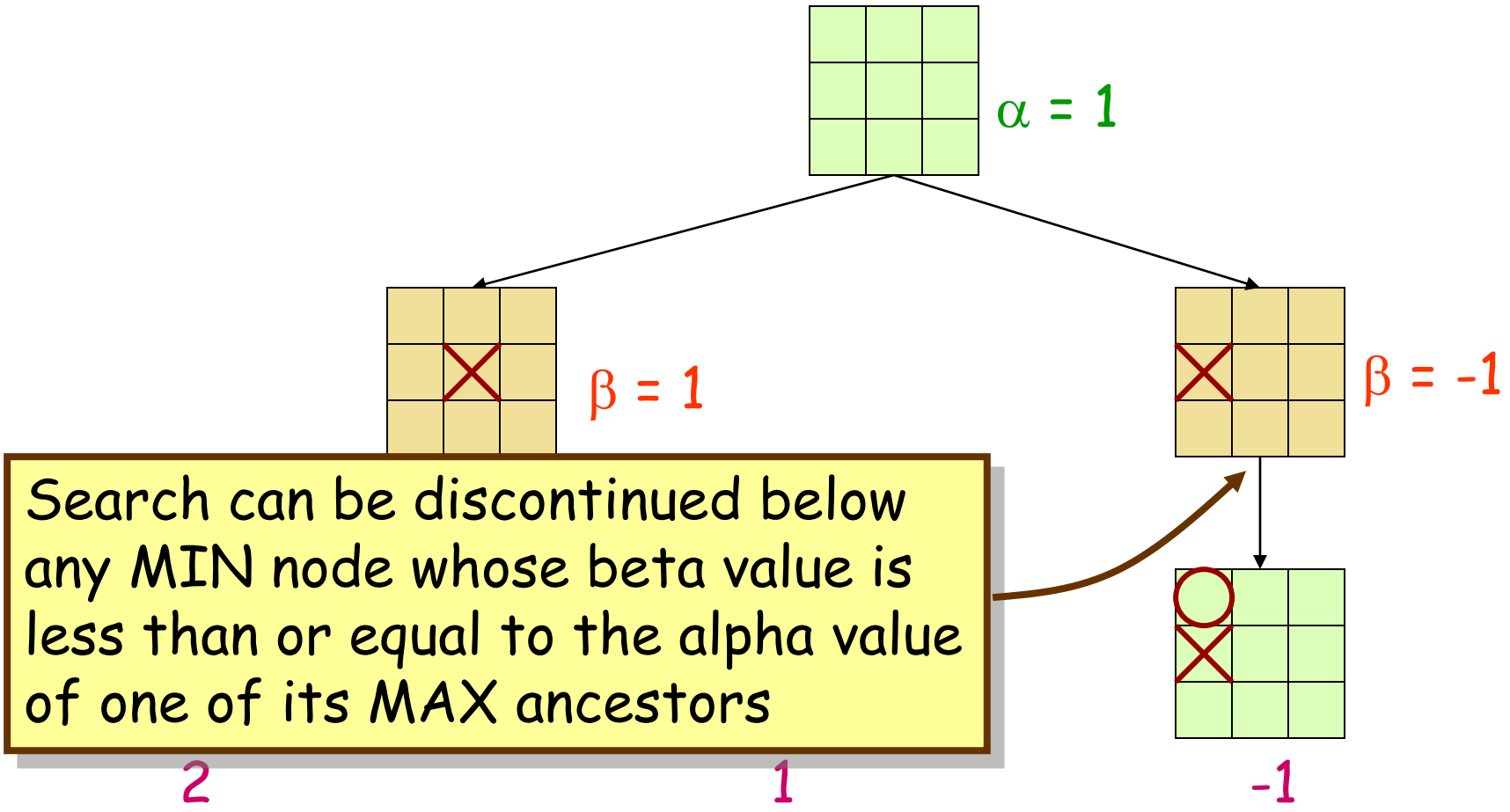


The alpha value of a MAX node is a lower bound on the final backed-up value. It can never decrease

Example



Example

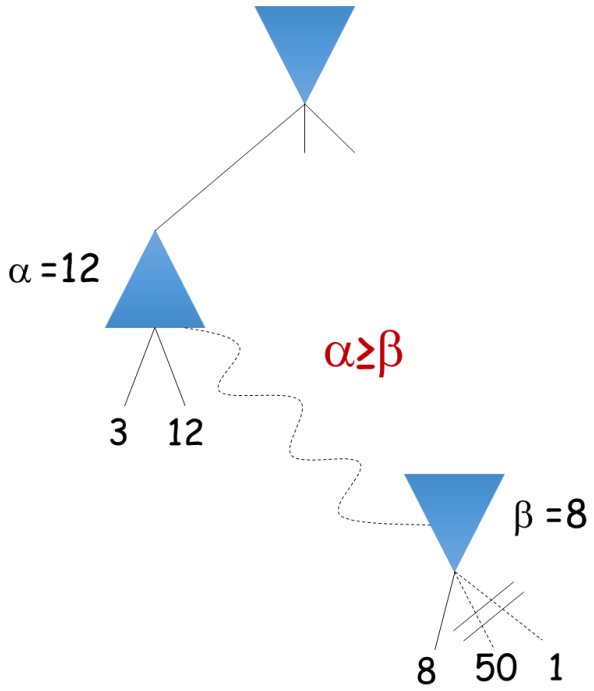

α = 1
β = 1
β = -1
Search can be discontinued below any MIN node whose beta value is less than or equal to the alpha value of one of its MAX ancestors
2
1
-1

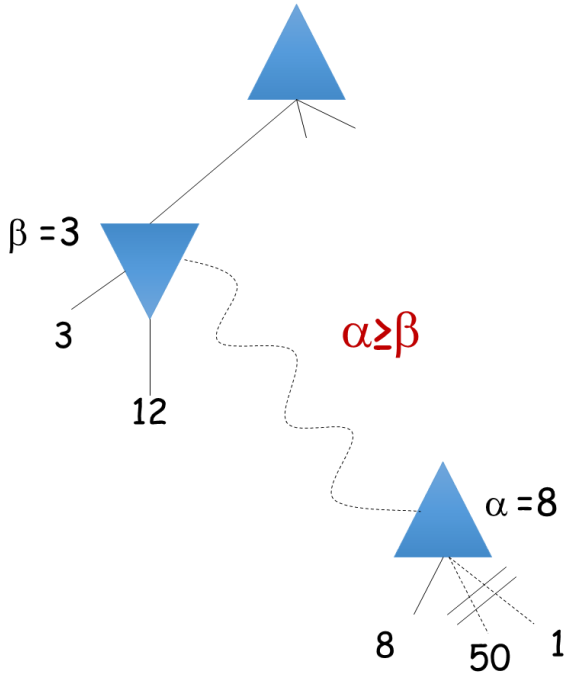
Alpha-Beta Pruning

- Explore the game tree to depth h in depth-first manner
- Back up alpha and beta values whenever possible
- Prune branches that can't lead to changing the final decision

Alpha-Beta Algorithm

- alpha (α)
  - Value of the best (highest) choice found so far at any choice point along the path for MAX
- beta (β)
  - Value of the best (lowest) choice found so far at any choice point along the path for MIN

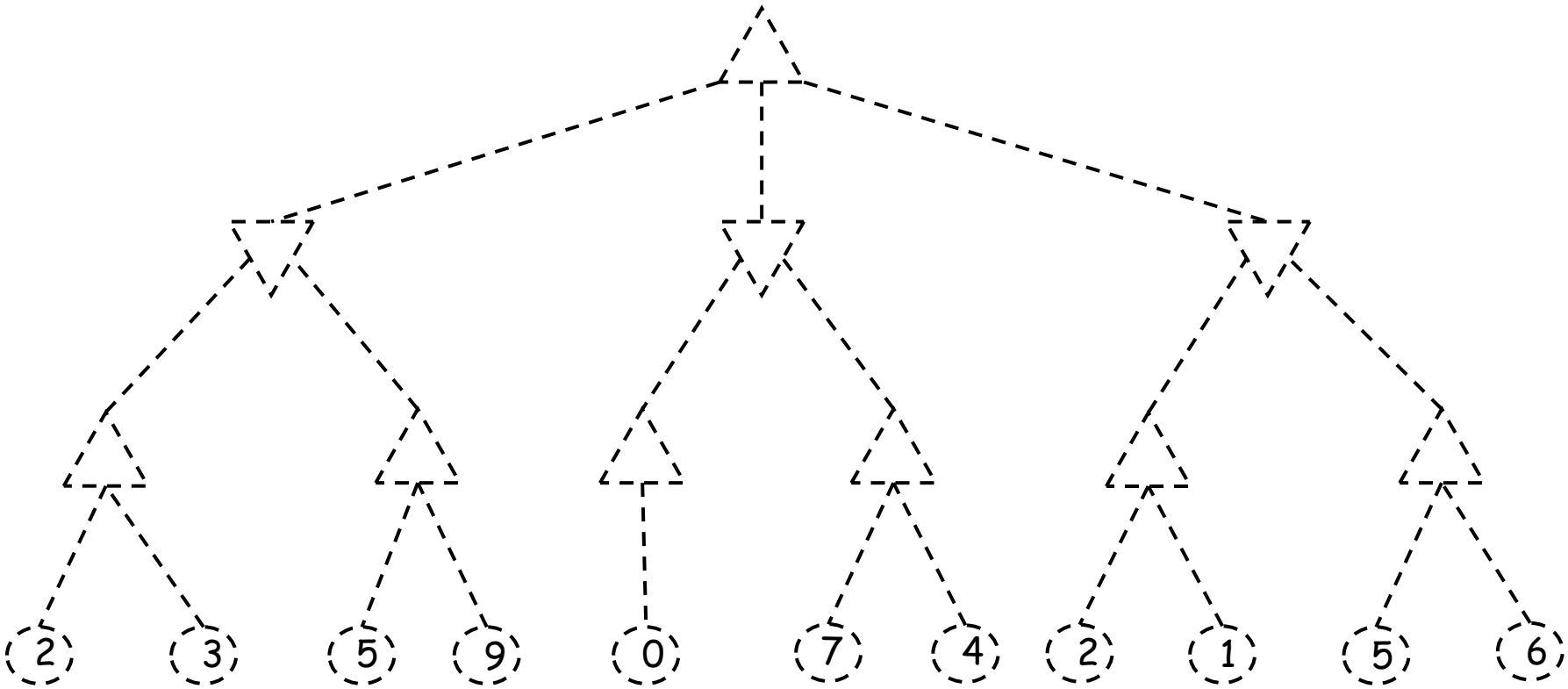




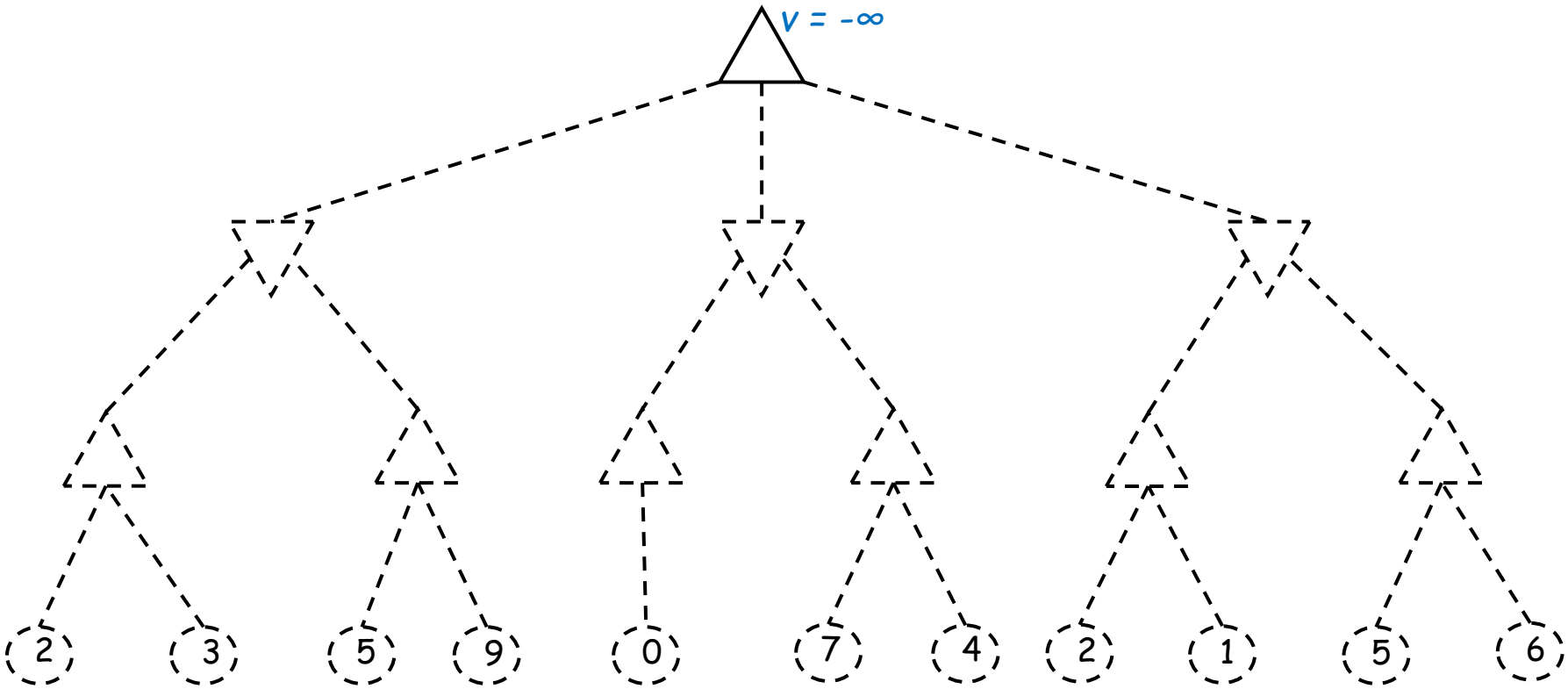
Alpha-Beta Algorithm

- Update the alpha/beta value of the parent of a node N when the search below N has been completed or discontinued
- Discontinue the search below a MAX node N if its alpha value is ≥ the beta value of a MIN ancestor of N
- Discontinue the search below a MIN node N if its beta value is ≤ the alpha value of a MAX ancestor of N

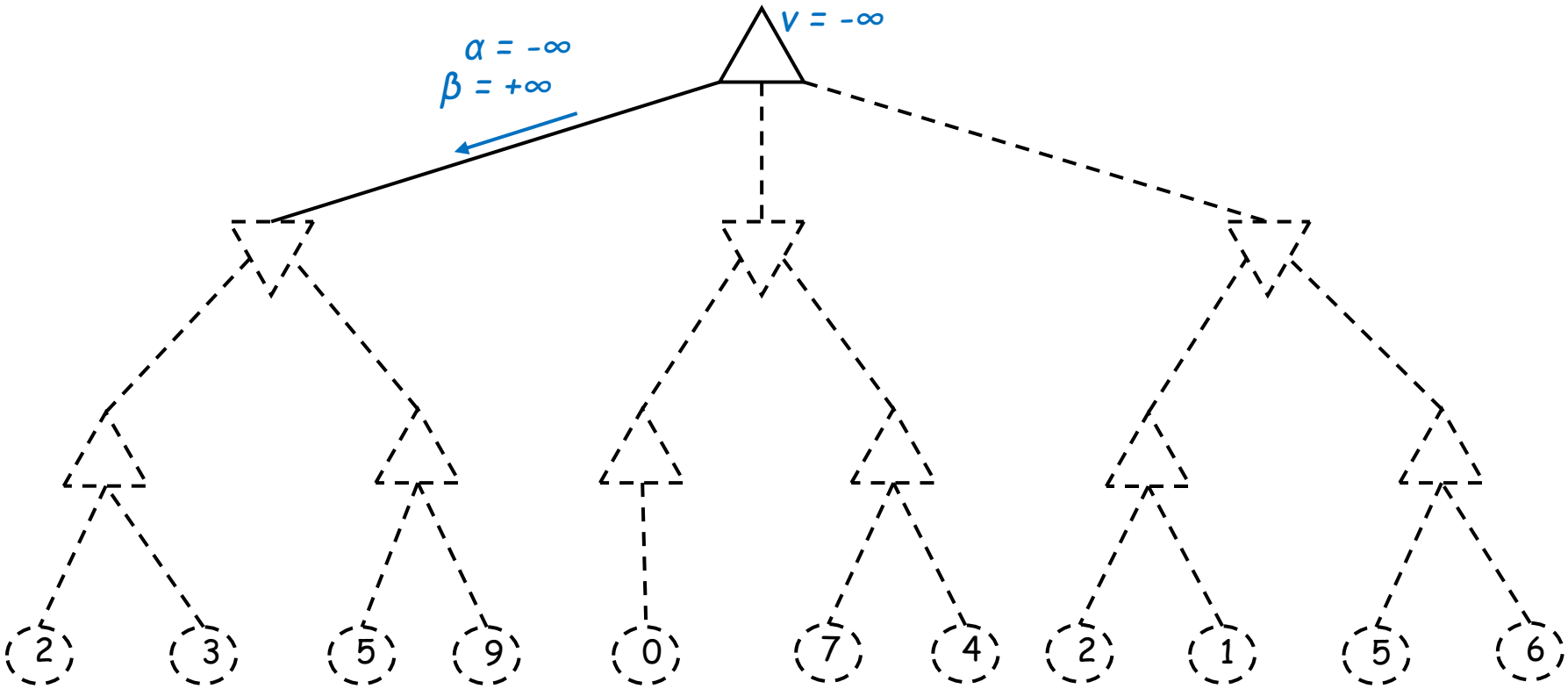
Example



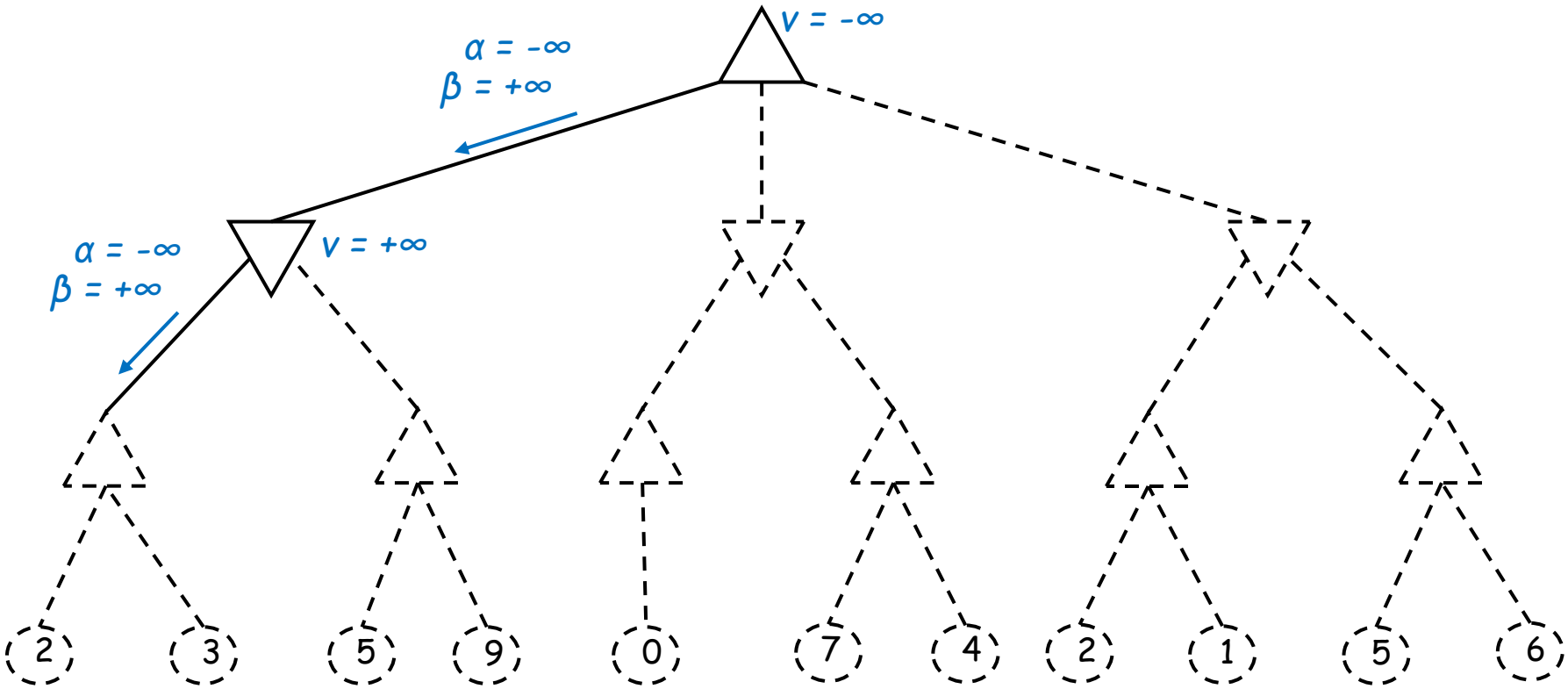
Example



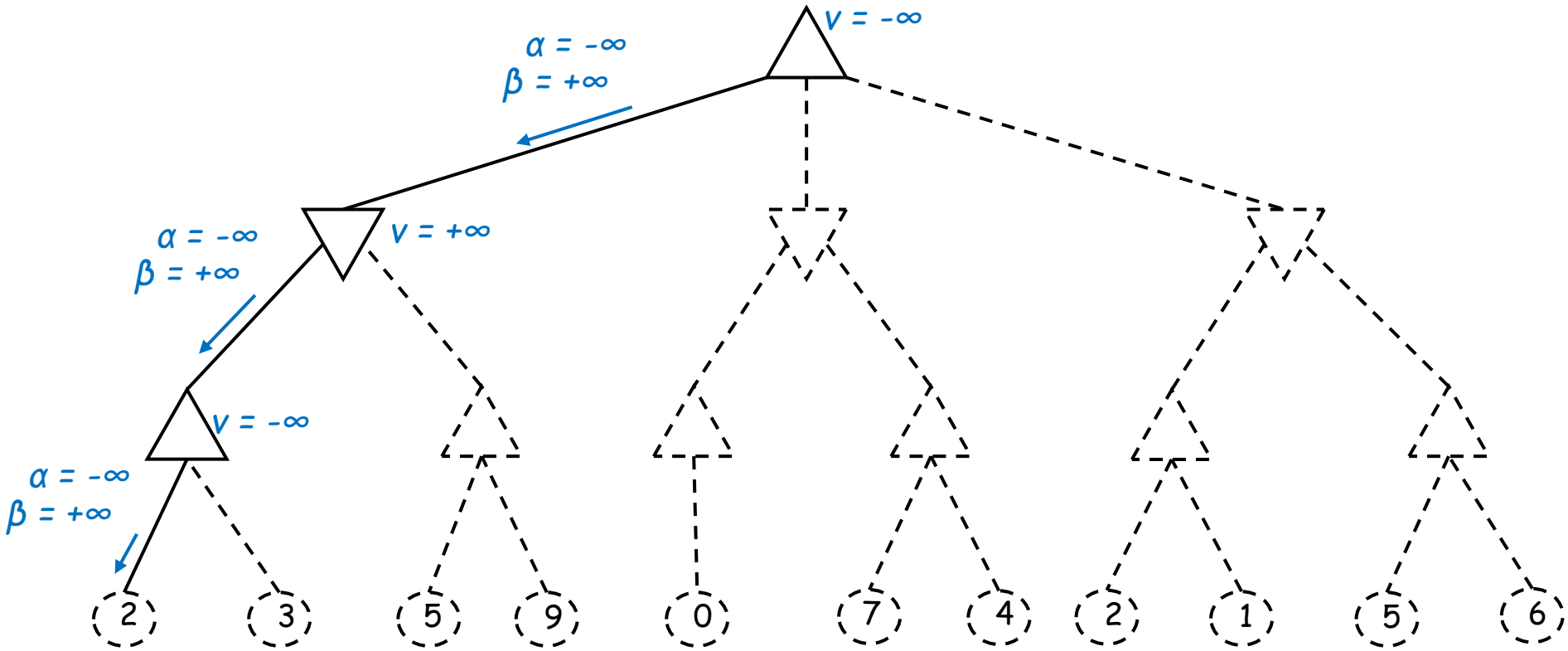
Example



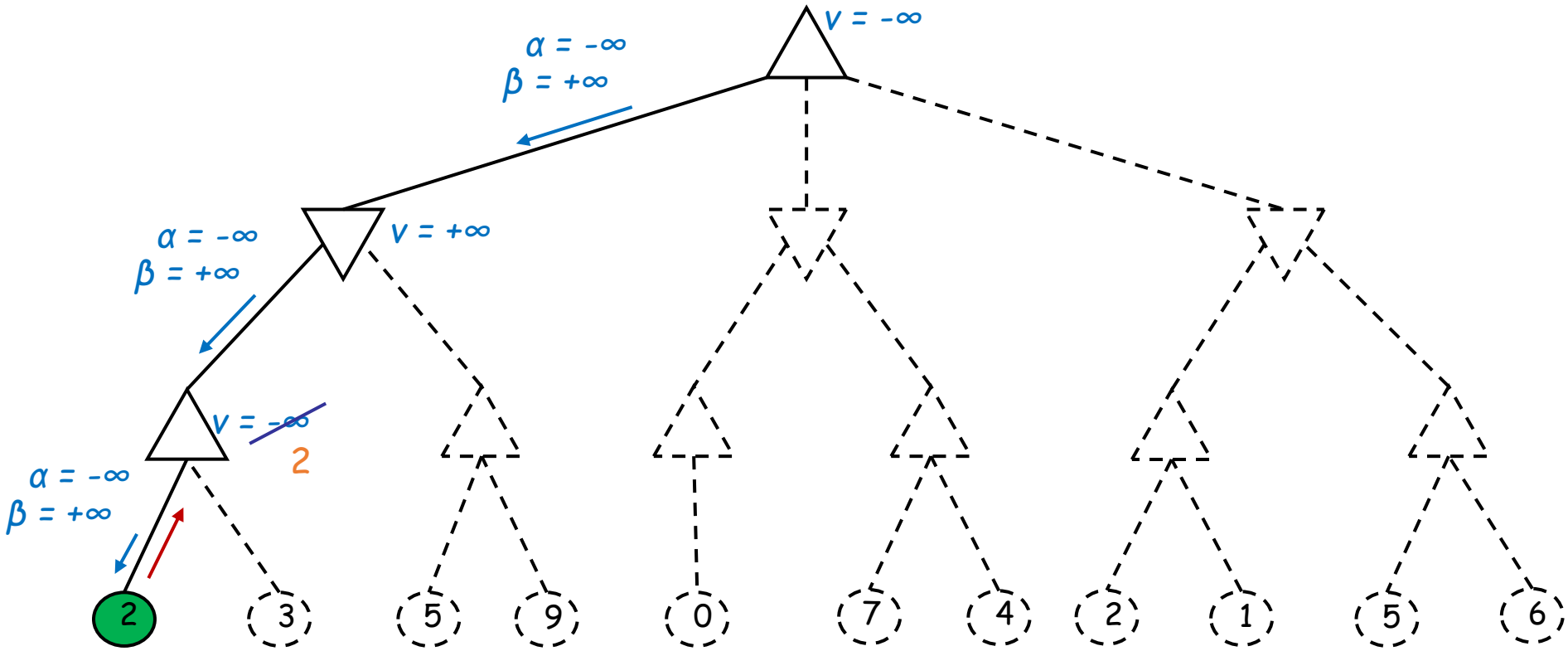
Example



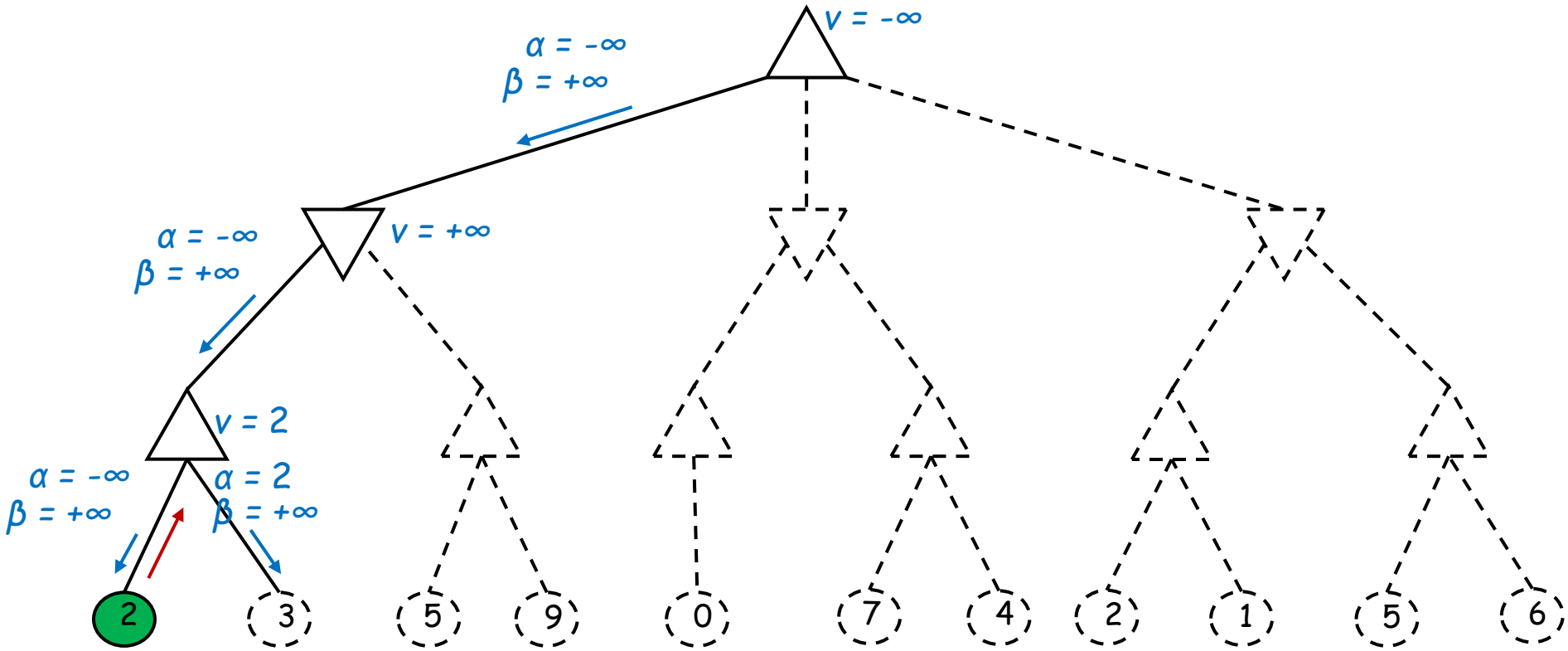
Example



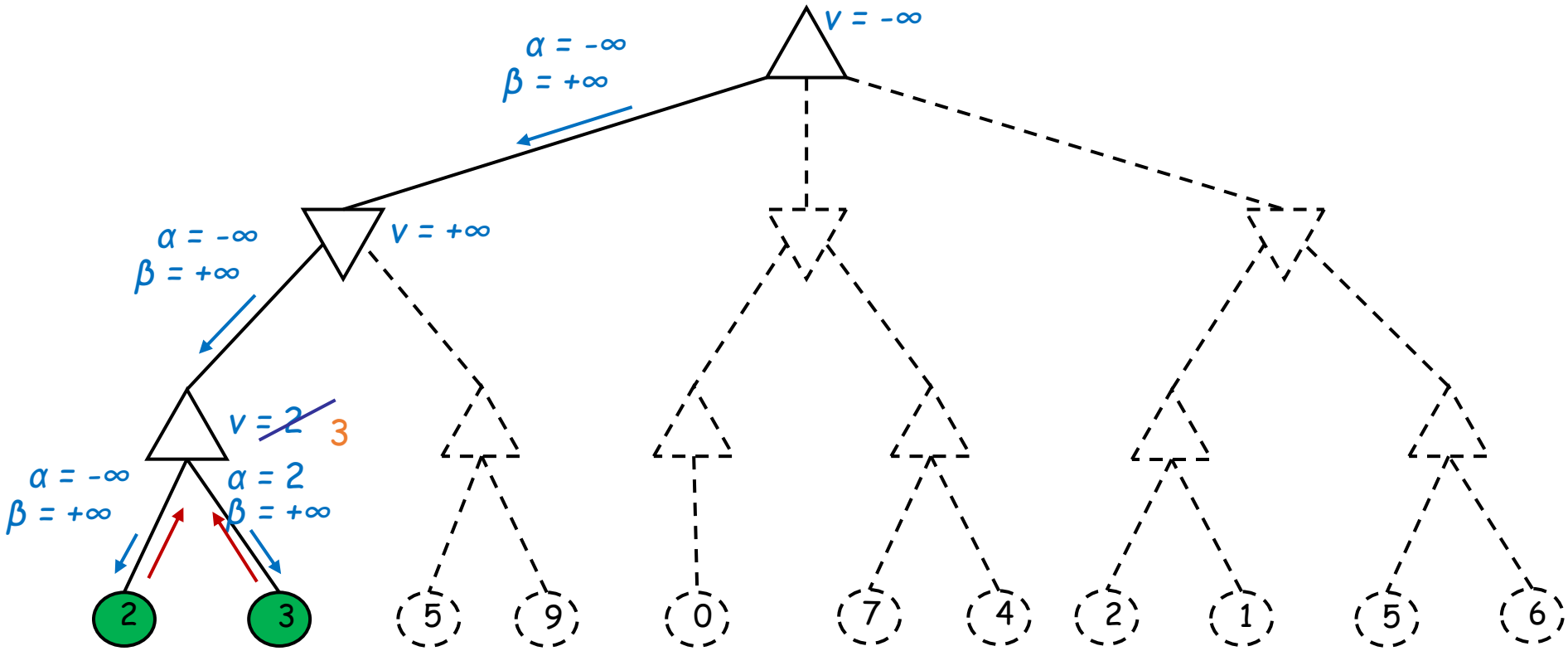
Example



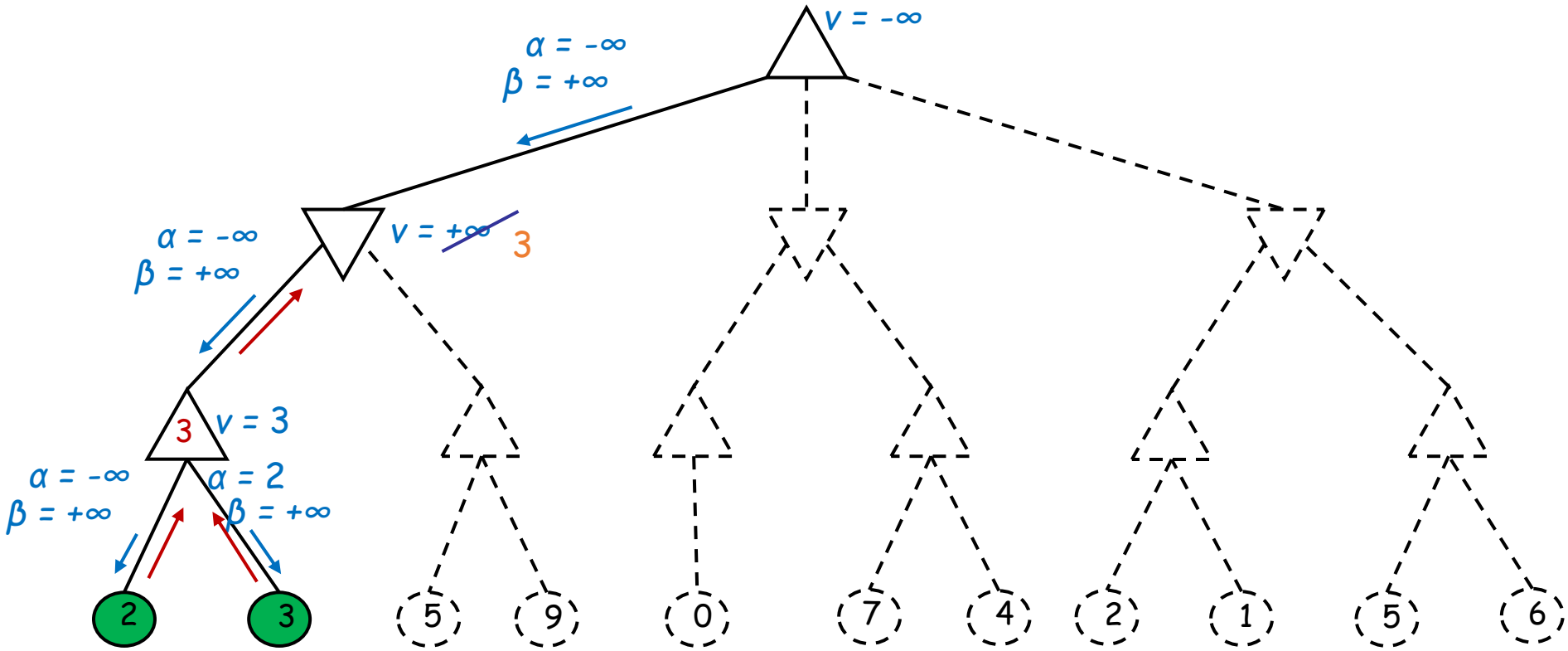
Example



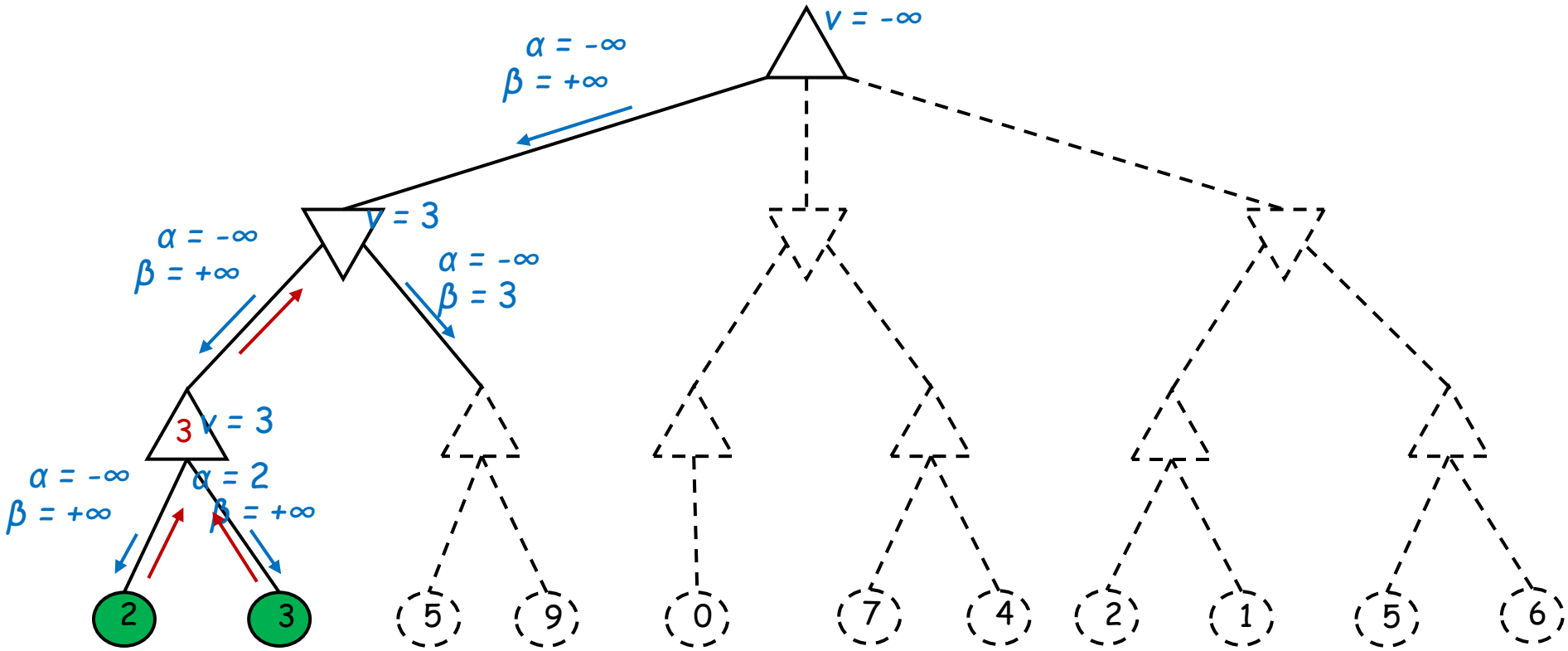
Example



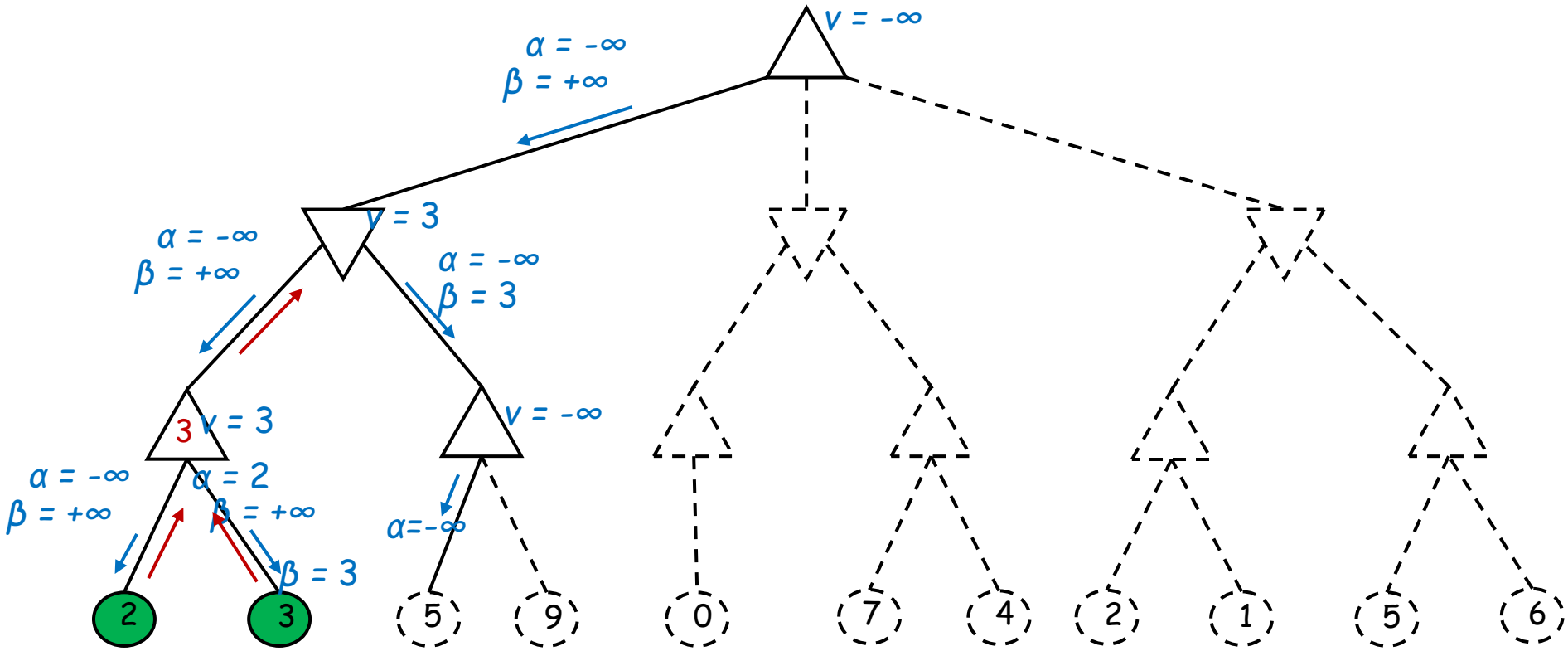
Example



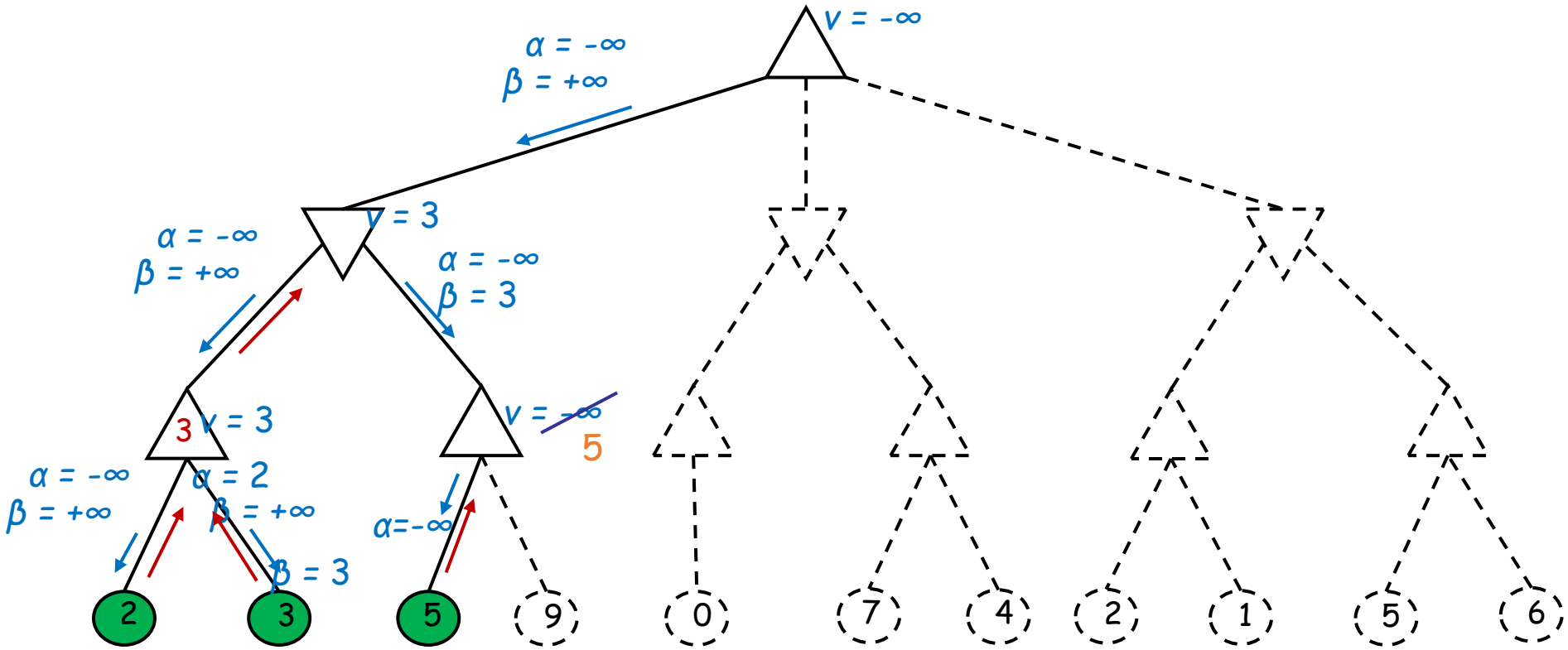
Example



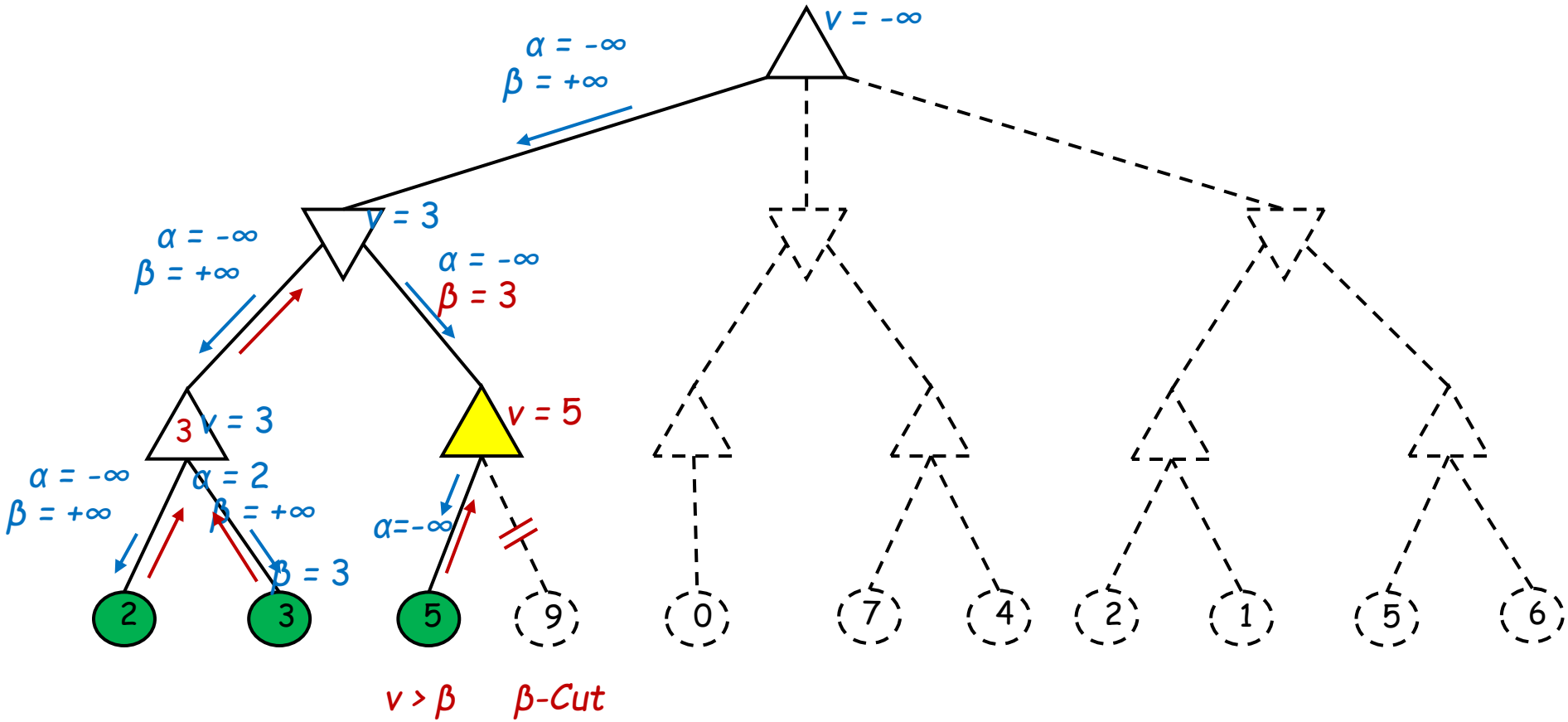
Example



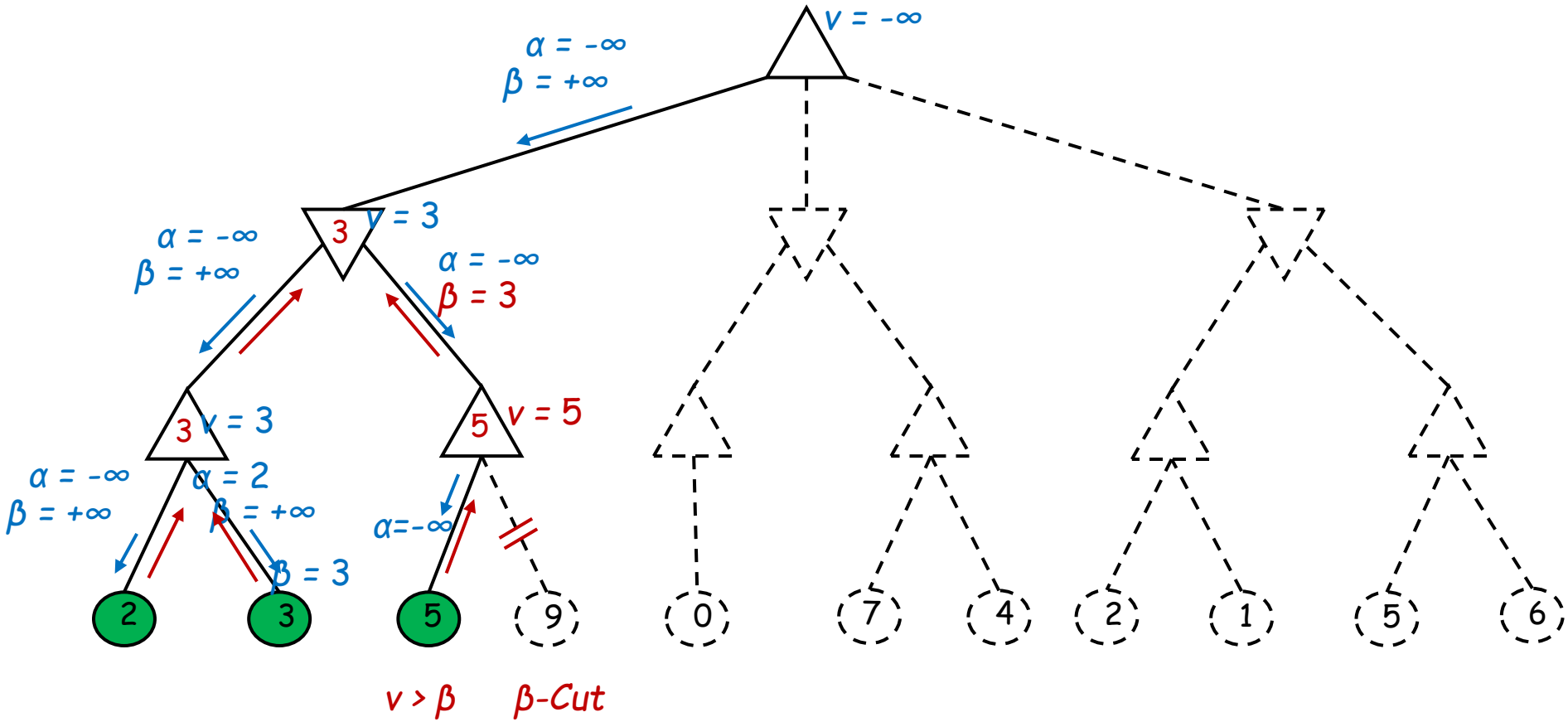
Example



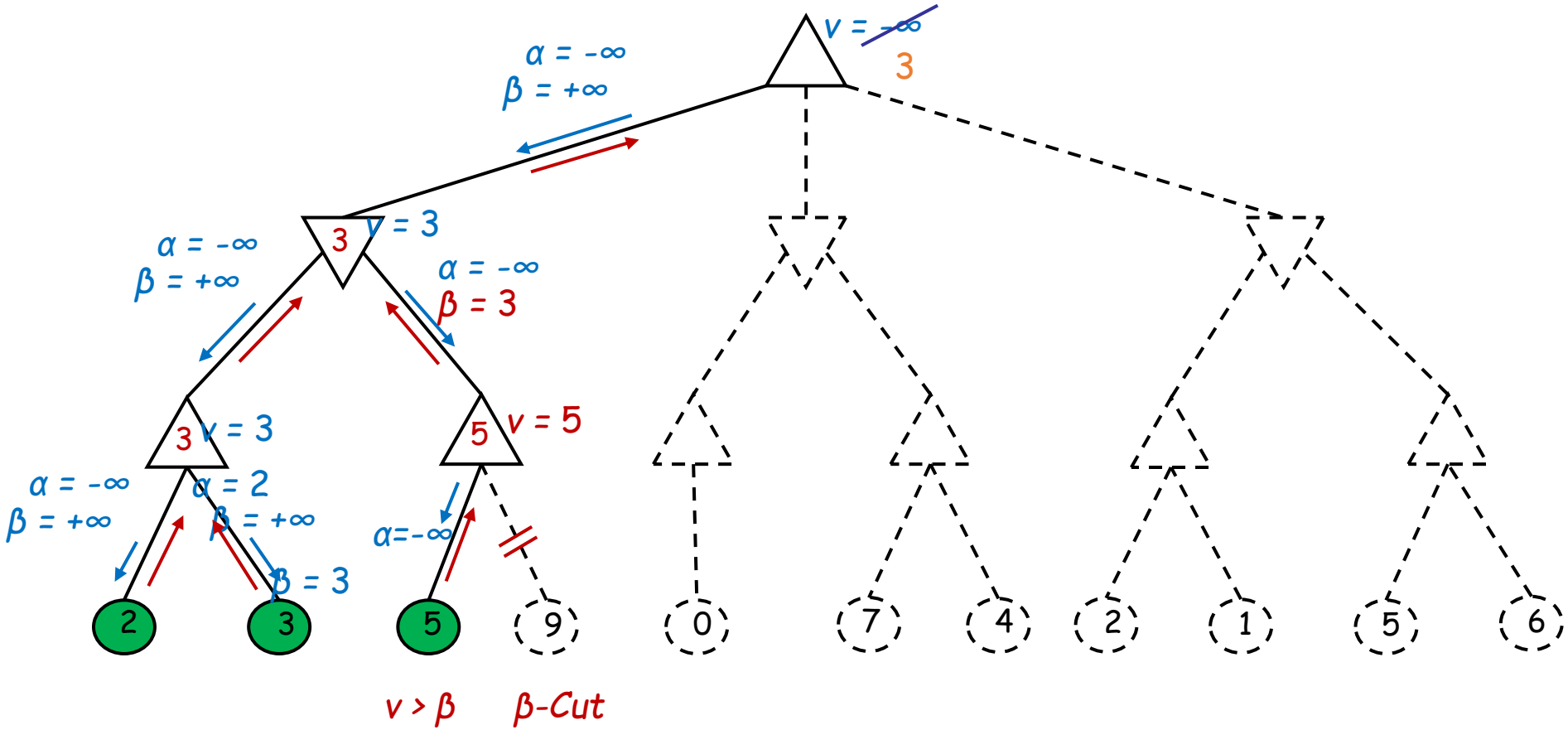
Example



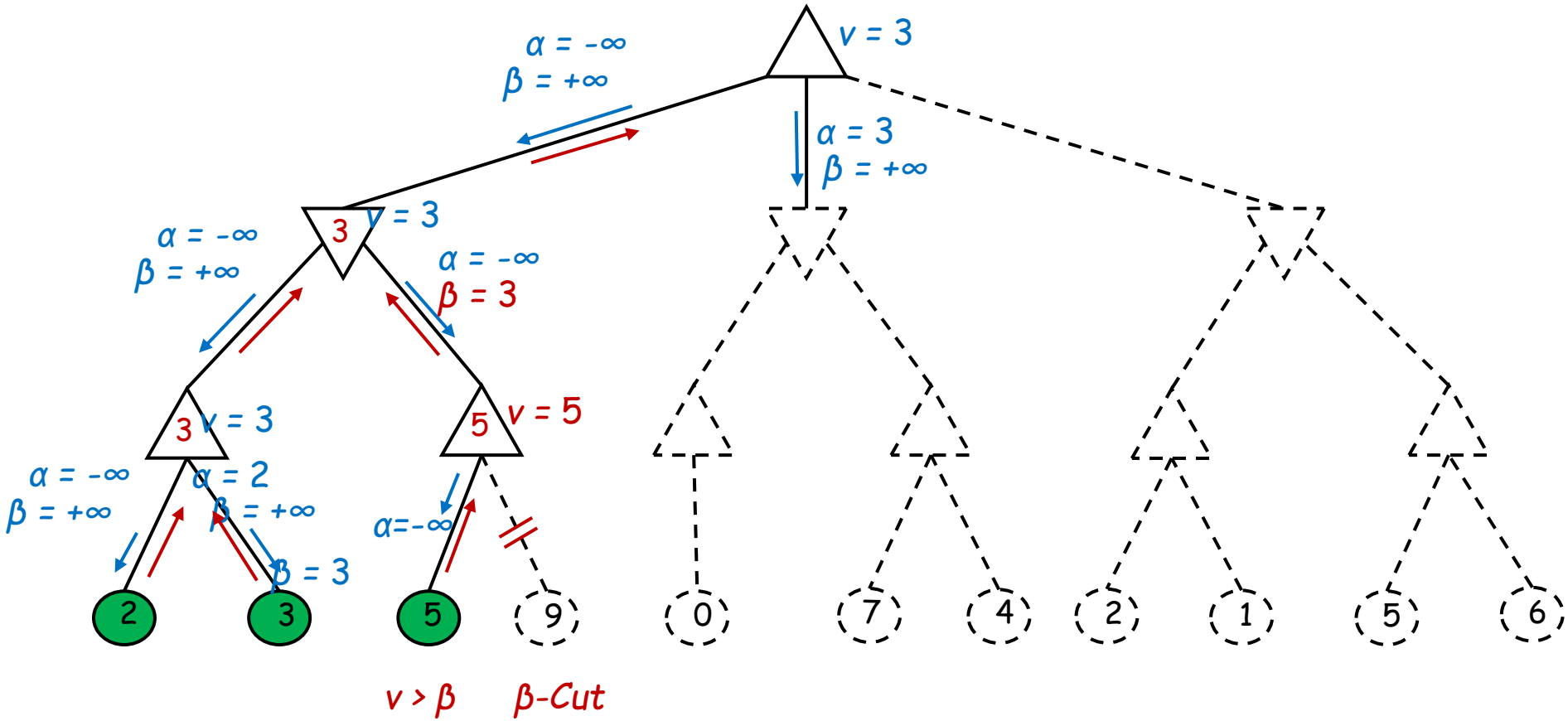
Example



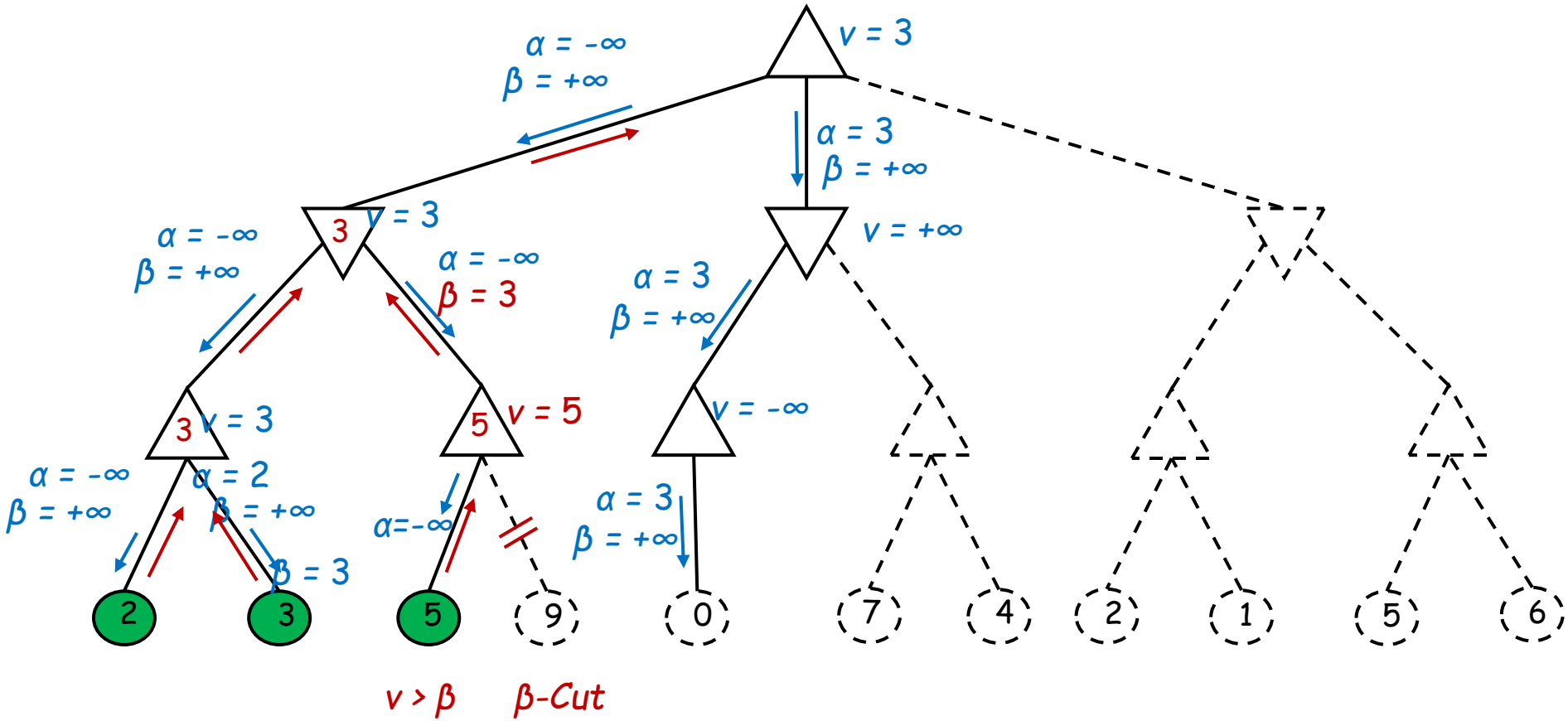
Example



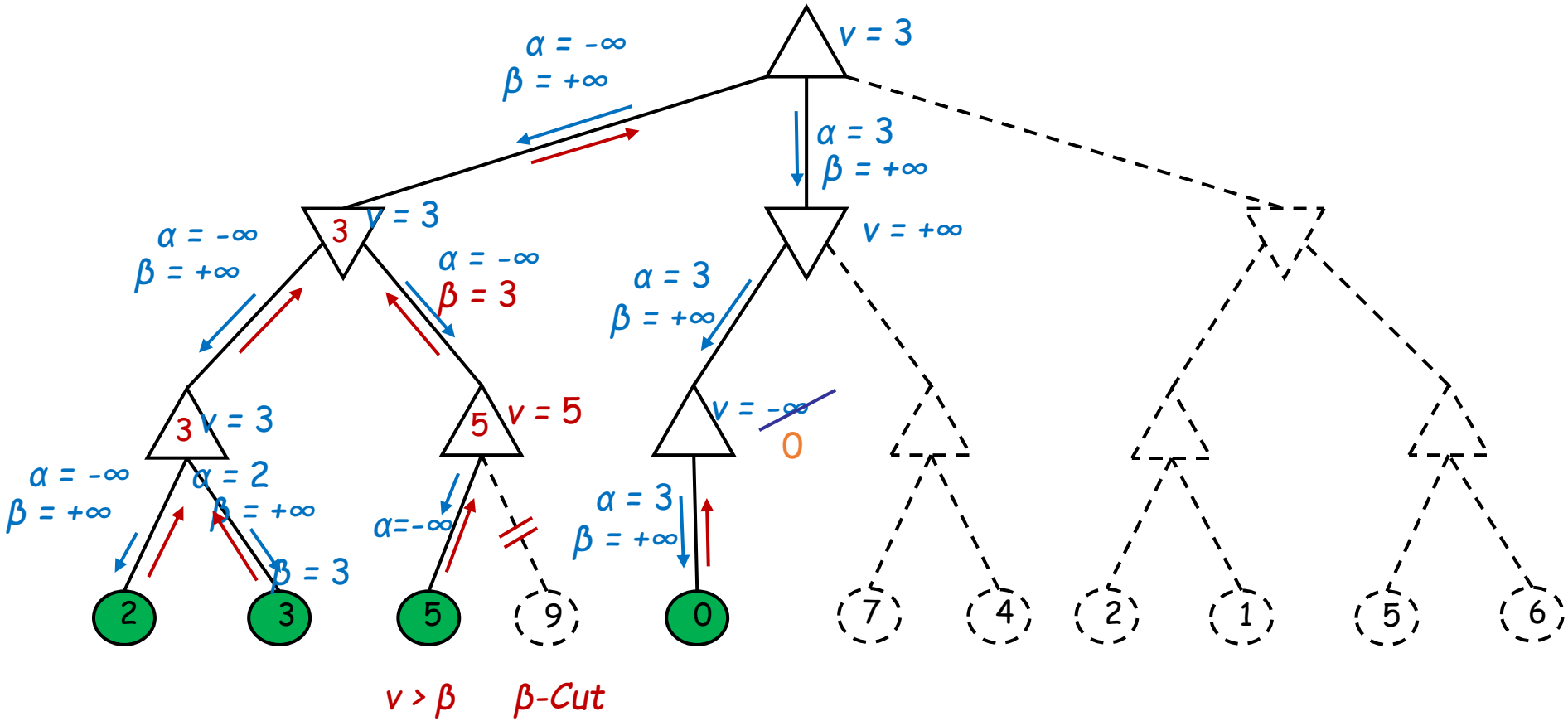
Example



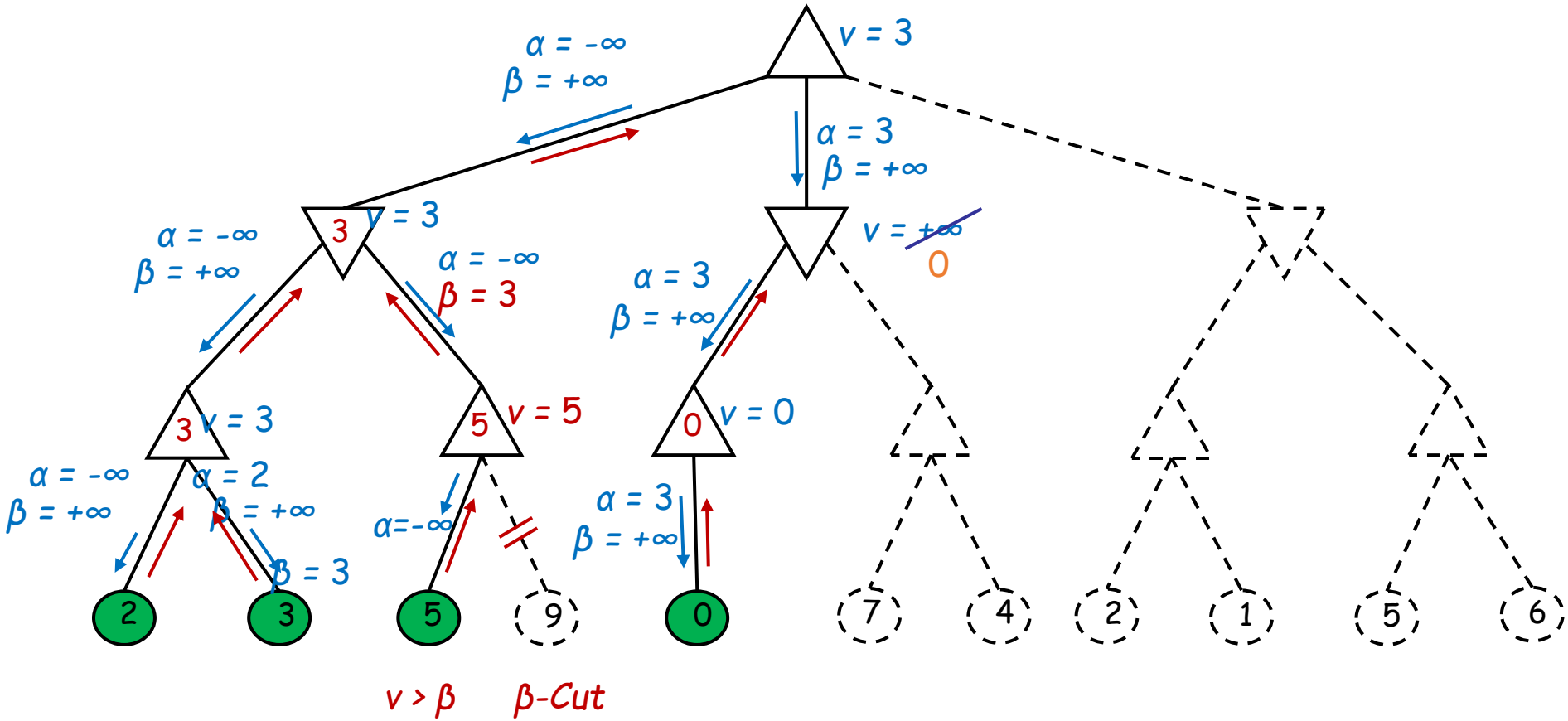
Example


α = -∞
β = +∞
v = 3
α = 3
β = +∞
α = -∞
β = +∞
3
v = 3
α = -∞
β = 3
v = +∞
α = 3
β = +∞
3
v = 3
5
v = 5
v = -∞
α = -∞
β = +∞
α = 2
β = +∞
α=-∞
β = 3
α = 3
β = +∞
2
3
5
9
0
7
4
2
1
5
6
v > β
β-Cut

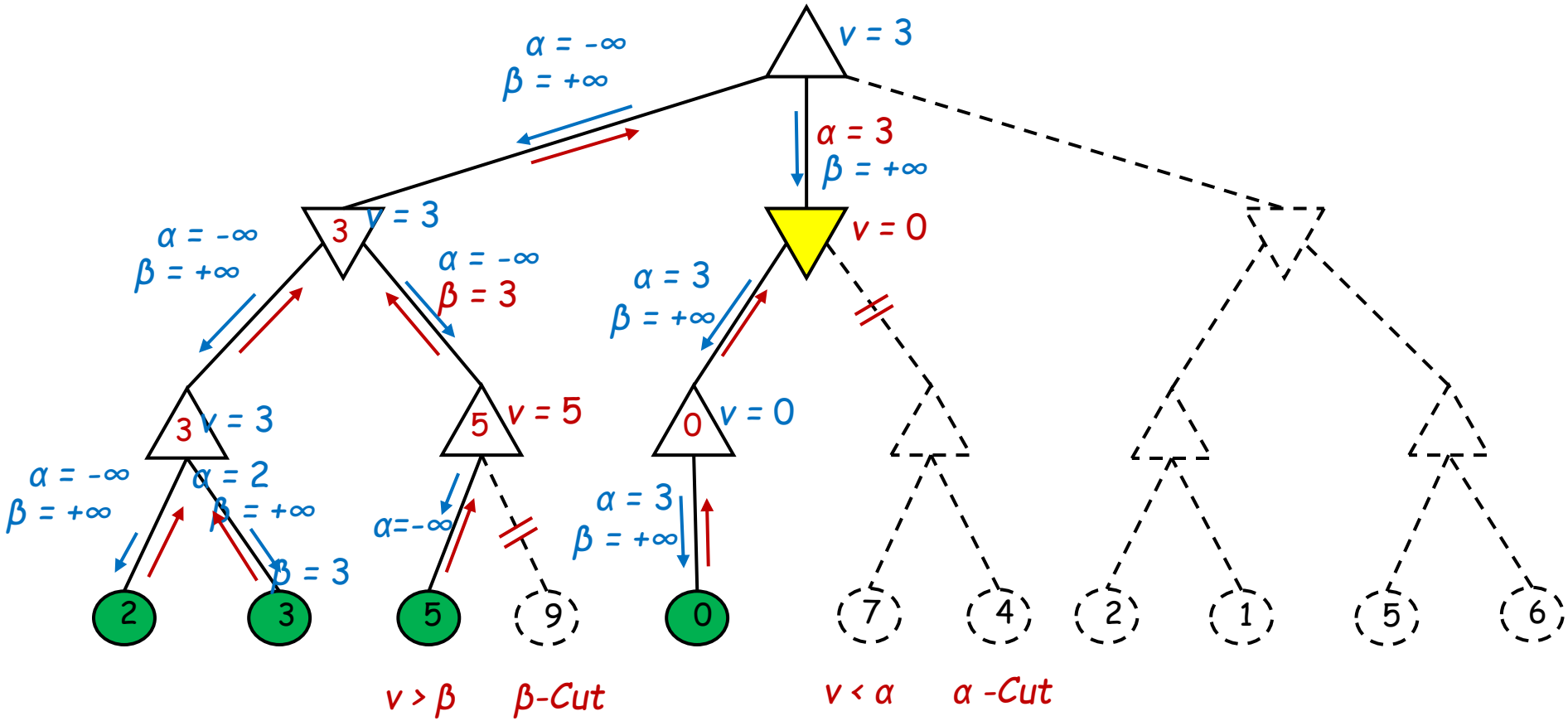
Example


α = -∞
β = +∞
v = 3
α = 3
β = +∞
3
v = 3
α = -∞
β = +∞
α = -∞
β = 3
v = +∞
α = 3
β = +∞
3
v = 3
5
v = 5
v = -∞
0
α = -∞
β = +∞
α = 2
β = +∞
α=-∞
β = 3
α = 3
β = +∞
2
3
5
9
0
7
4
2
1
5
6
v > β
β-Cut

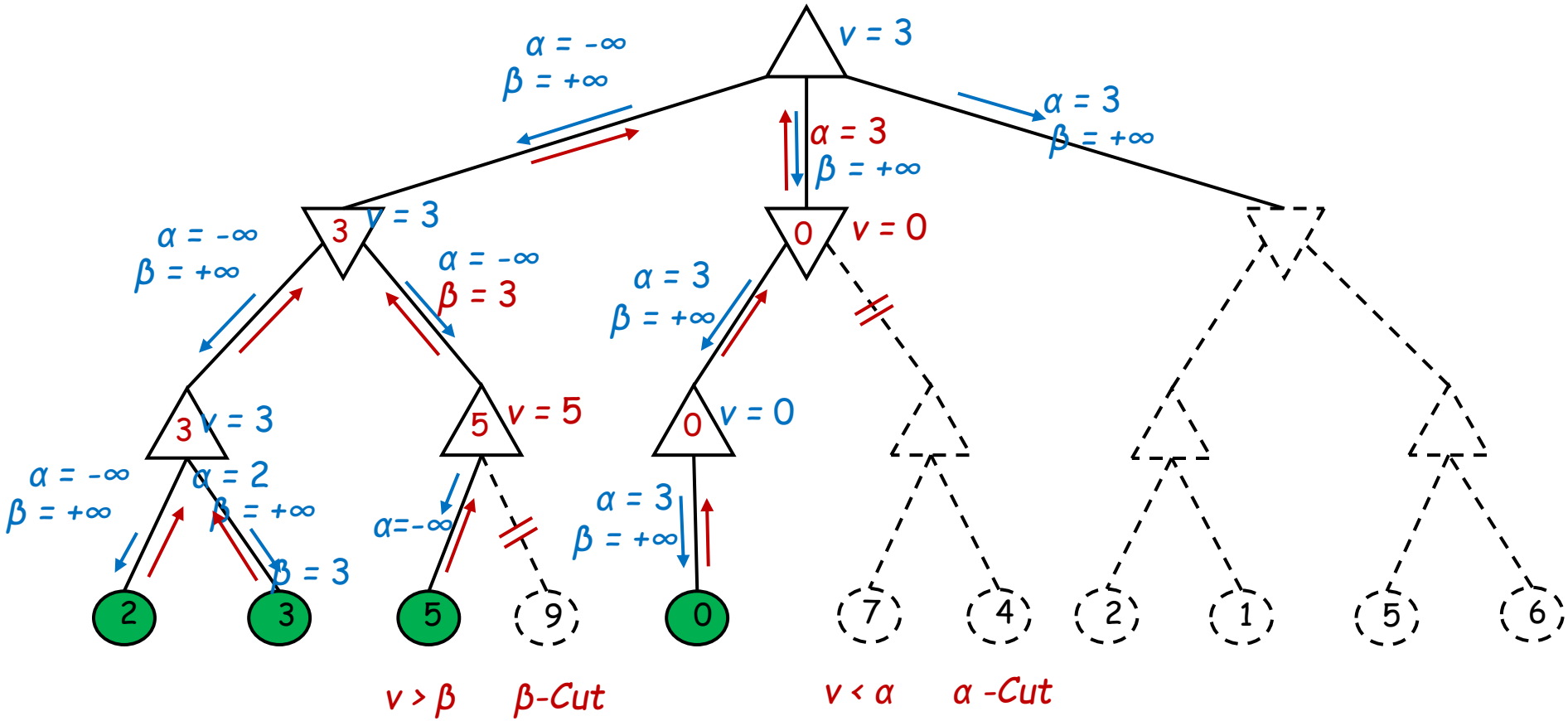
Example


α = -∞
β = +∞
v = 3
α = 3
β = +∞
v = +∞
0
3
v = 3
α = -∞
β = +∞
α = -∞
β = 3
α = 3
β = +∞
3
v = 3
5
v = 5
0
v = 0
α = -∞
β = +∞
α = 2
β = +∞
β = 3
α=-∞
α = 3
β = +∞
2
3
5
9
0
7
4
2
1
5
6
v > β
β-Cut

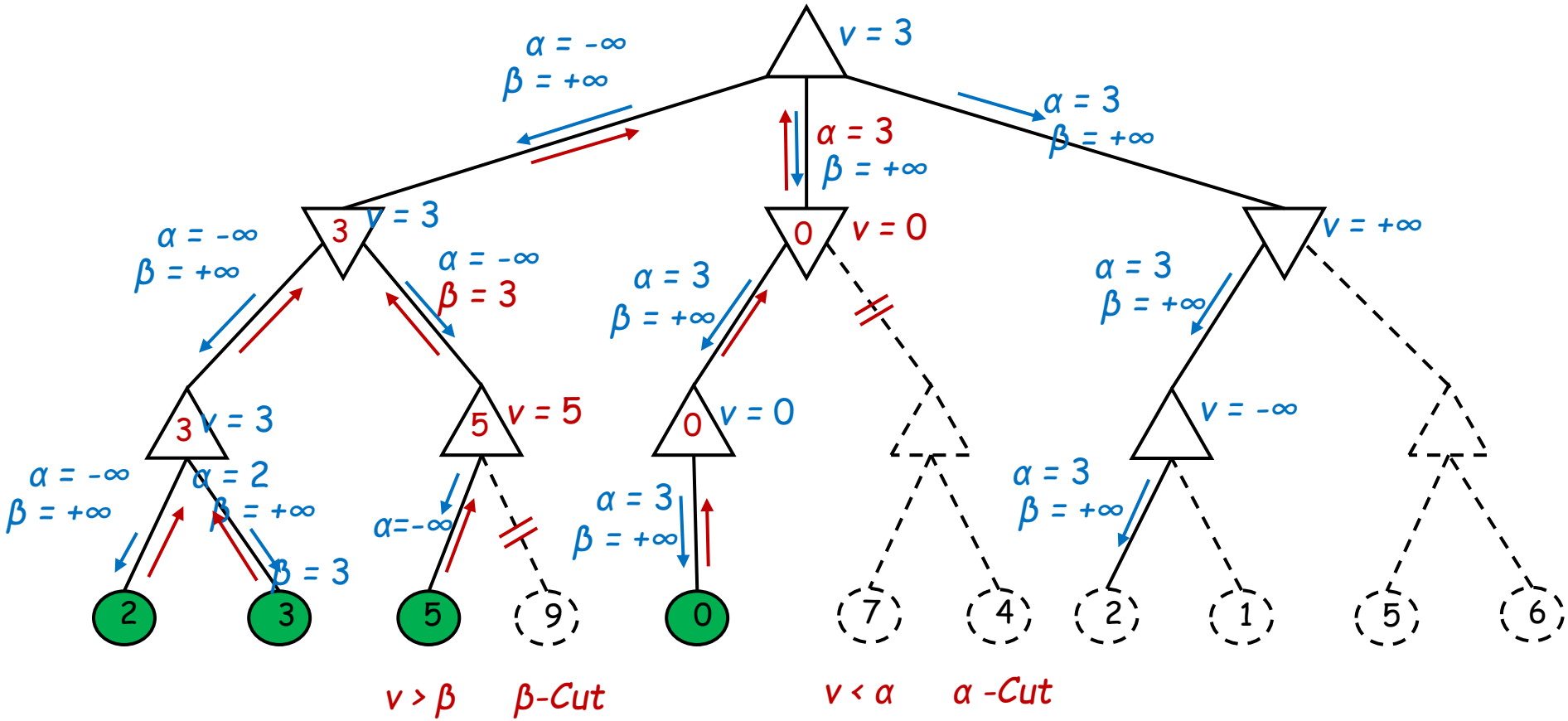
Example


α = -∞
β = +∞
v = 3
α = 3
β = +∞
α = -∞
β = +∞
3
v = 3
α = -∞
β = 3
v = 0
α = 3
β = +∞
3
v = 3
5
v = 5
0
v = 0
α = -∞
α = 2
β = +∞
β = +∞
α=-∞
β = 3
α = 3
β = +∞
2
3
5
9
0
7
4
2
1
5
6
v > β
β-Cut
v < α
α -Cut

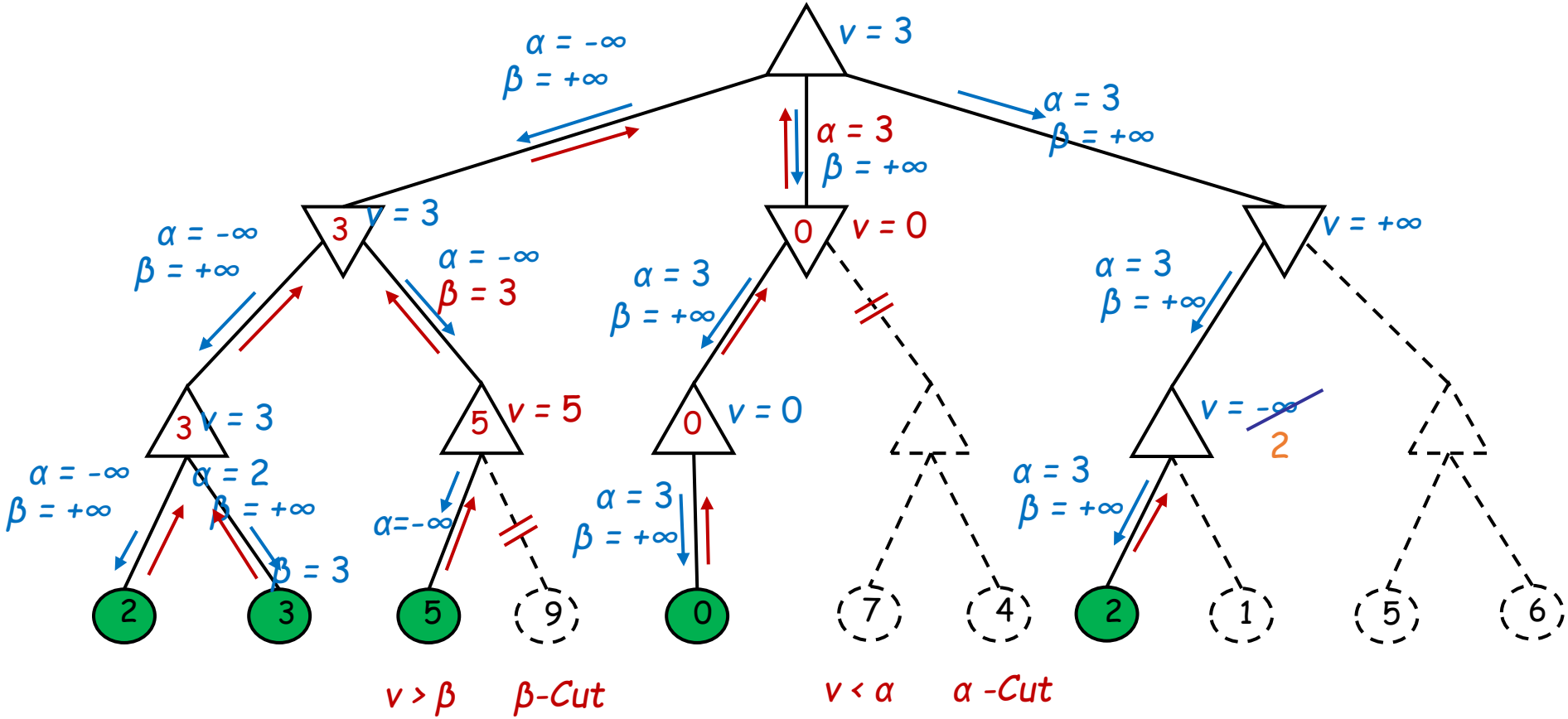
Example


α = -∞
β = +∞
v = 3
α = 3
β = +∞
α = 3
β = +∞
3
v = 3
α = -∞
β = +∞
α = -∞
β = 3
0
v = 0
α = 3
β = +∞
3
v = 3
5
v = 5
0
v = 0
α = -∞
α = 2
β = +∞
β = +∞
α=-∞
β = 3
α = 3
β = +∞
2
3
5
9
0
7
4
2
1
5
6
v > β
β-Cut
v < α
α -Cut

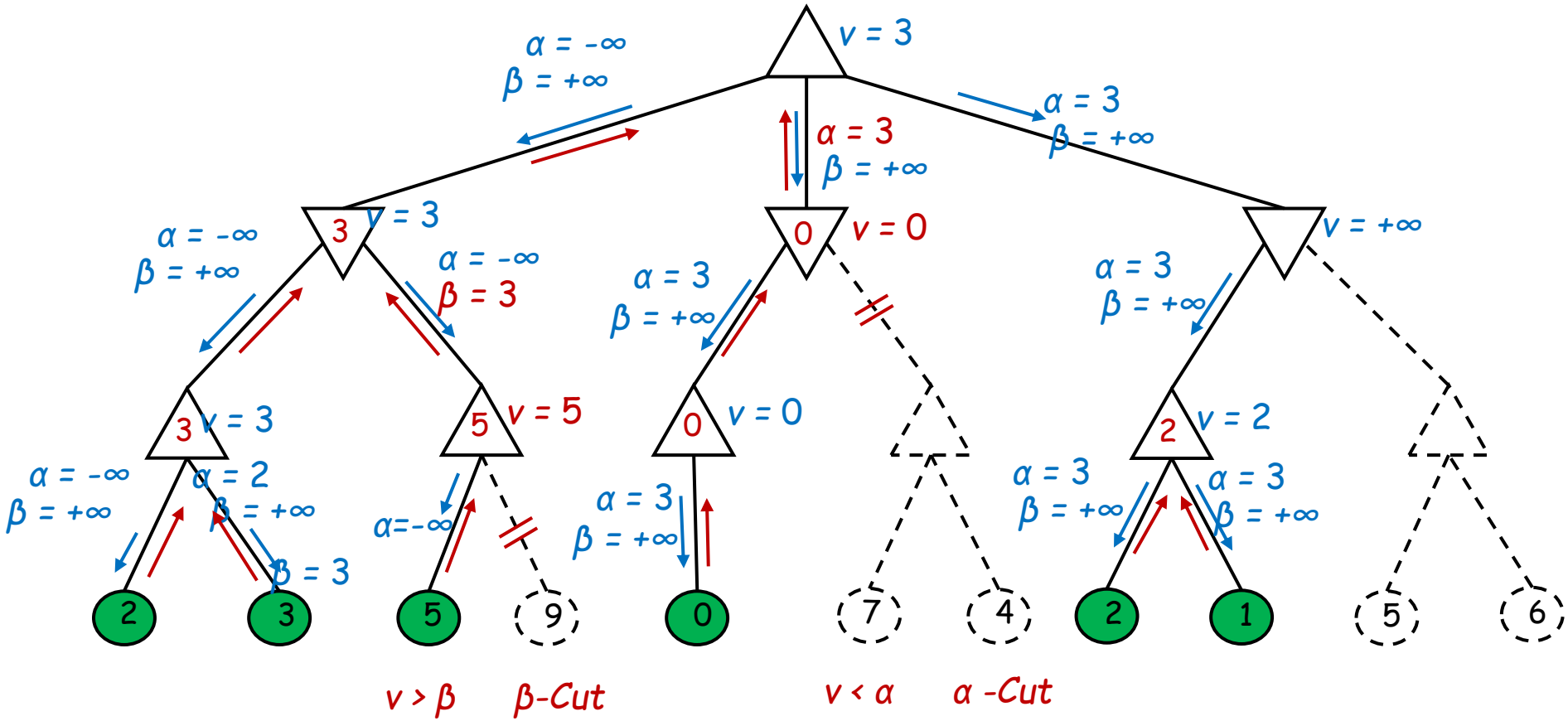
Example


α = -∞
β = +∞
v = 3
α = 3
β = +∞
α = 3
β = +∞
α = -∞
β = +∞
3
v = 3
α = -∞
β = 3
0
v = 0
v = +∞
α = 3
β = +∞
α = 3
β = +∞
3
v = 3
5
v = 5
0
v = 0
v = -∞
α = -∞
β = +∞
α = 2
β = +∞
α=-∞
β = 3
α = 3
β = +∞
α = 3
β = +∞
2
3
5
9
0
7
4
2
1
5
6
v > β
β-Cut
v < α
α -Cut

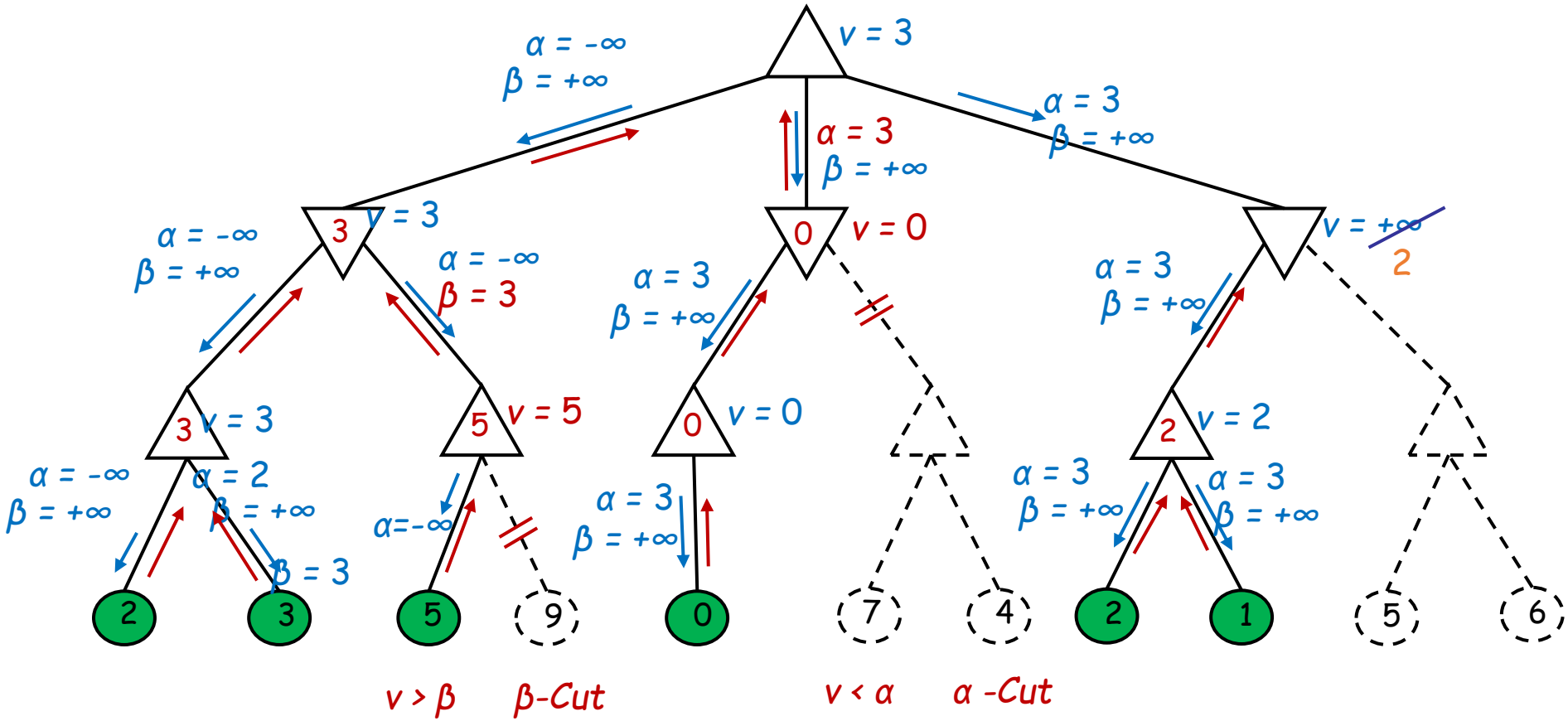
Example


α = -∞
β = +∞
v = 3
α = 3
β = +∞
α = 3
β = +∞
α = -∞
β = +∞
3
v = 3
α = -∞
β = 3
0
v = 0
v = +∞
α = 3
β = +∞
α = 3
β = +∞
3
v = 3
5
v = 5
0
v = 0
v = -∞
2
α = -∞
β = +∞
α = 2
β = +∞
β = 3
α=-∞
α = 3
β = +∞
α = 3
β = +∞
2
3
5
9
0
7
4
2
1
5
6
v > β
β-Cut
v < α
α -Cut

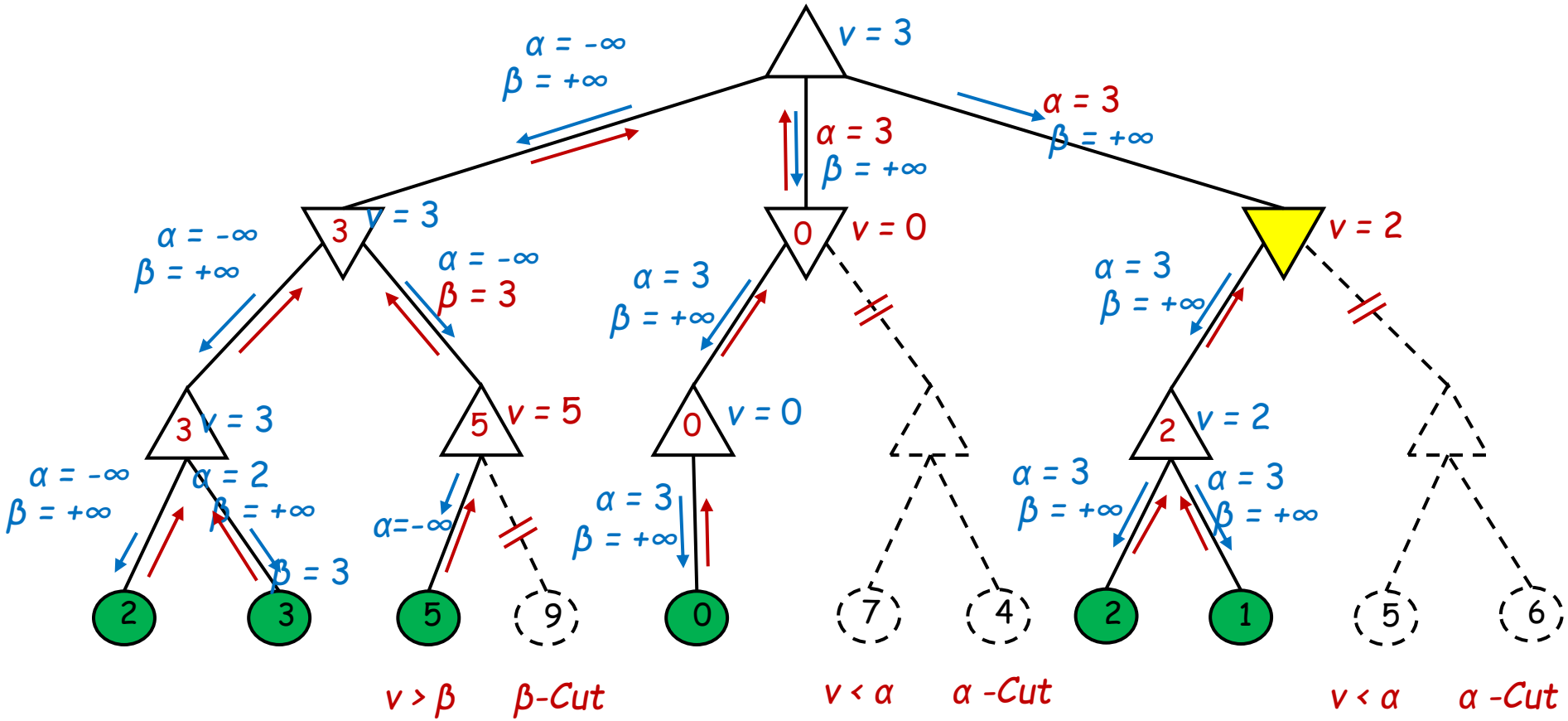
Example


α = -∞
β = +∞
v = 3
α = 3
β = +∞
α = 3
β = +∞
3
v = 3
α = -∞
β = +∞
α = -∞
β = 3
0
v = 0
α = 3
β = +∞
v = +∞
α = 3
β = +∞
3
v = 3
5
v = 5
0
v = 0
2
v = 2
α = -∞
β = +∞
α = 2
β = +∞
β = 3
α=-∞
α = 3
β = +∞
α = 3
β = +∞
α = 3
β = +∞
2
3
5
9
0
7
4
2
1
5
6
v > β
β-Cut
v < α
α -Cut

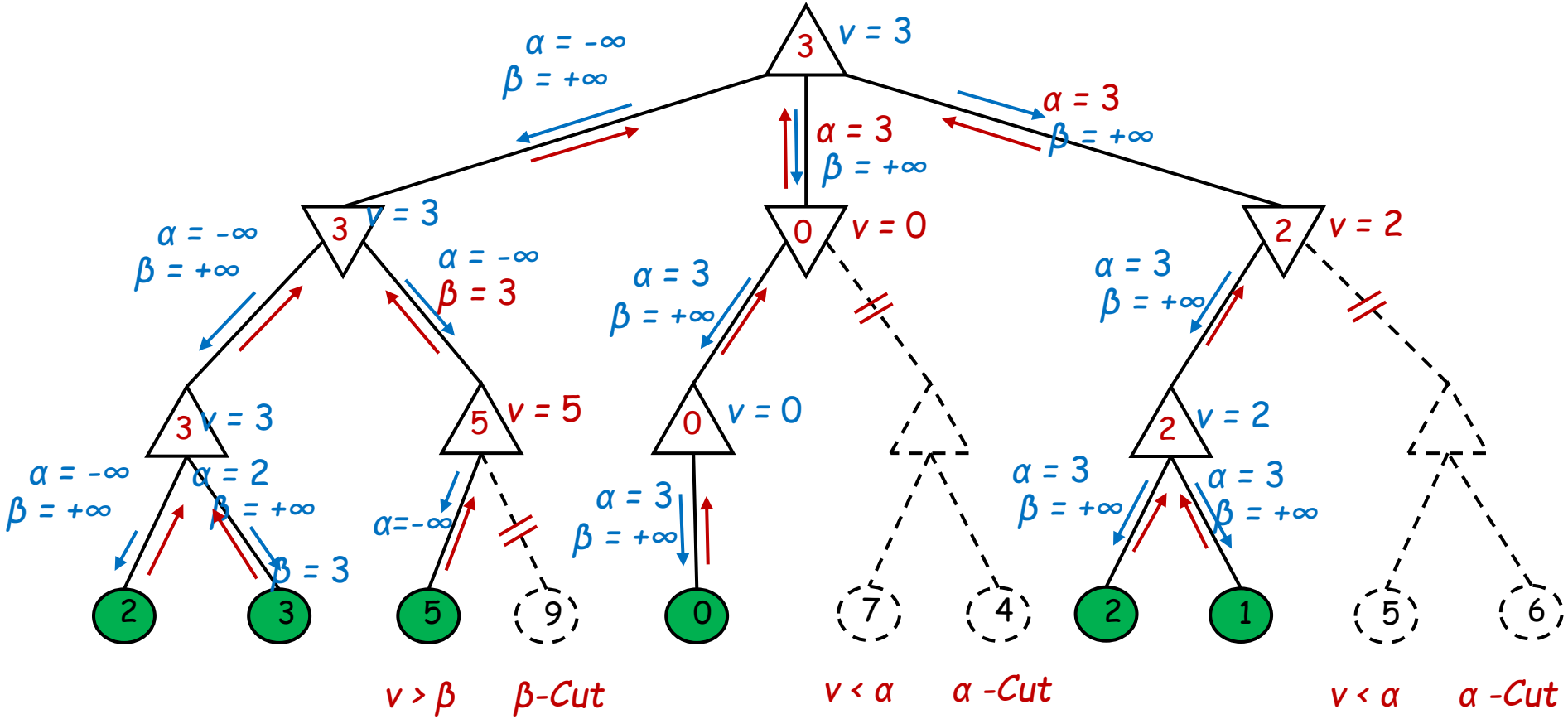
Example


α = -∞
β = +∞
v = 3
α = 3
β = +∞
α = 3
β = +∞
α = -∞
β = +∞
3
v = 3
α = -∞
β = 3
0
v = 0
v = +∞
2
α = 3
β = +∞
α = 3
β = +∞
3
v = 3
5
v = 5
0
v = 0
2
v = 2
α = -∞
β = +∞
α = 2
β = +∞
α=-∞
α = 3
β = +∞
α = 3
β = +∞
α = 3
β = +∞
β = 3
2
3
5
9
0
7
4
2
1
5
6
v > β
β-Cut
v < α
α -Cut

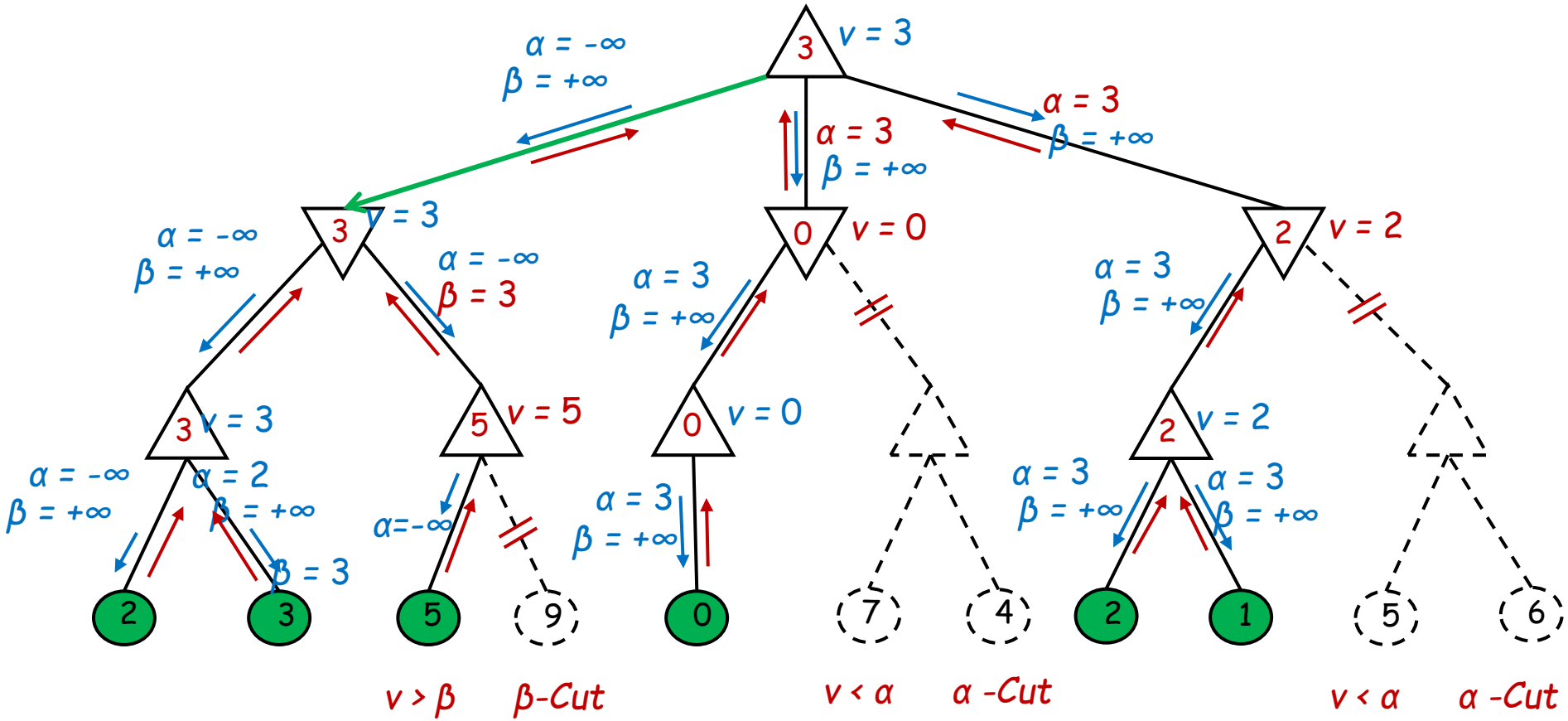
Example


α = -∞
β = +∞
v = 3
α = 3
β = +∞
α = 3
β = +∞
α = -∞
β = +∞
3
v = 3
α = -∞
β = 3
0
v = 0
α = 3
β = +∞
v = 2
α = 3
β = +∞
3
v = 3
5
v = 5
0
v = 0
2
v = 2
α = -∞
β = +∞
α = 2
β = +∞
β = 3
α=-∞
α = 3
β = +∞
α = 3
β = +∞
α = 3
β = +∞
2
3
5
9
0
7
4
2
1
5
6
v > β
β-Cut
v < α
α -Cut
v < α
α -Cut

Example

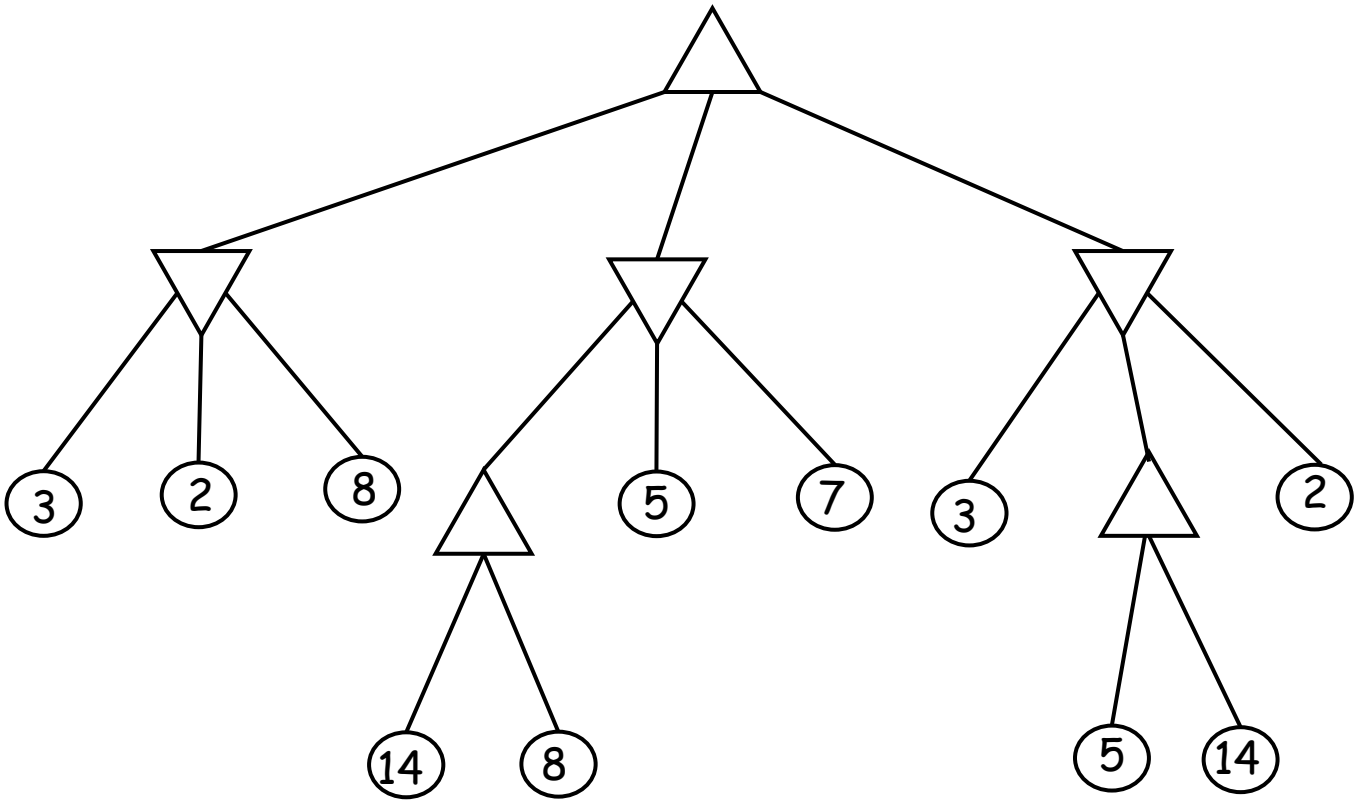

α = -∞
β = +∞
3
v = 3
α = 3
β = +∞
α = 3
β = +∞
α = -∞
β = +∞
3
v = 3
α = -∞
β = 3
0
v = 0
2
v = 2
α = 3
β = +∞
α = 3
β = +∞
3
v = 3
5
v = 5
0
v = 0
2
v = 2
α = -∞
β = +∞
α = 2
β = +∞
α=-∞
α = 3
β = +∞
α = 3
β = +∞
α = 3
β = +∞
β = 3
2
3
5
9
0
7
4
2
1
5
6
v > β
β-Cut
v < α
α -Cut
v < α
α -Cut

Example


α = -∞
β = +∞
3
v = 3
α = 3
β = +∞
α = 3
β = +∞
α = -∞
β = +∞
3
v = 3
α = -∞
β = 3
0
v = 0
2
v = 2
α = 3
β = +∞
α = 3
β = +∞
3
v = 3
5
v = 5
0
v = 0
2
v = 2
α = -∞
β = +∞
α = 2
β = +∞
α=-∞
α = 3
β = +∞
α = 3
β = +∞
α = 3
β = +∞
β = 3
2
3
5
9
0
7
4
2
1
5
6
v > β
β-Cut
v < α
α -Cut
v < α
α -Cut

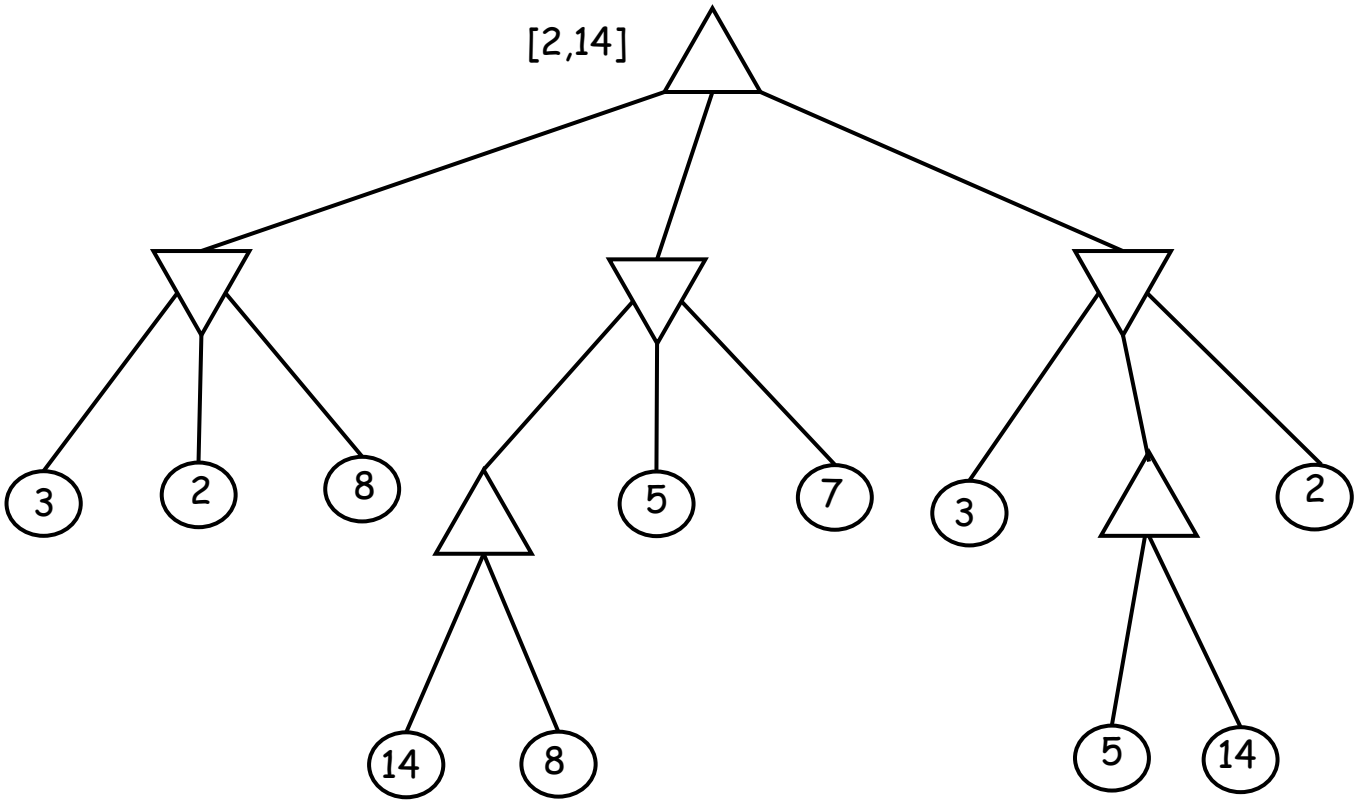
Example

- Effect of knowing the lower bound and upper bound.
  - [2, 14] for this example



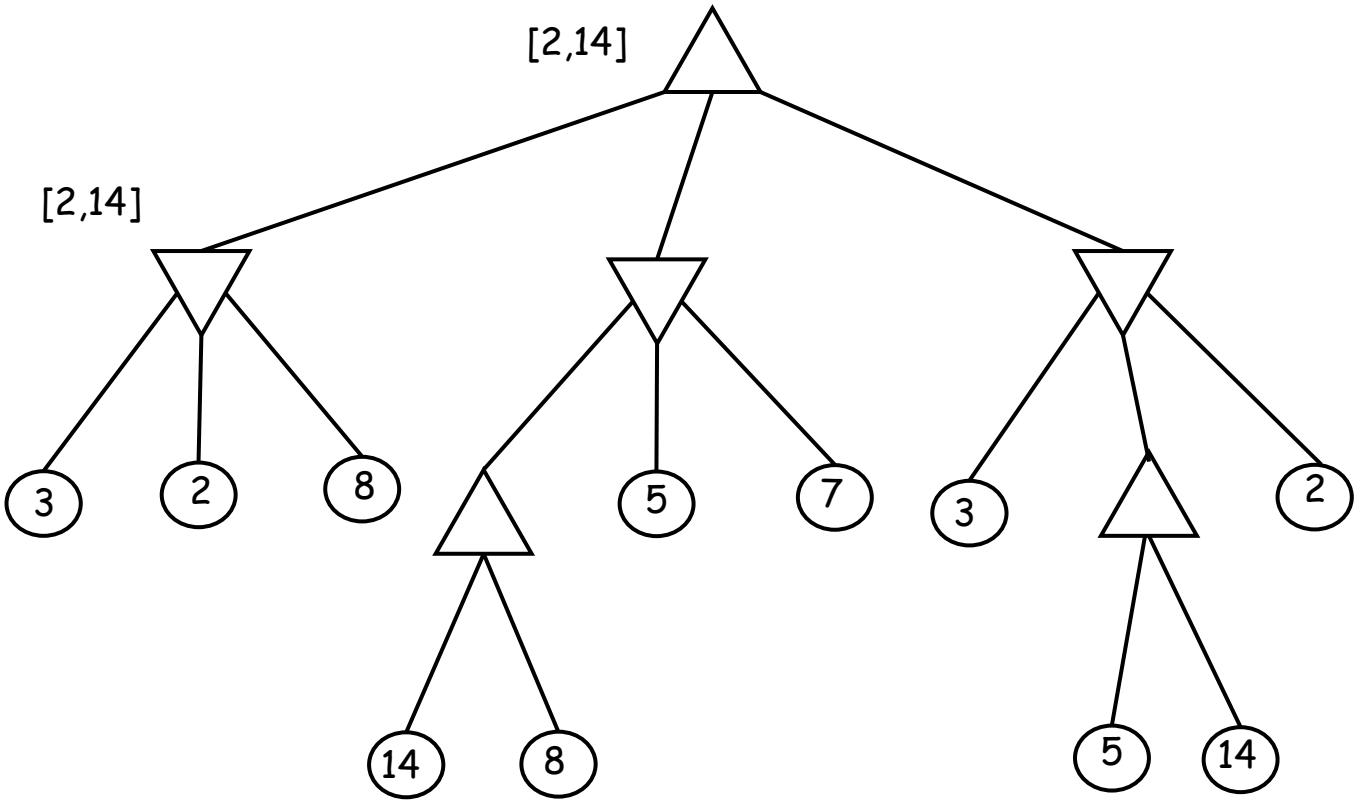
Example

- Effect of knowing the lower bound and upper bound.
  - [2, 14] for this example



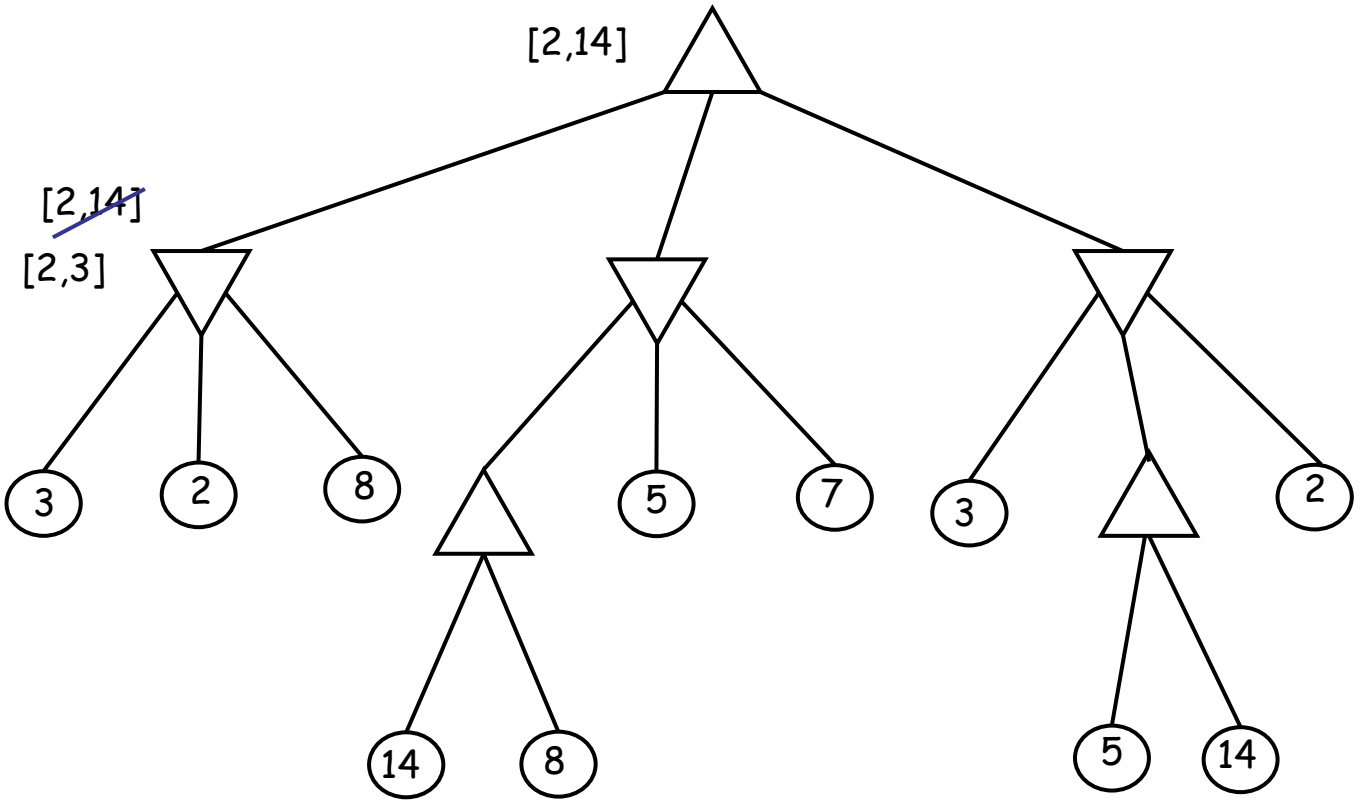
Example

- Effect of knowing the lower bound and upper bound.
  - [2, 14] for this example



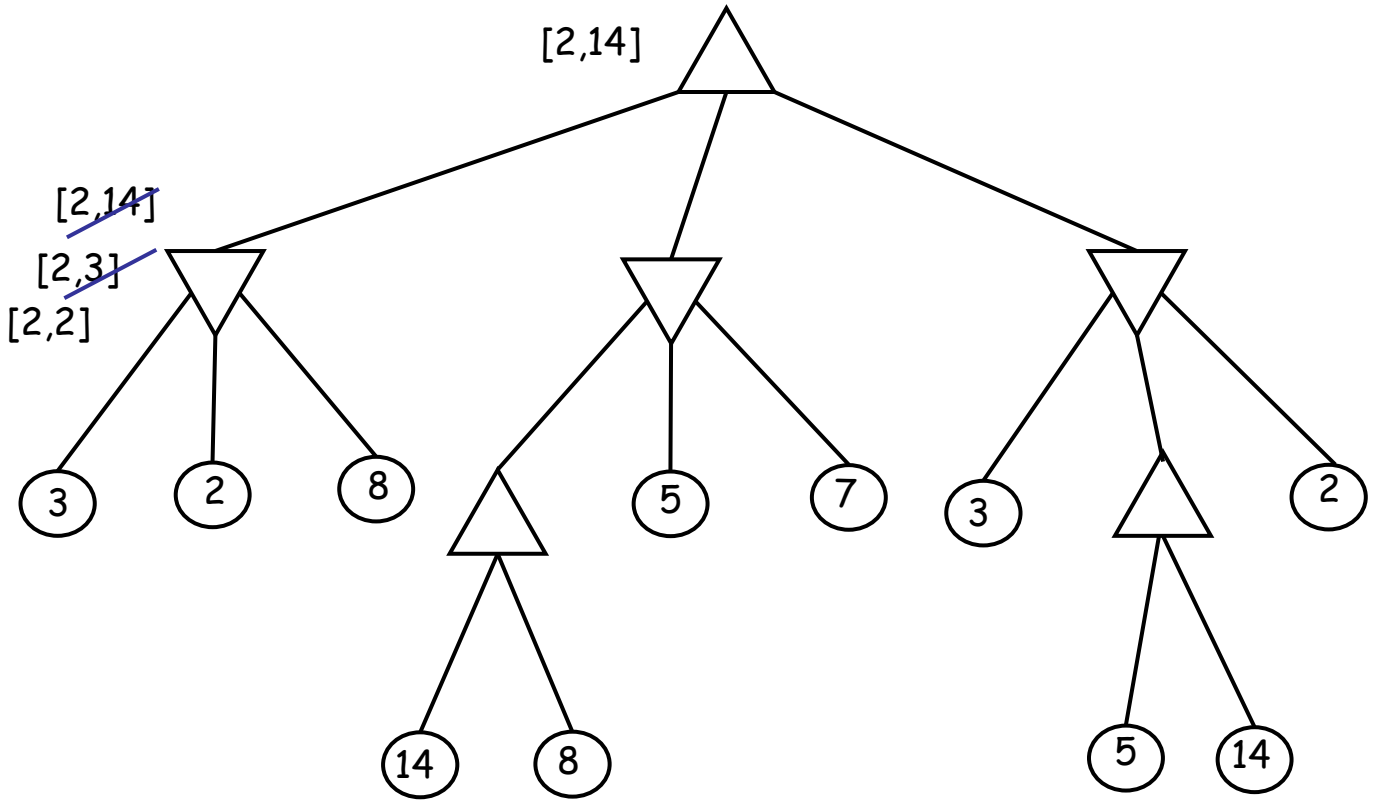
Example

- Effect of knowing the lower bound and upper bound.
  - [2, 14] for this example



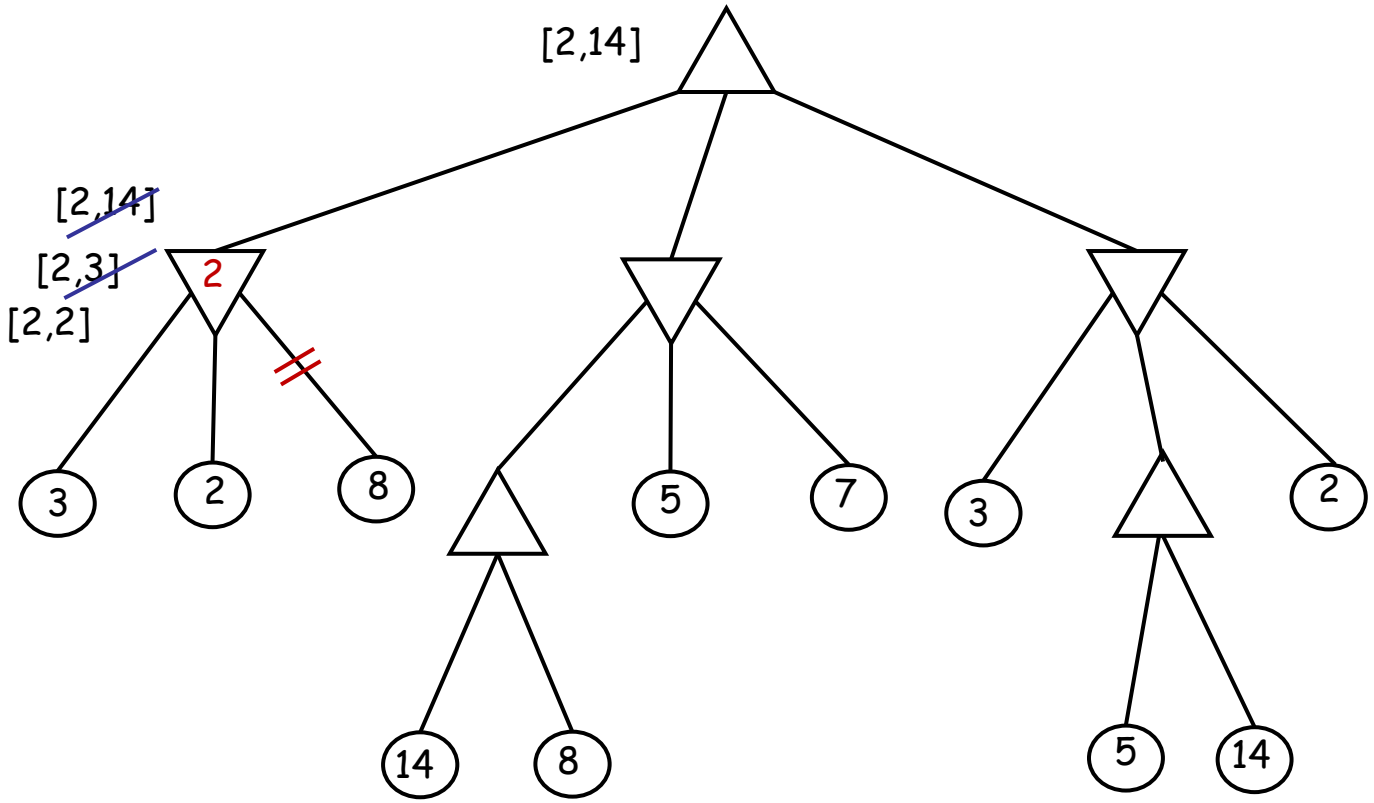
Example

- Effect of knowing the lower bound and upper bound.
  - [2, 14] for this example



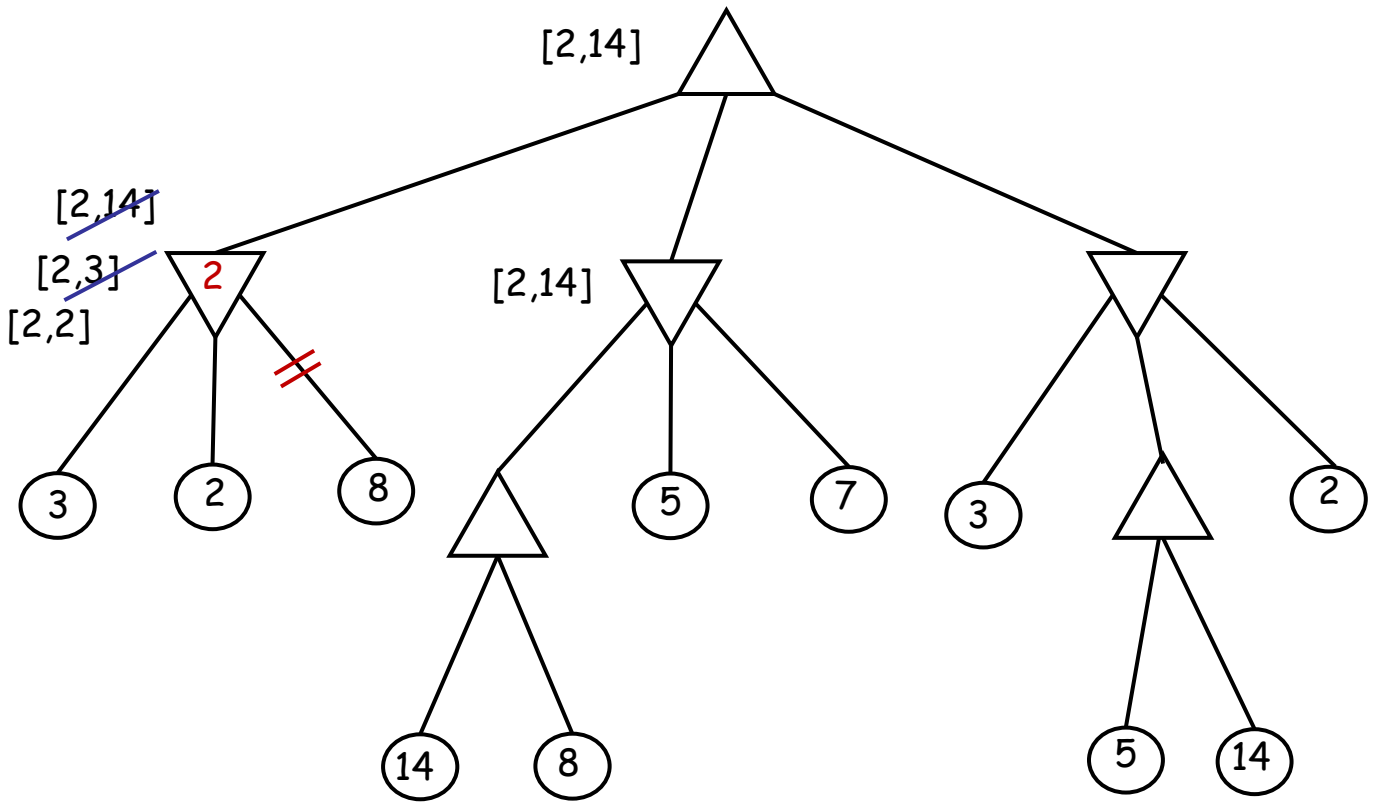
Example

- Effect of knowing the lower bound and upper bound.
  - [2, 14] for this example



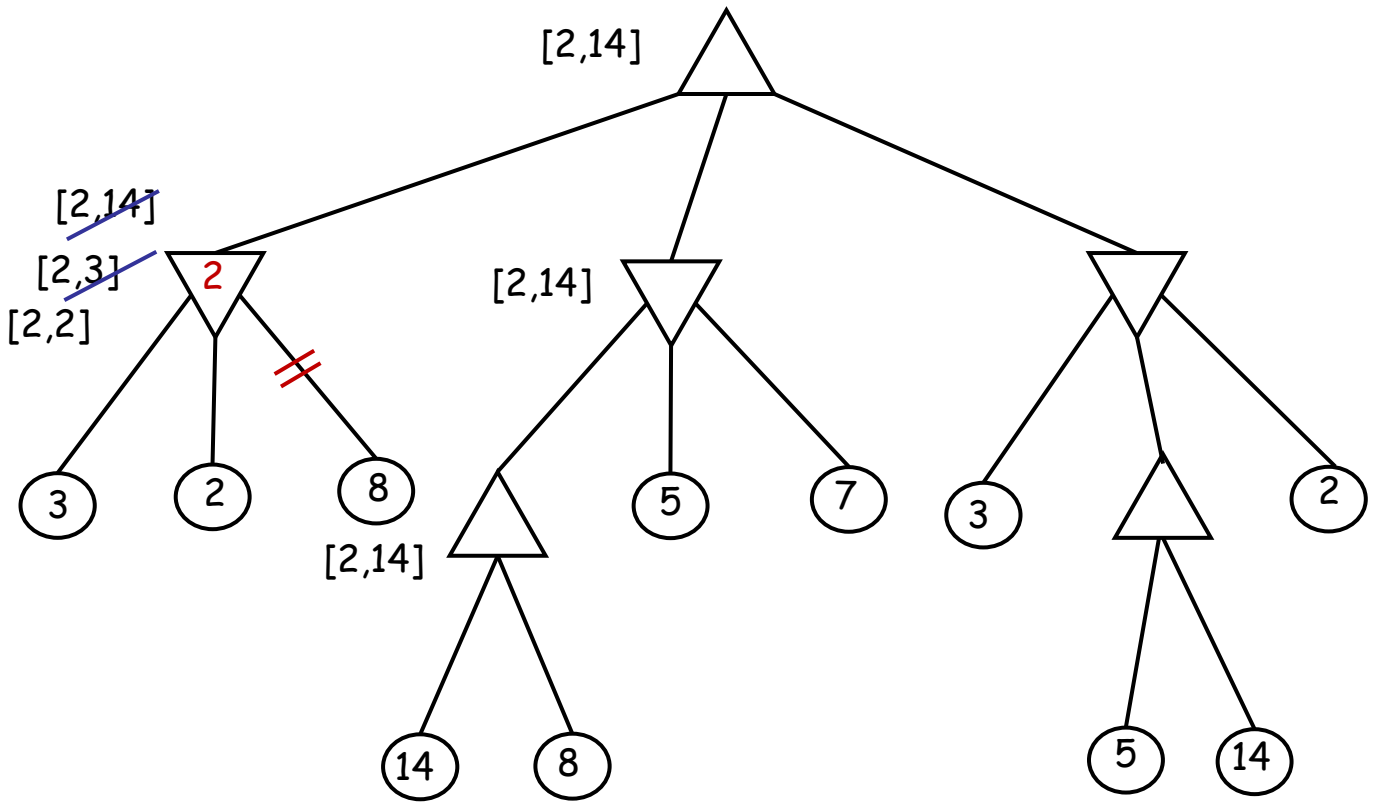
Example

- Effect of knowing the lower bound and upper bound.
  - [2, 14] for this example



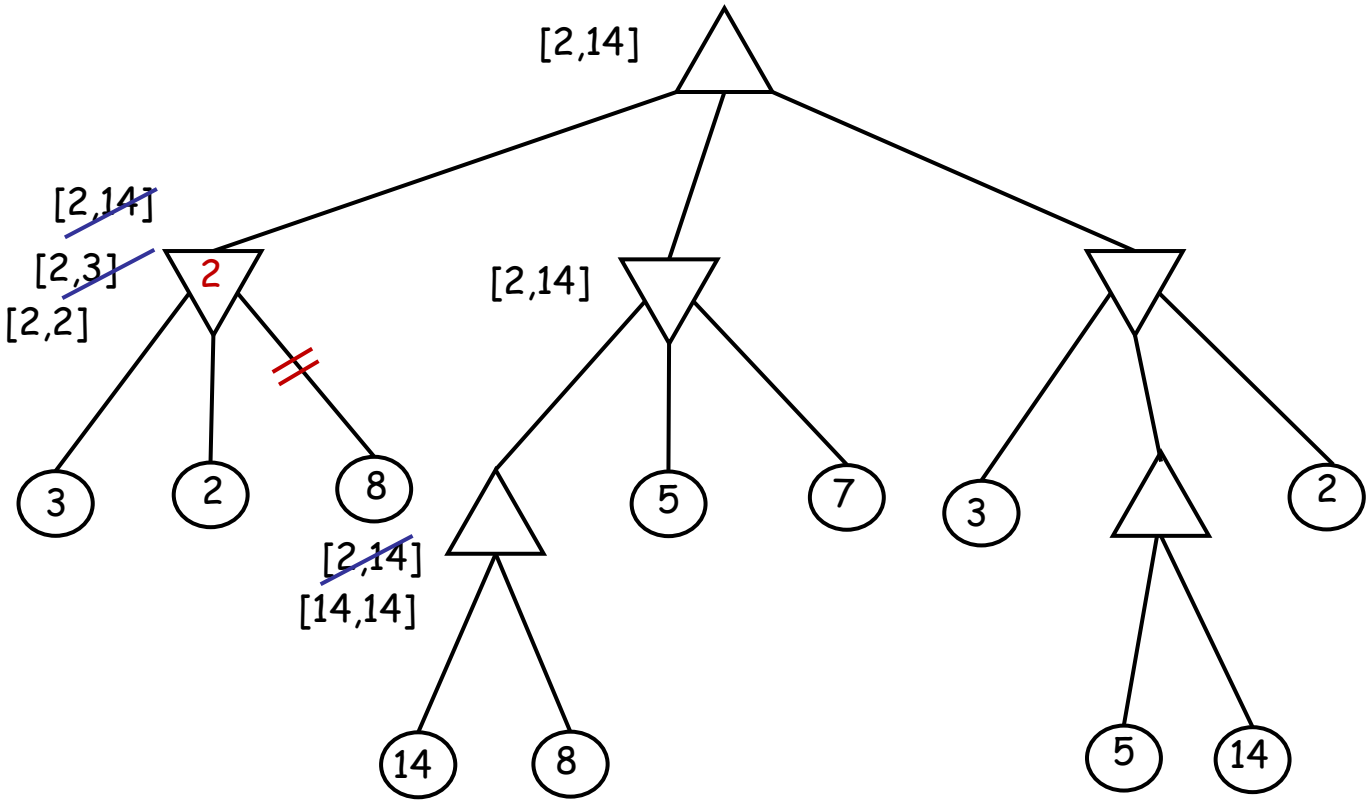
Example

- Effect of knowing the lower bound and upper bound.
  - [2, 14] for this example



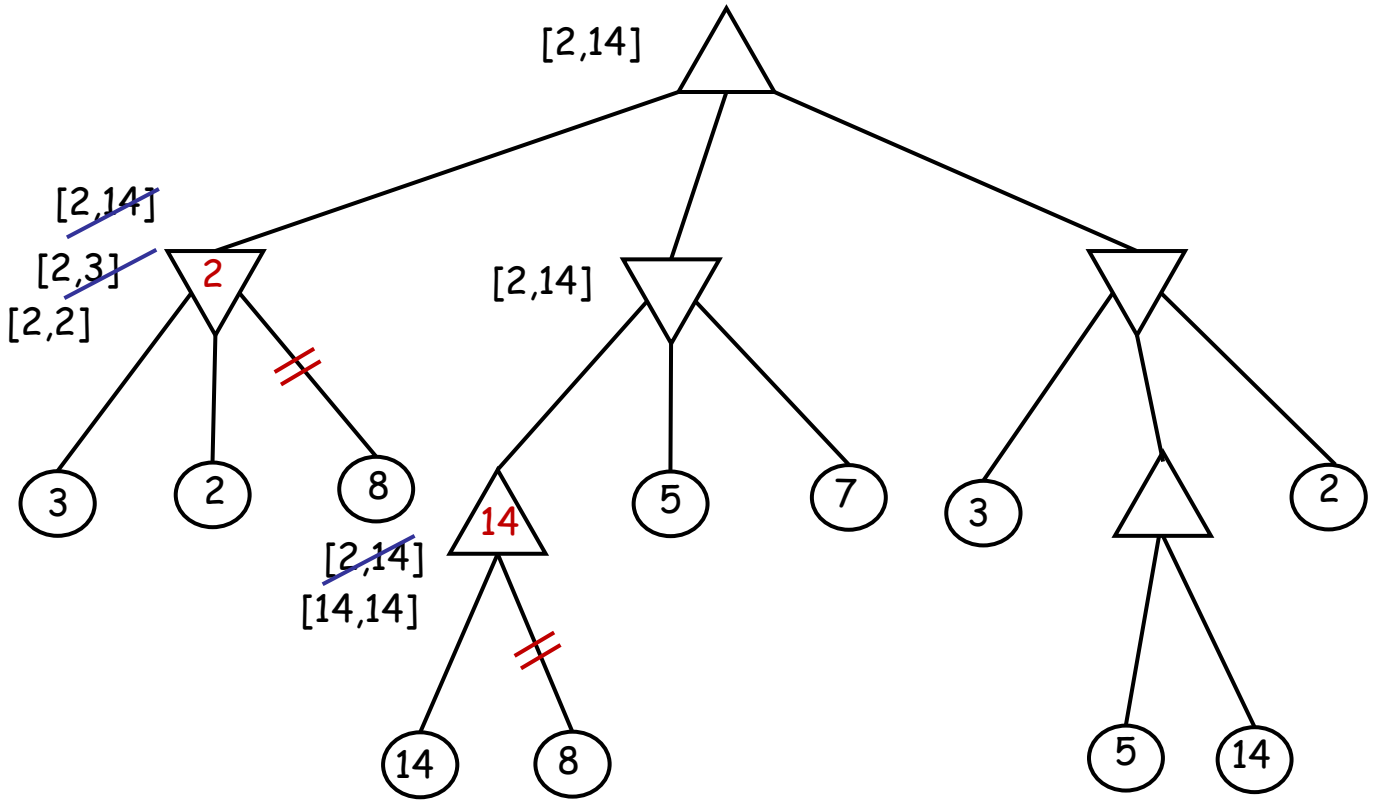
Example

- Effect of knowing the lower bound and upper bound.
  - [2, 14] for this example



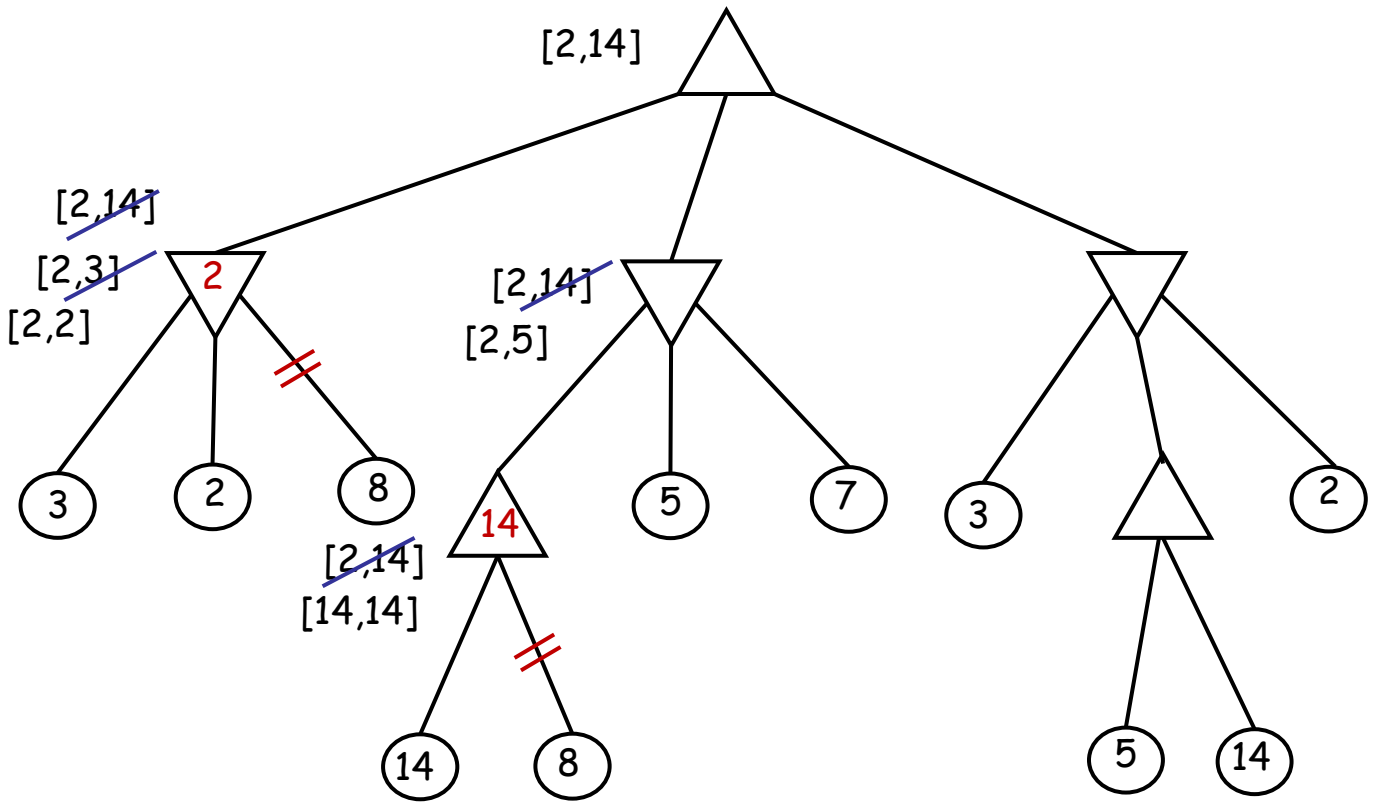
Example

- Effect of knowing the lower bound and upper bound.
  - [2, 14] for this example



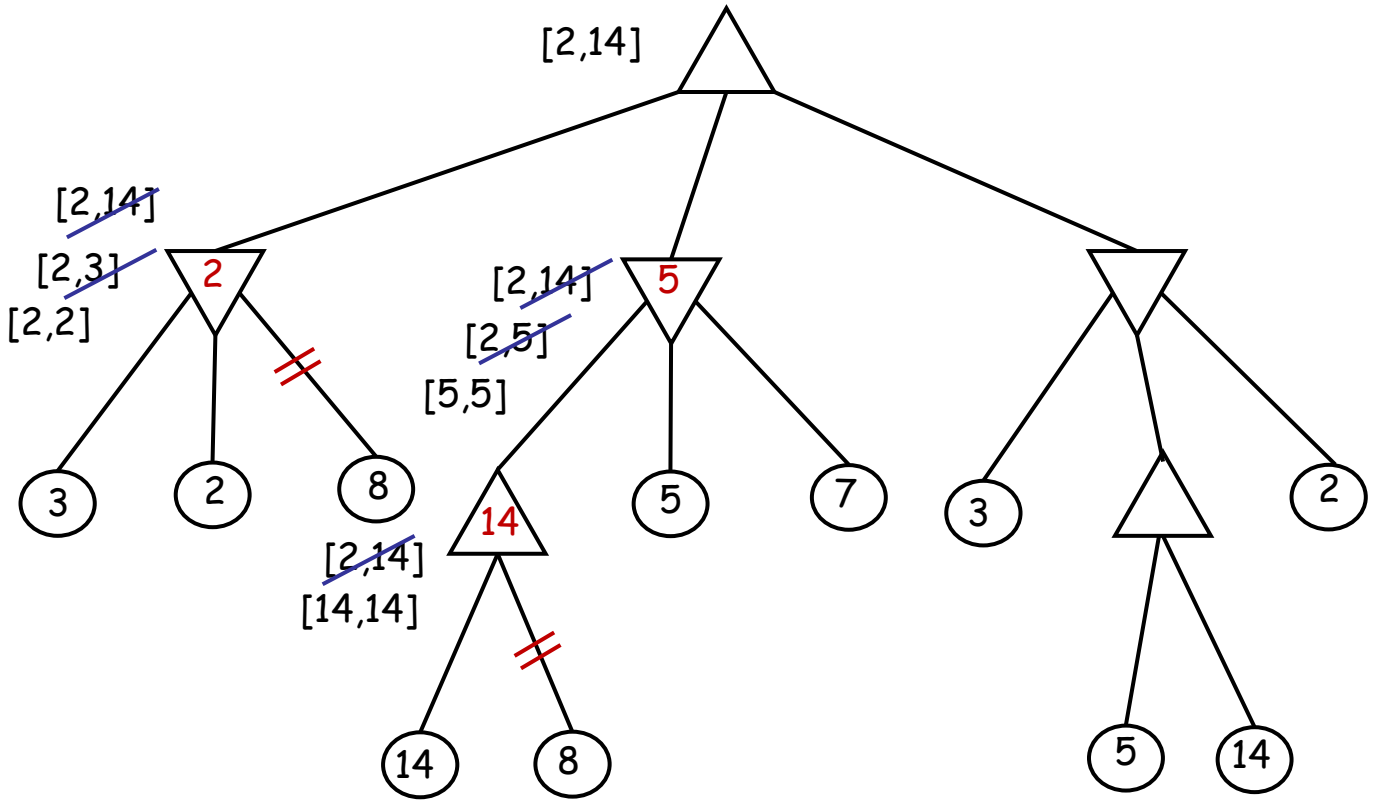
Example

- Effect of knowing the lower bound and upper bound.
  - [2, 14] for this example



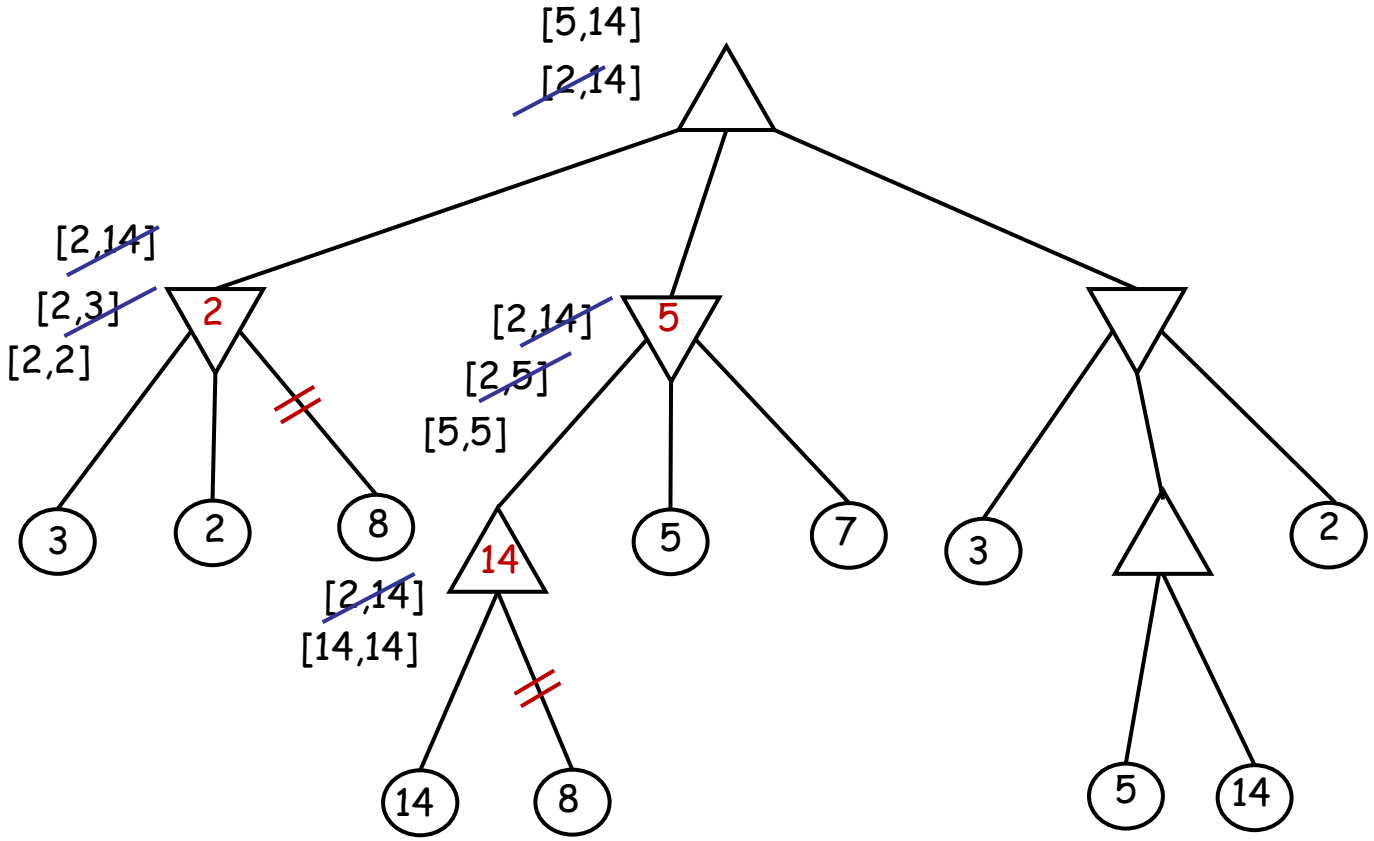
Example

- Effect of knowing the lower bound and upper bound.
  - [2, 14] for this example



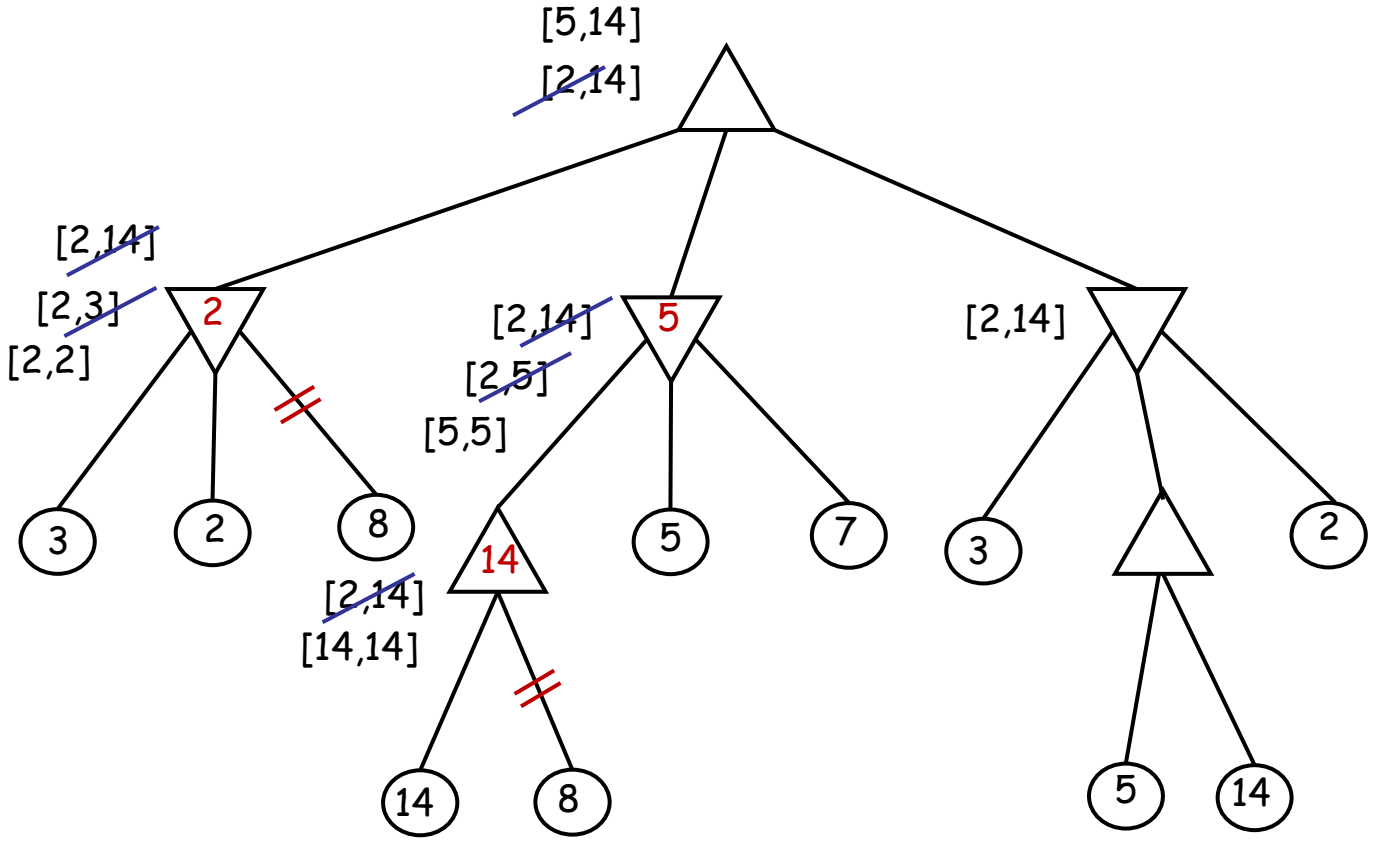
Example

- Effect of knowing the lower bound and upper bound.
  - [2, 14] for this example



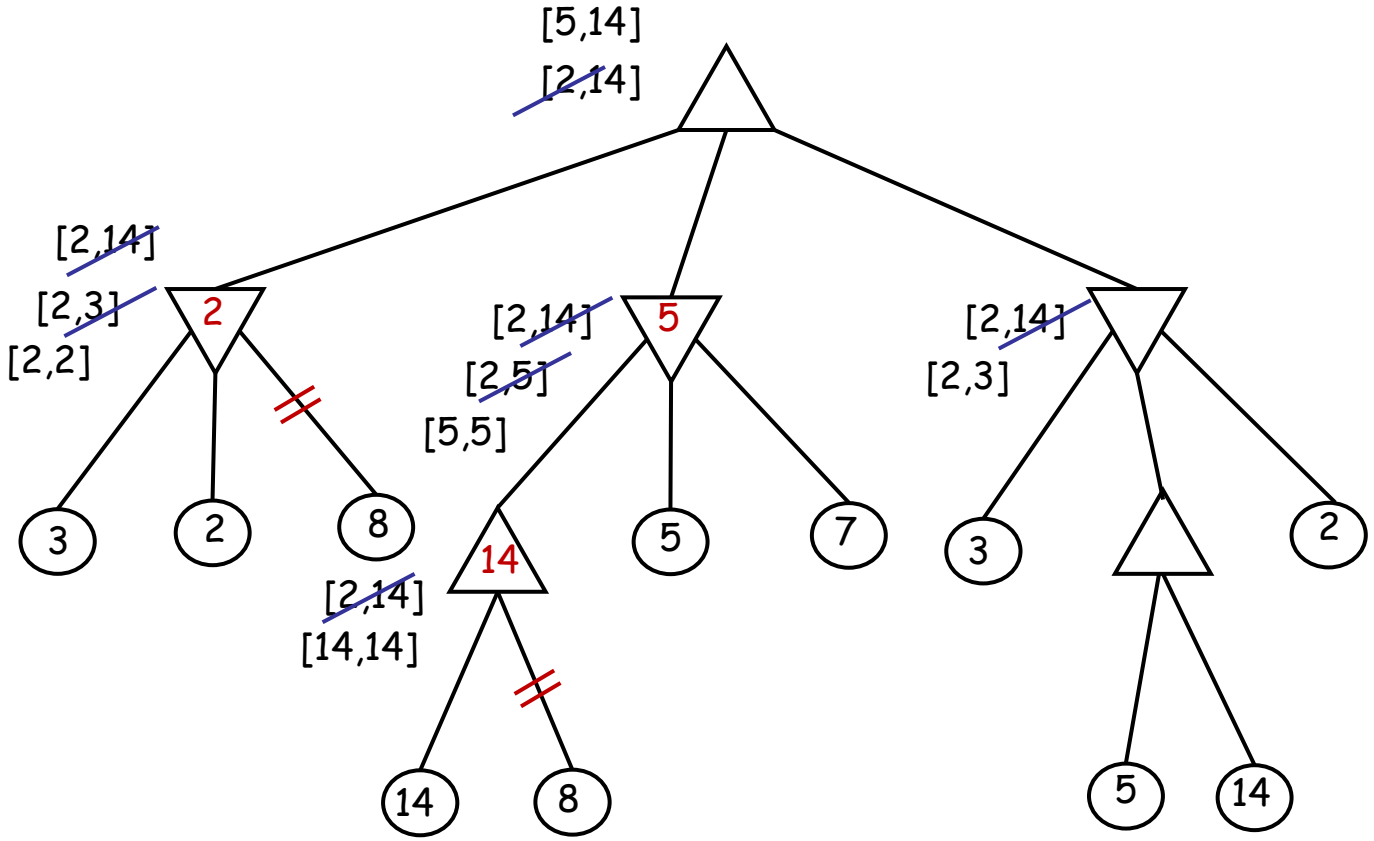
Example

- Effect of knowing the lower bound and upper bound.
  - [2, 14] for this example



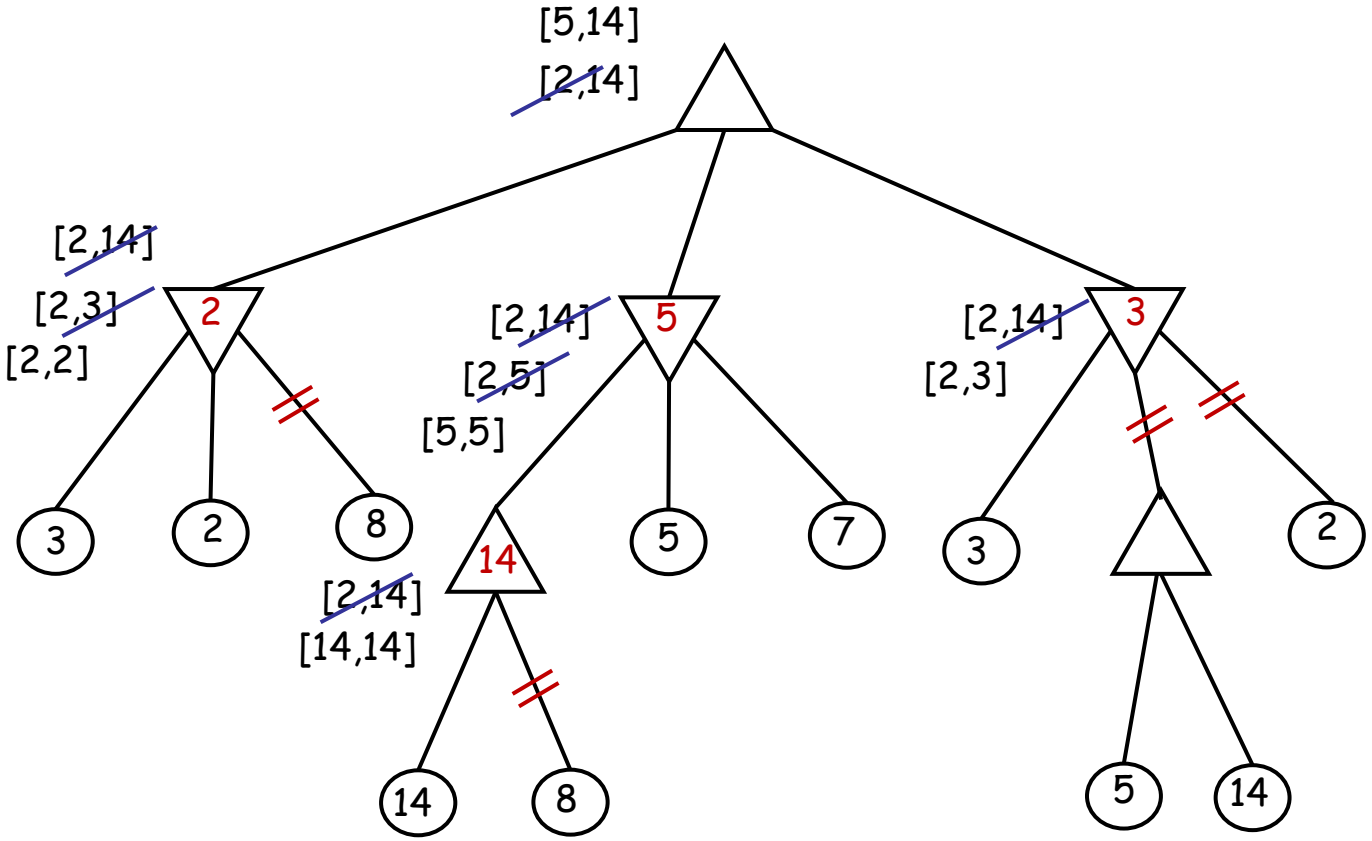
Example

- Effect of knowing the lower bound and upper bound.
  - [2, 14] for this example



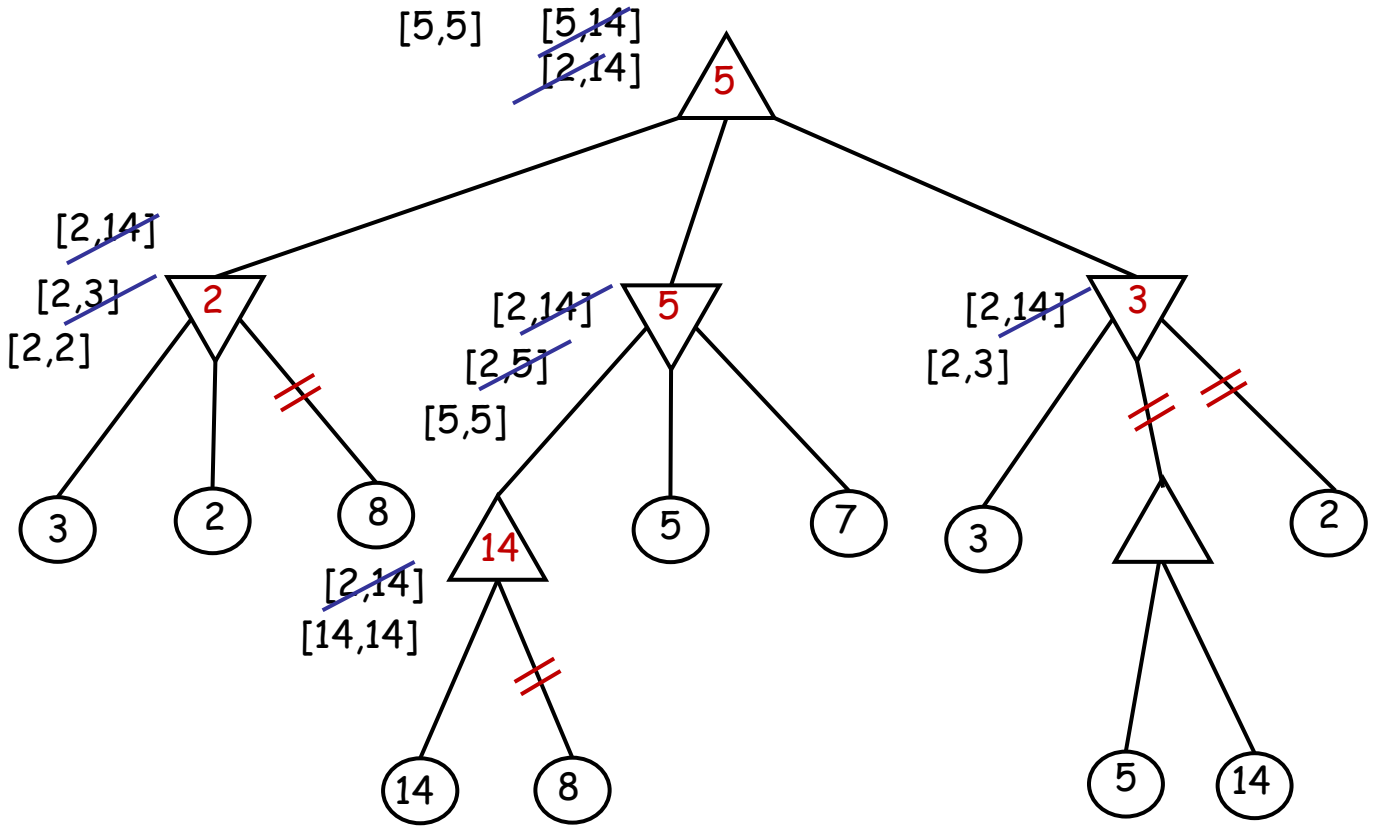
Example

- Effect of knowing the lower bound and upper bound.
  - [2, 14] for this example

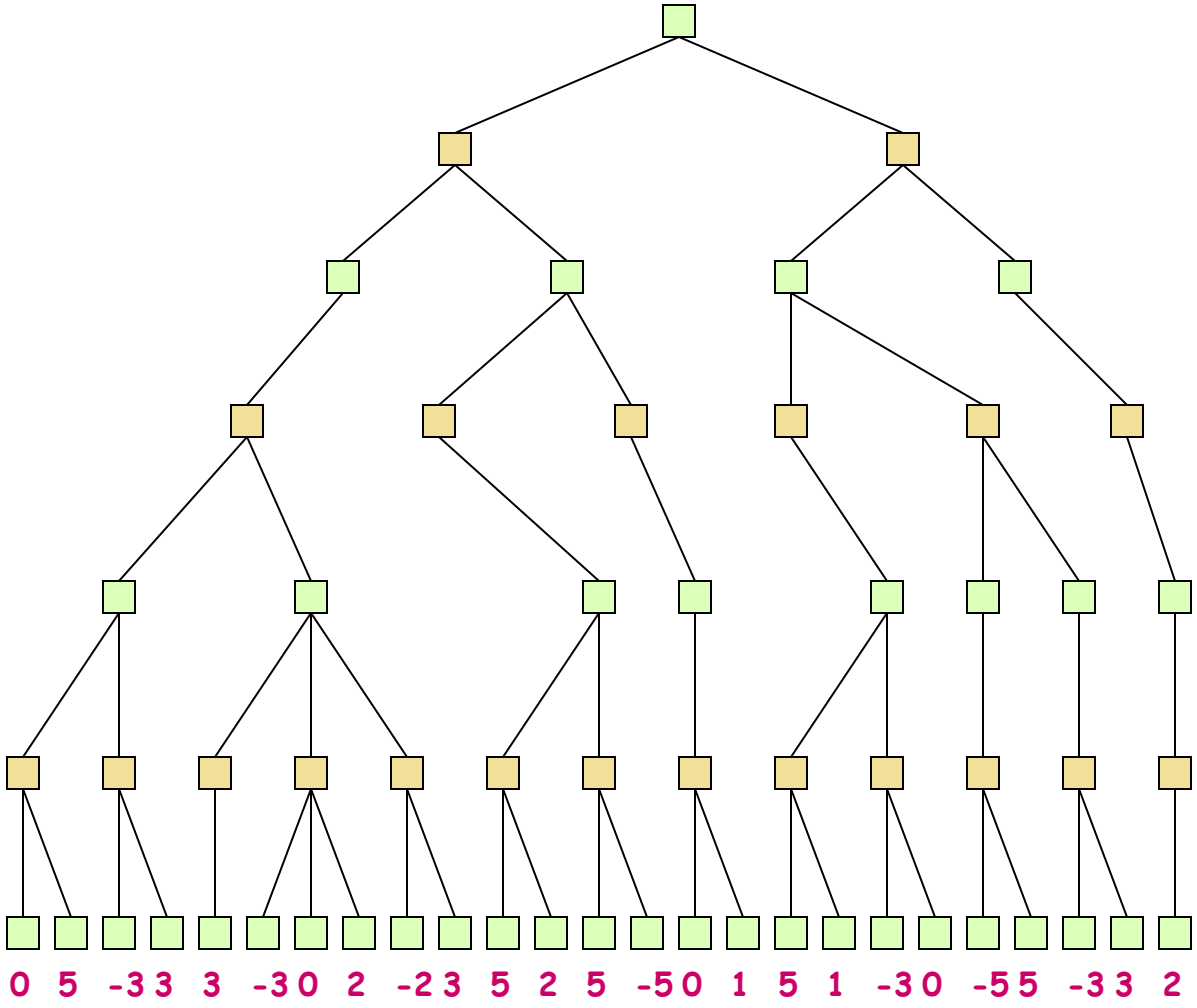


Example

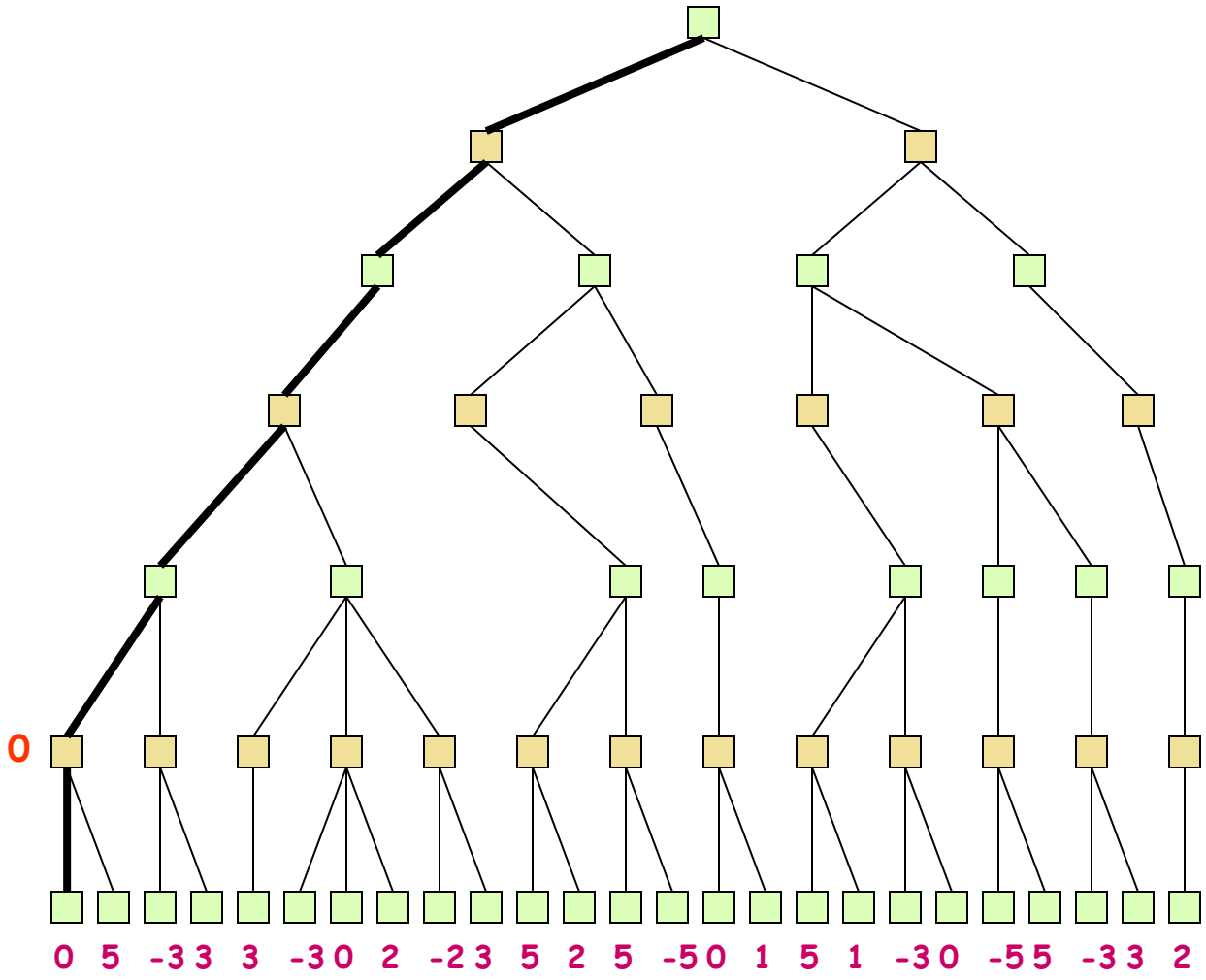
- Effect of knowing the lower bound and upper bound.
  - [2, 14] for this example



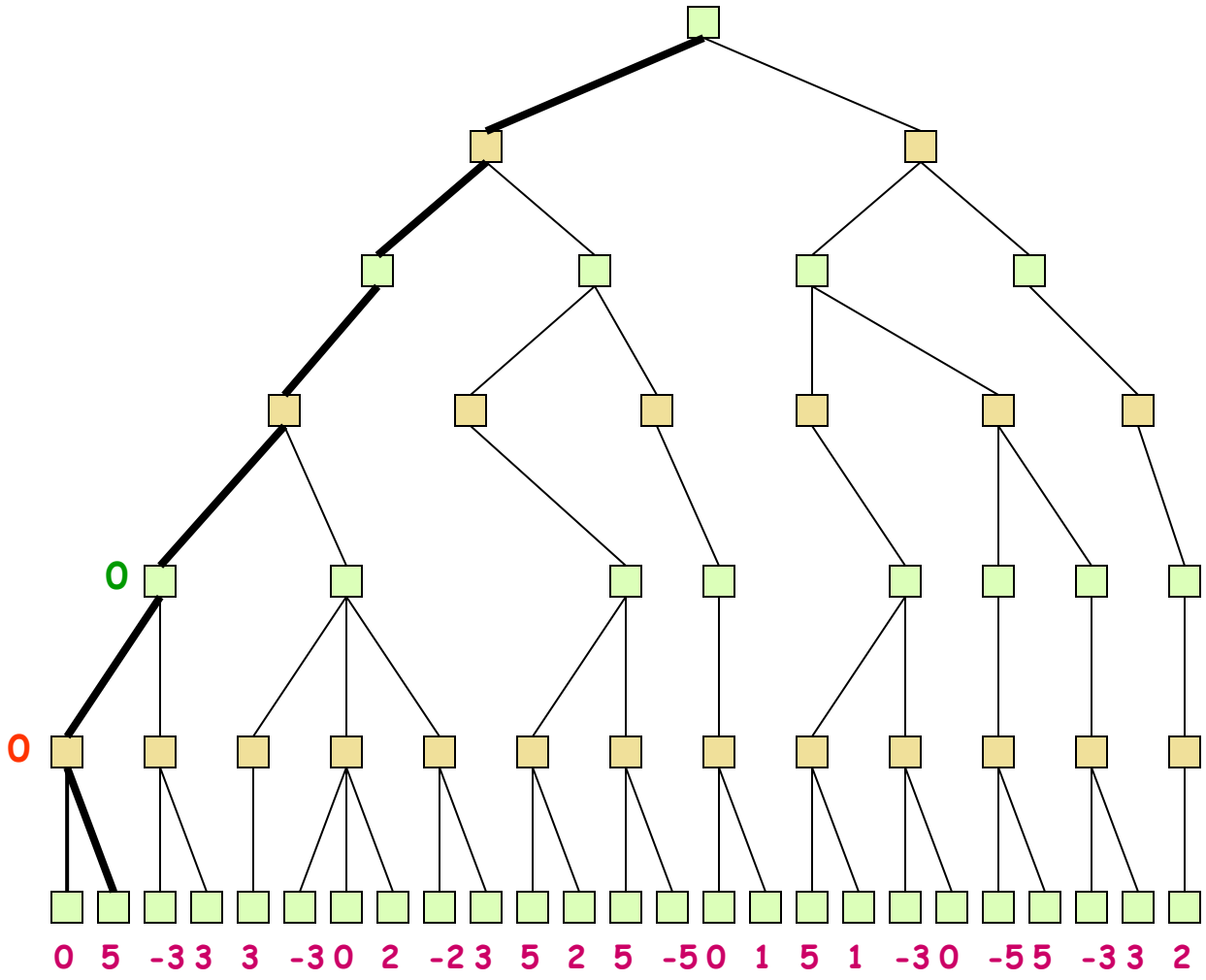
Example



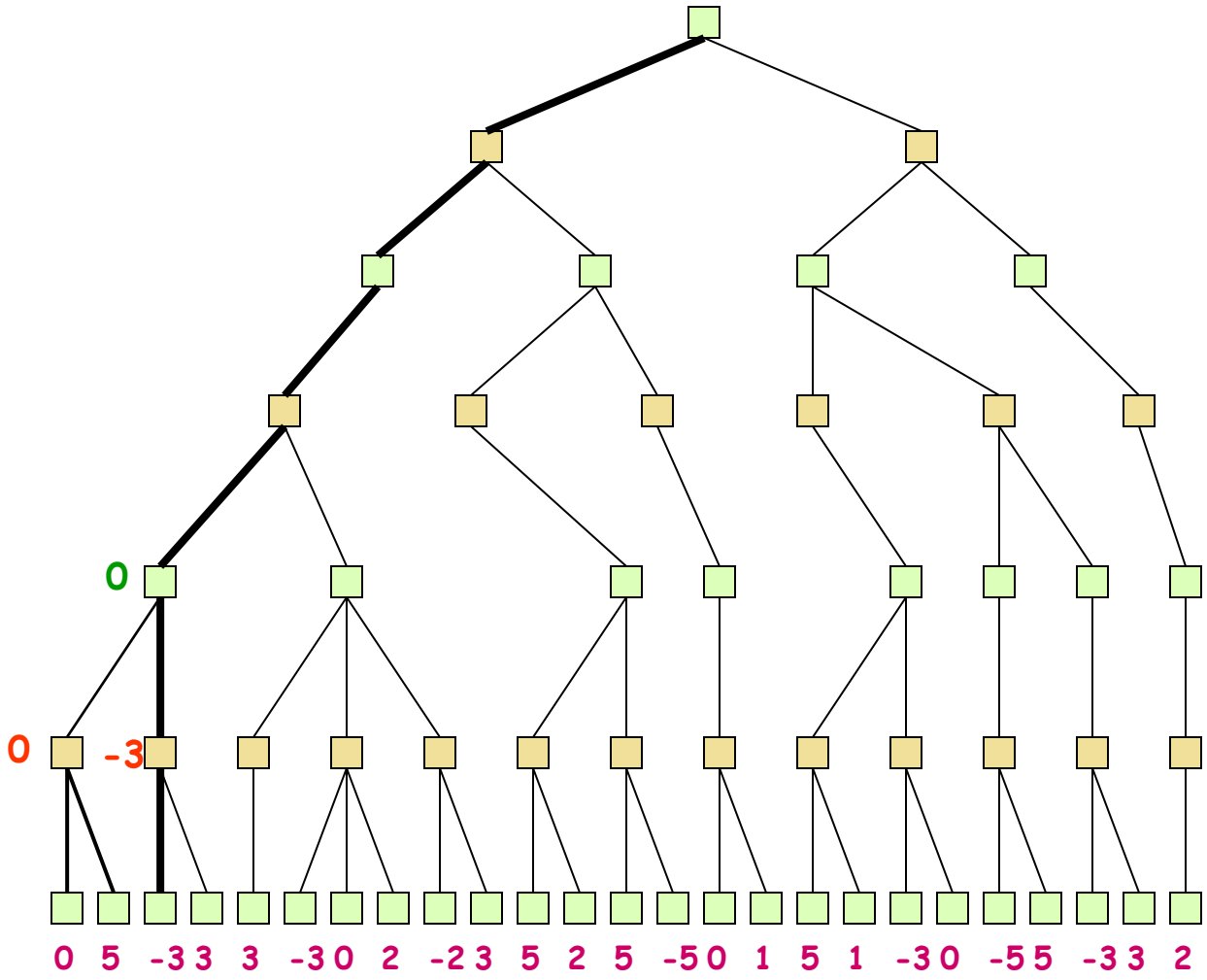
Example



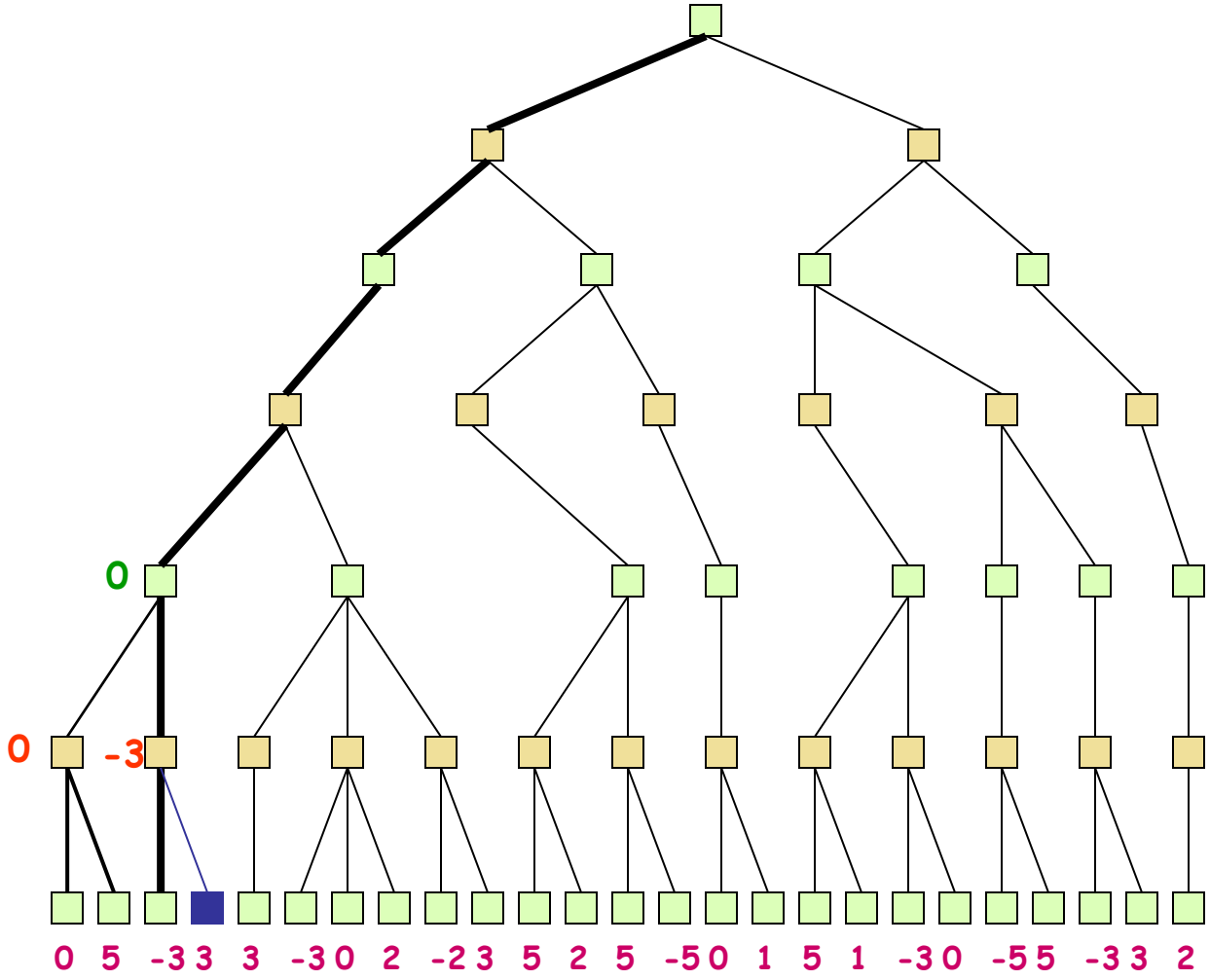
Example



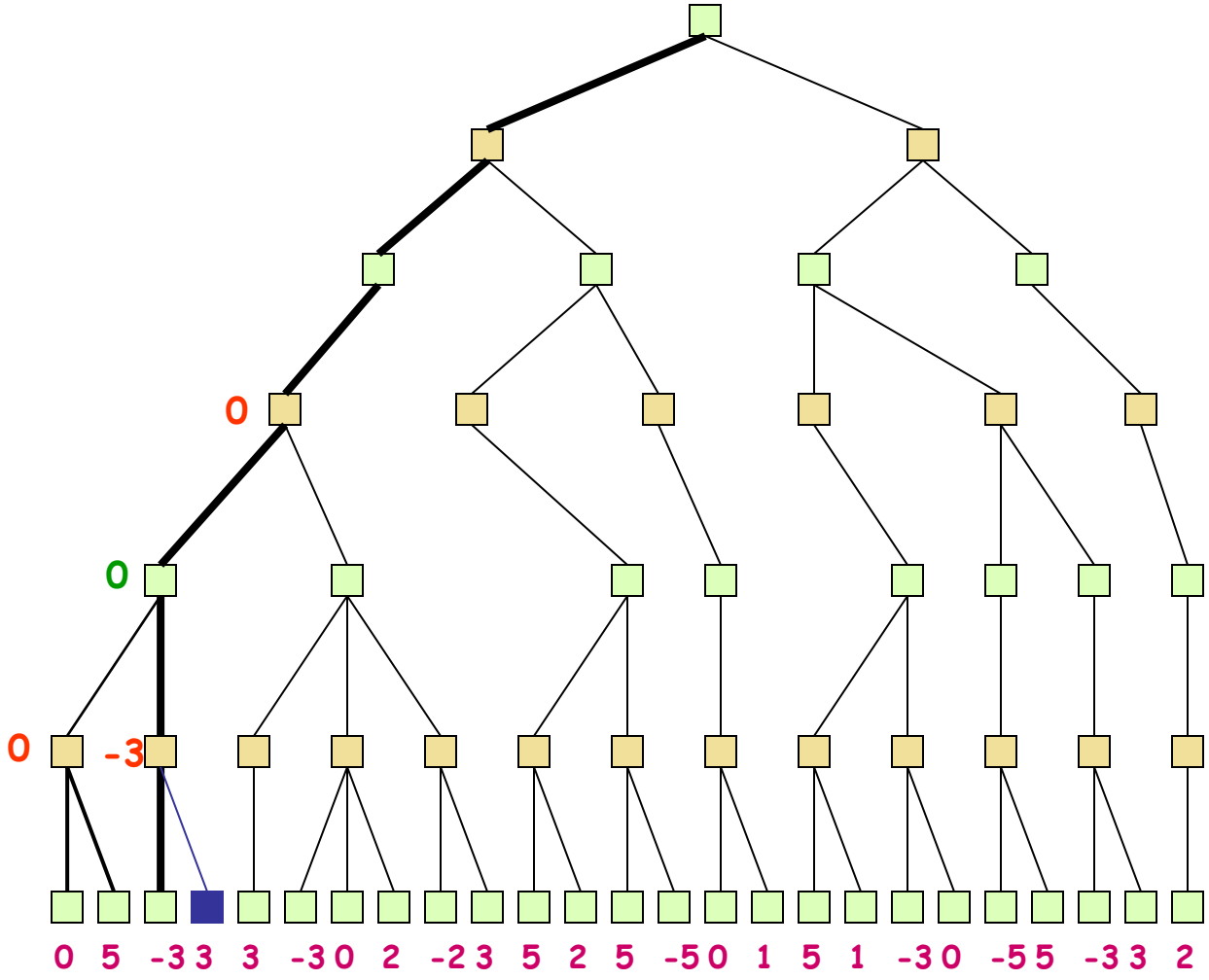
Example



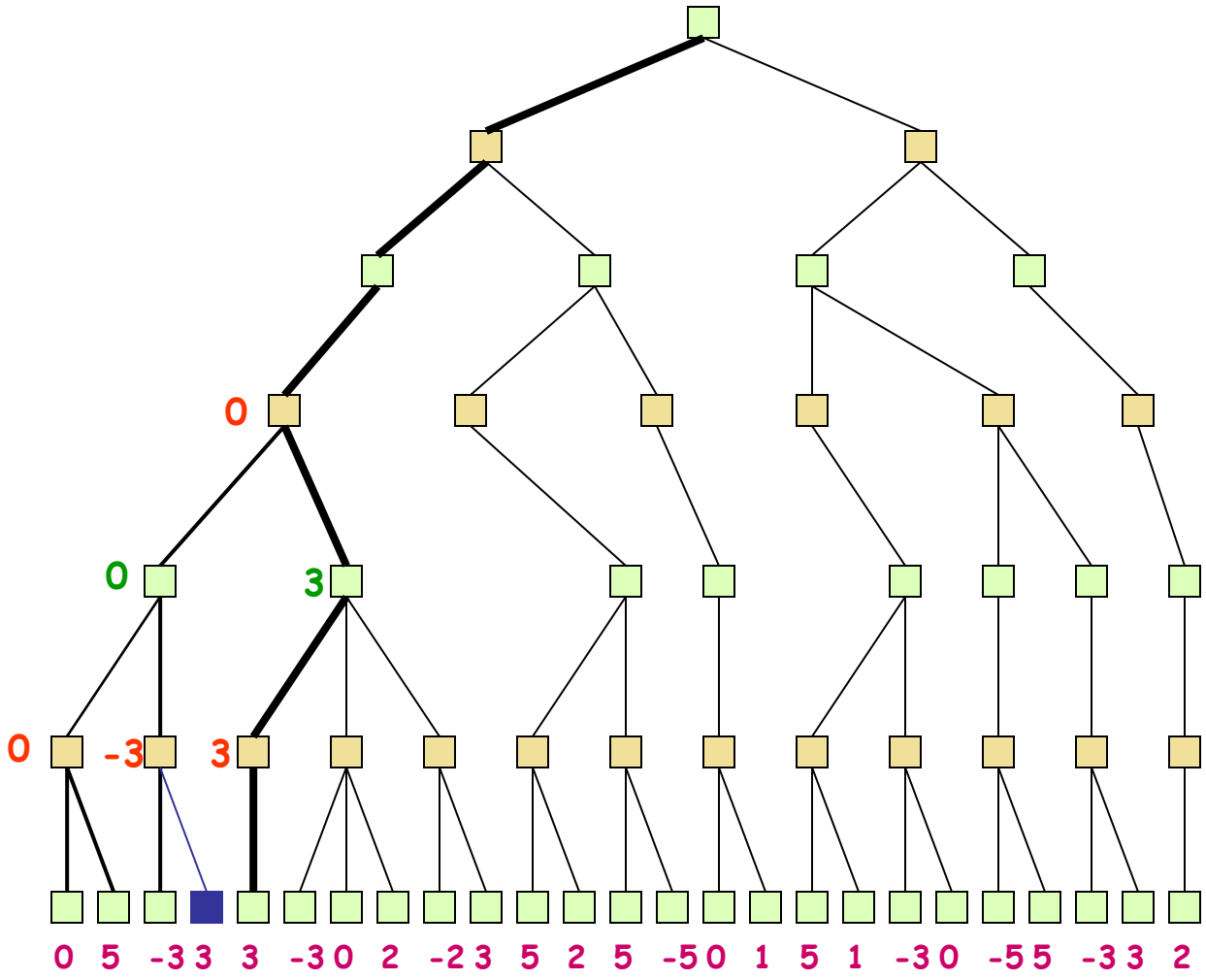
Example



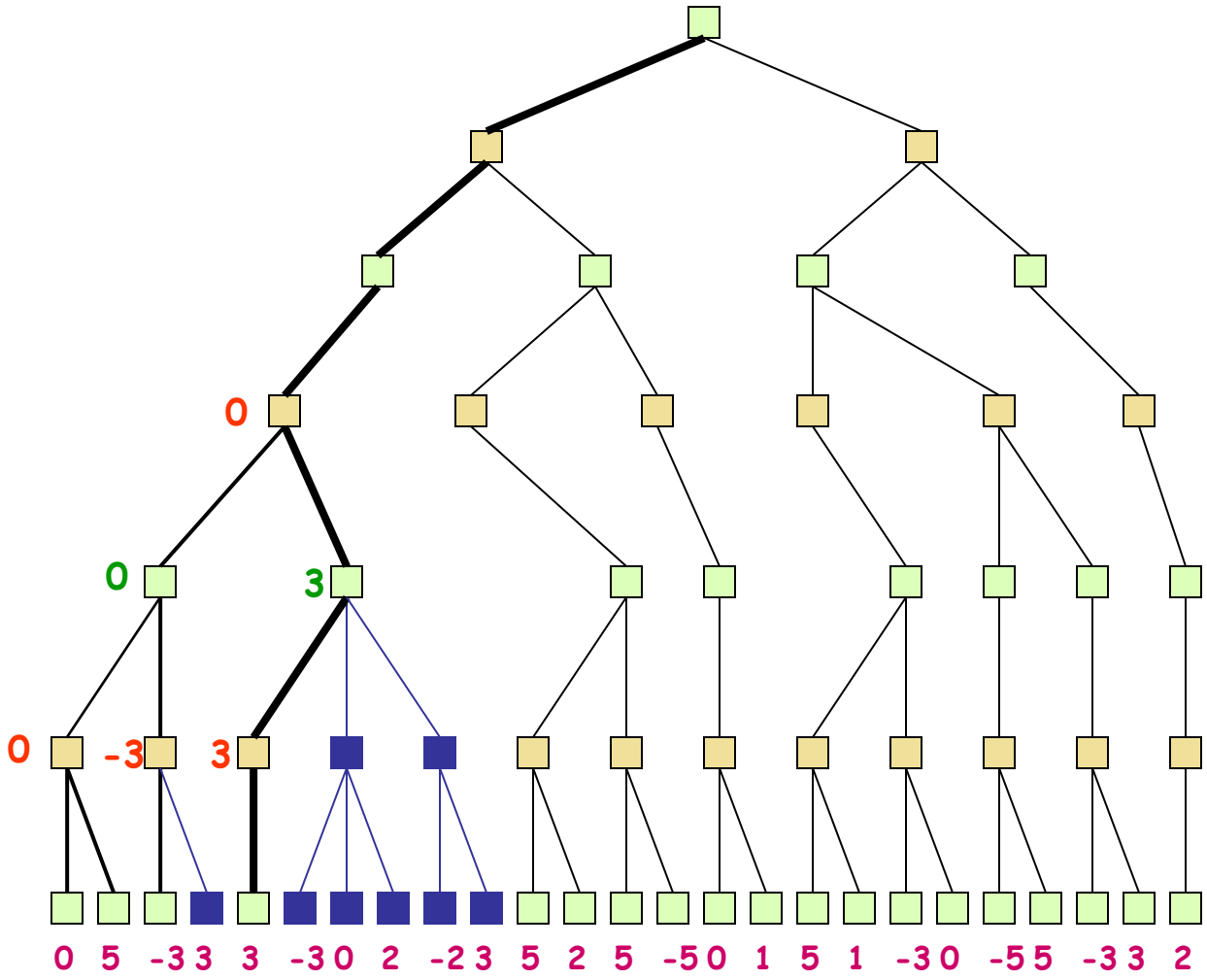
Example



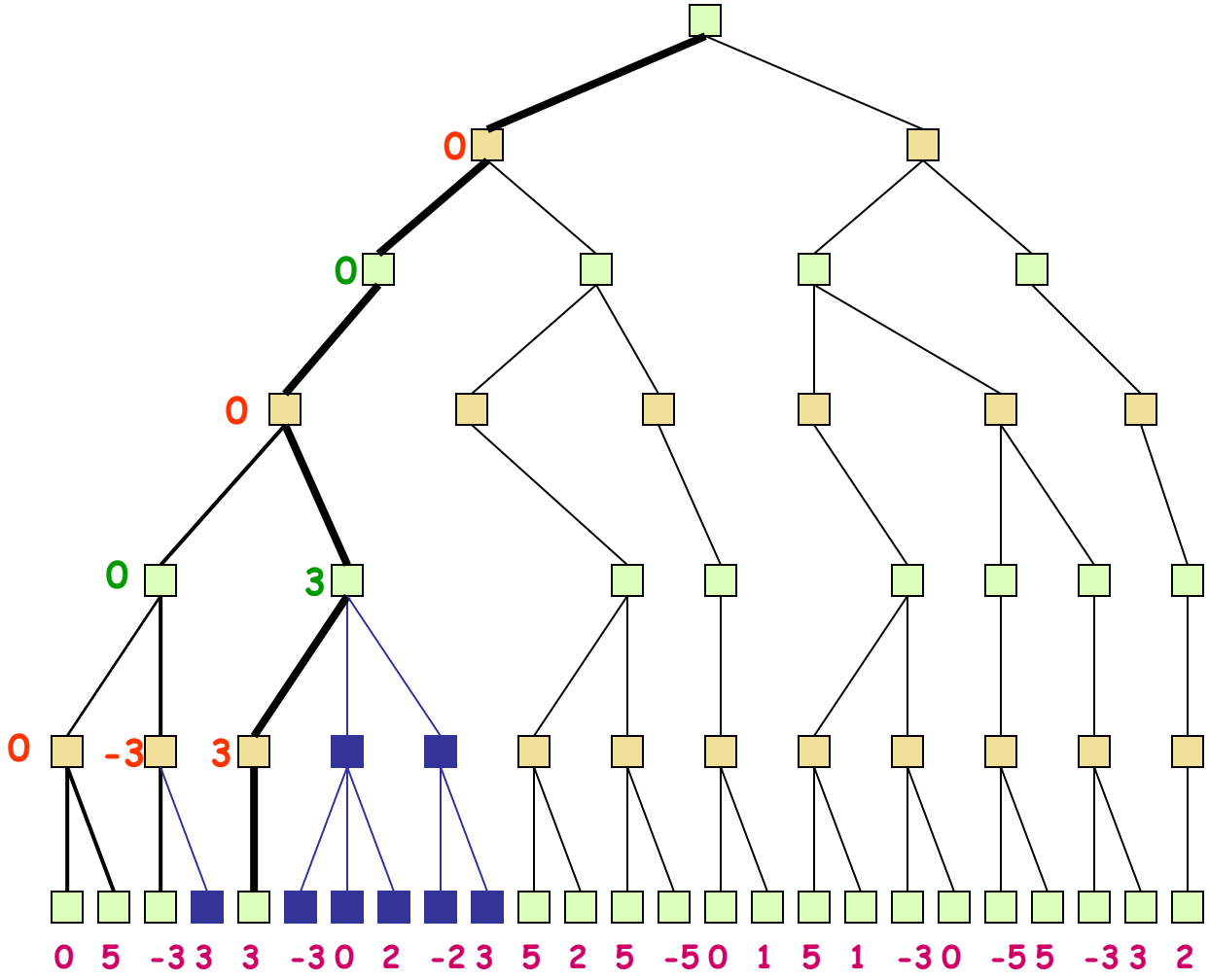
Example



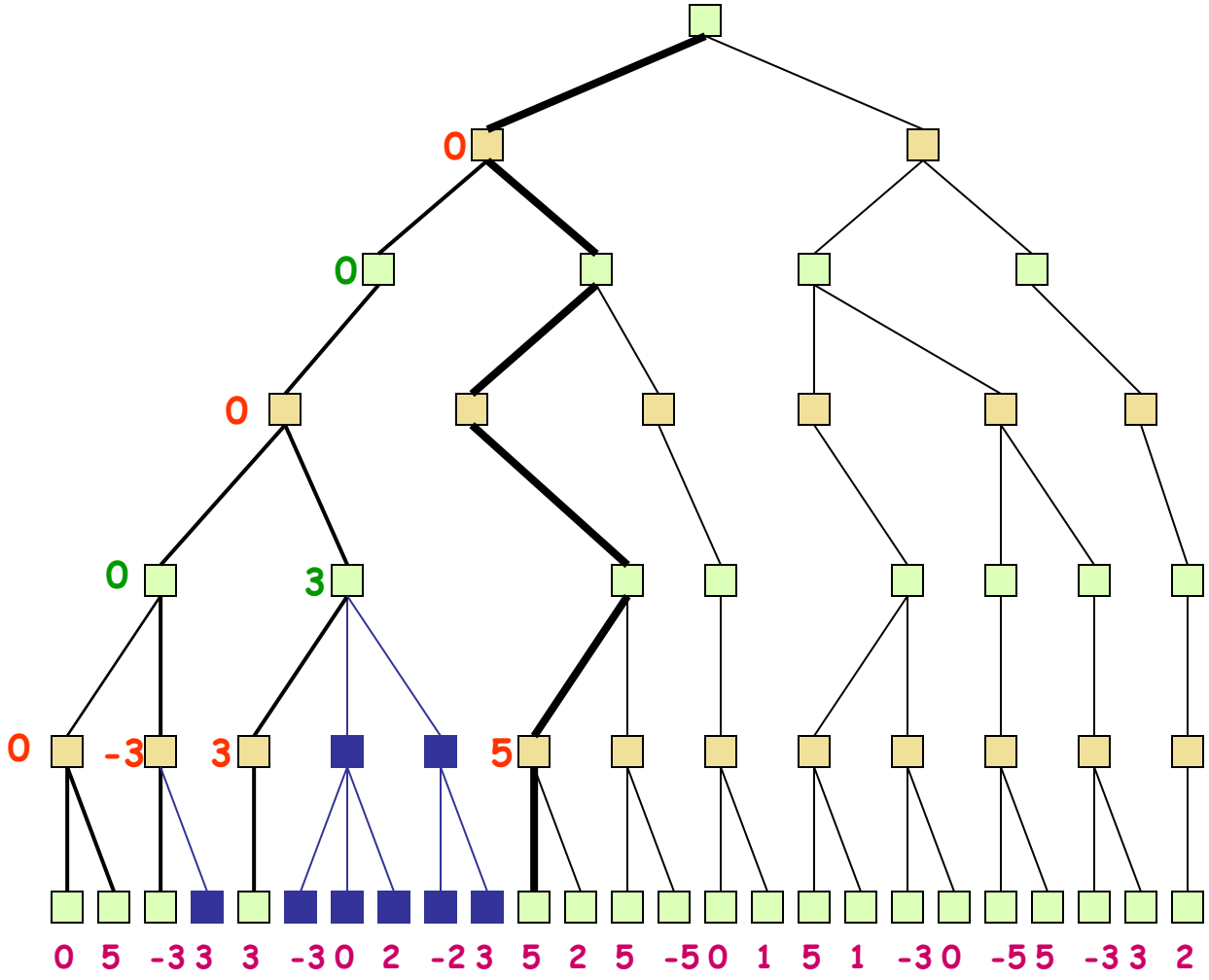
Example



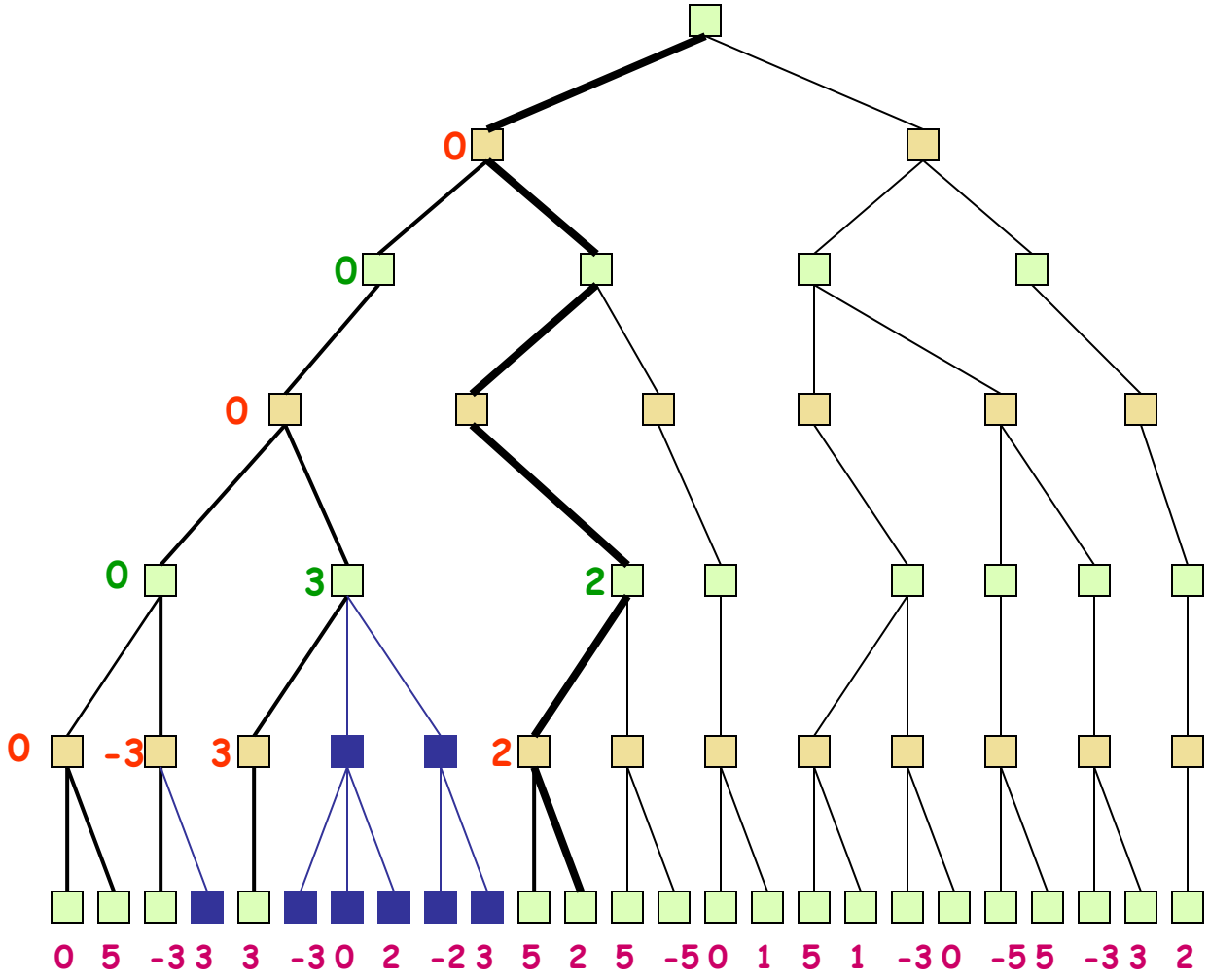
Example



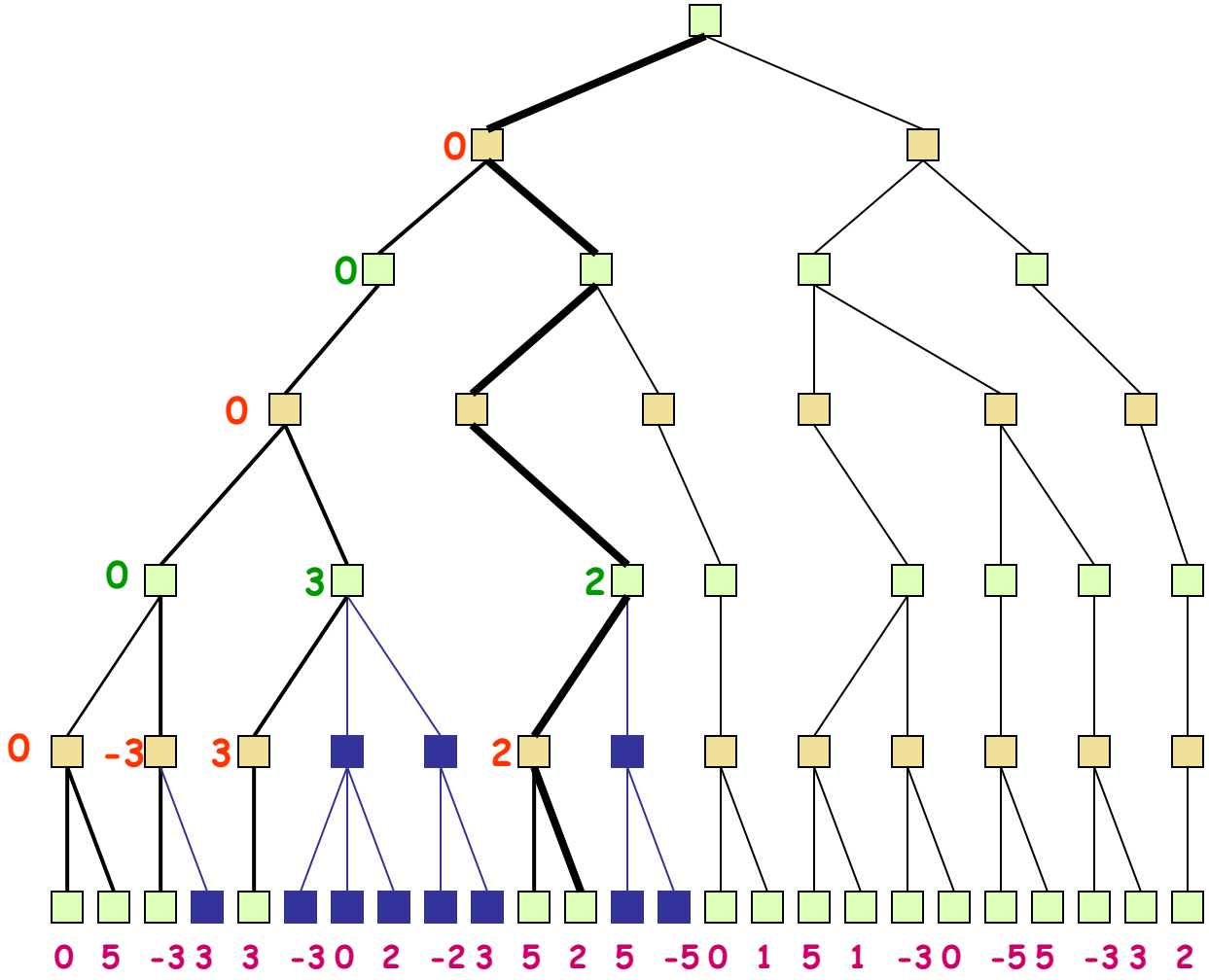
Example



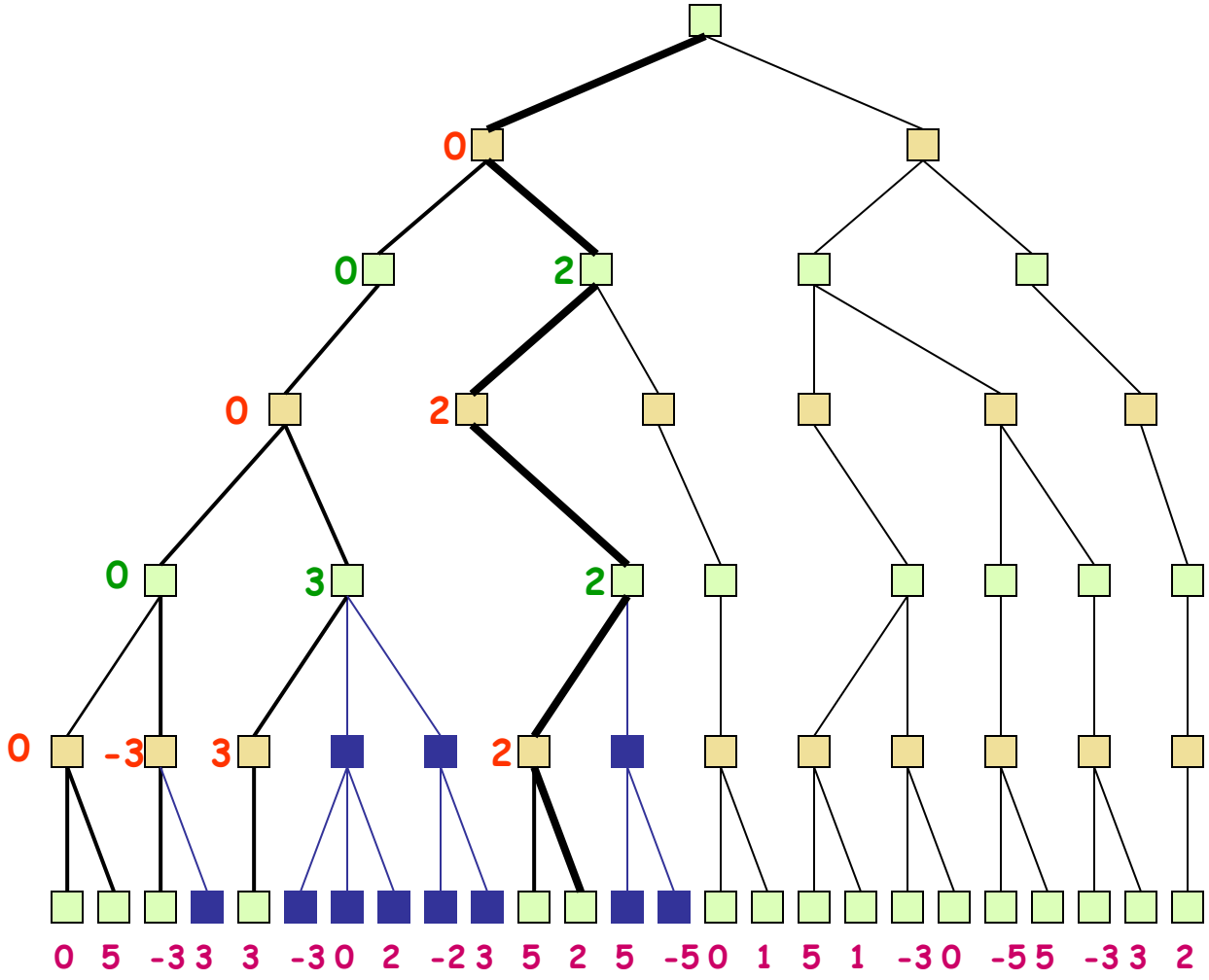
Example



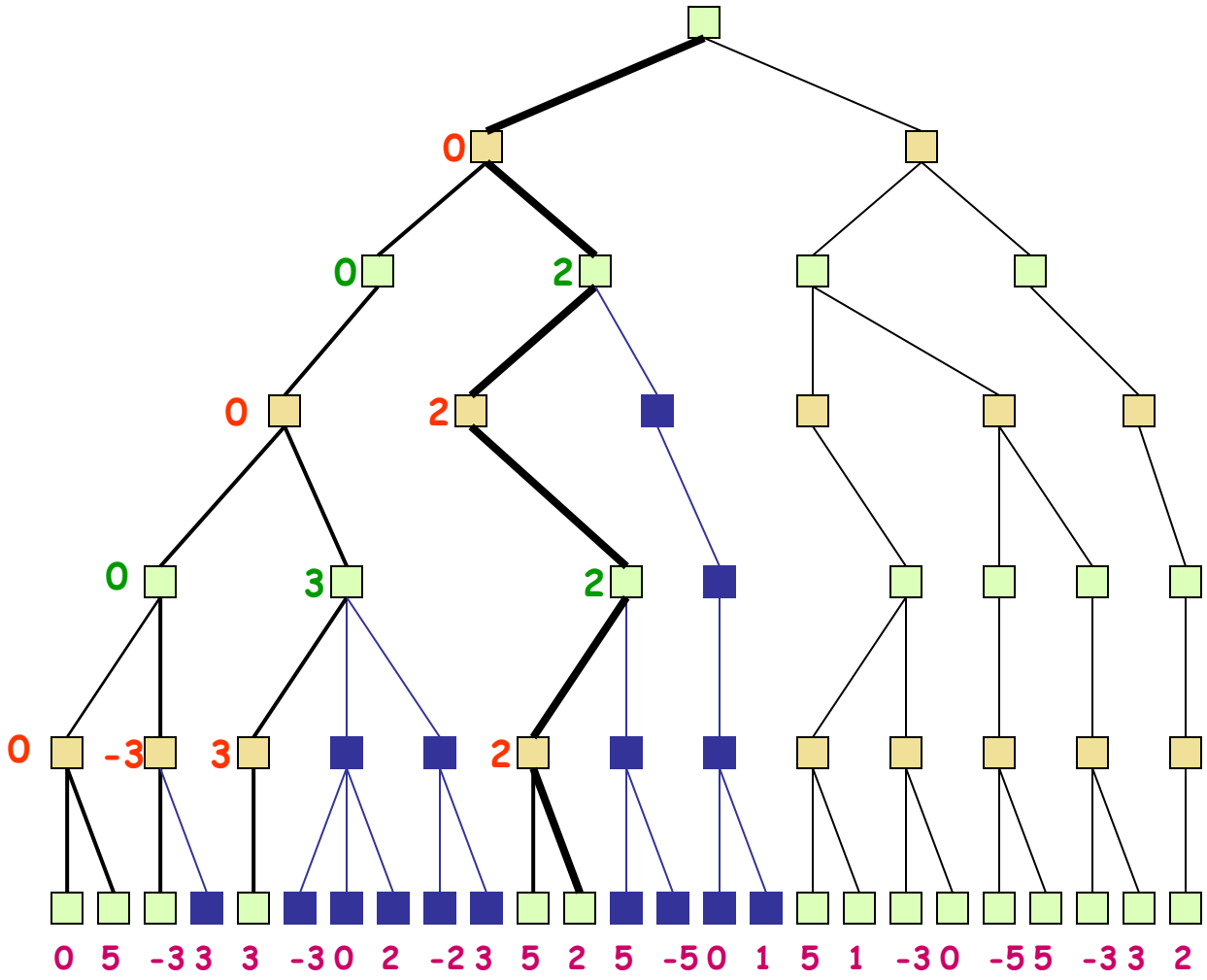
Example



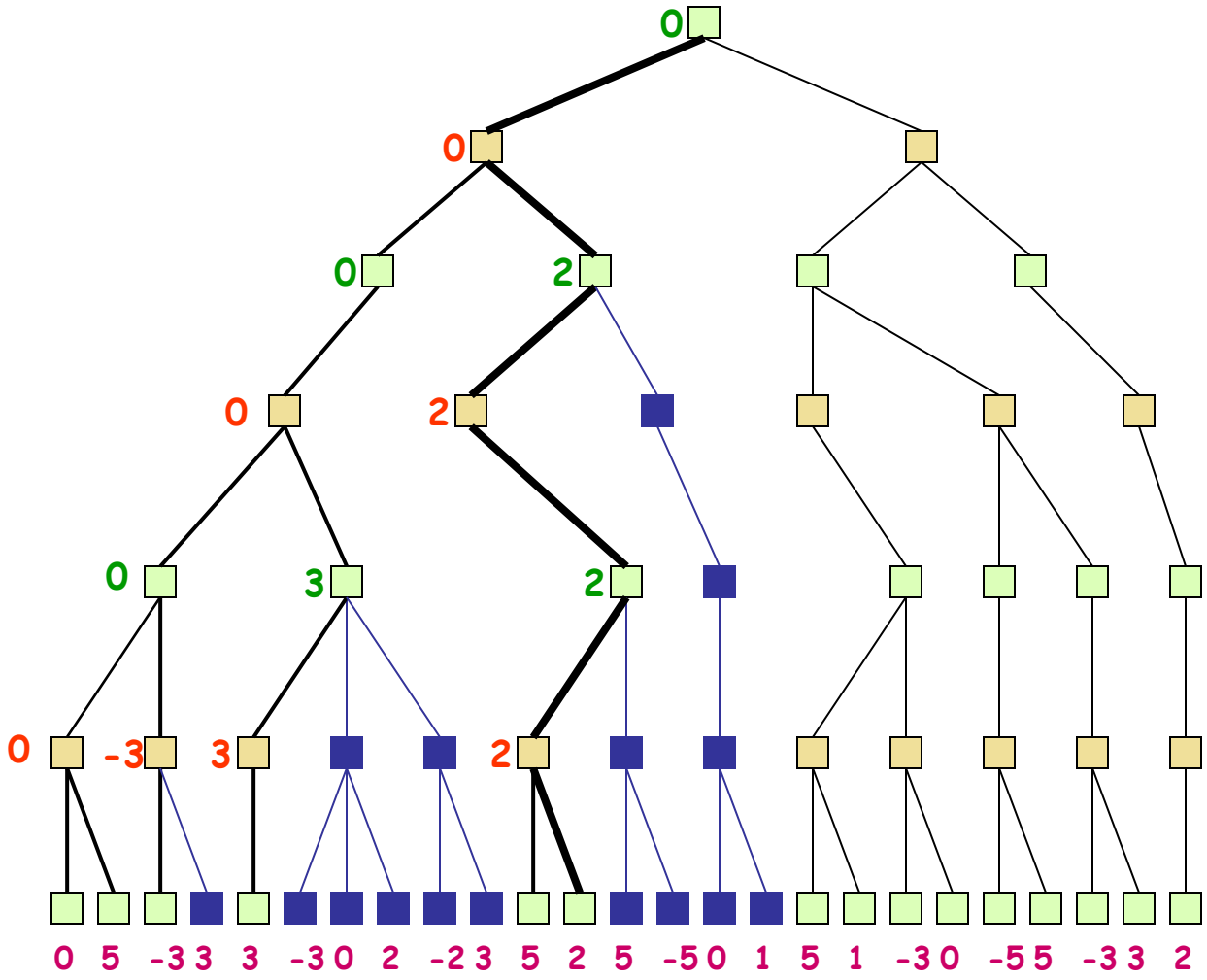
Example



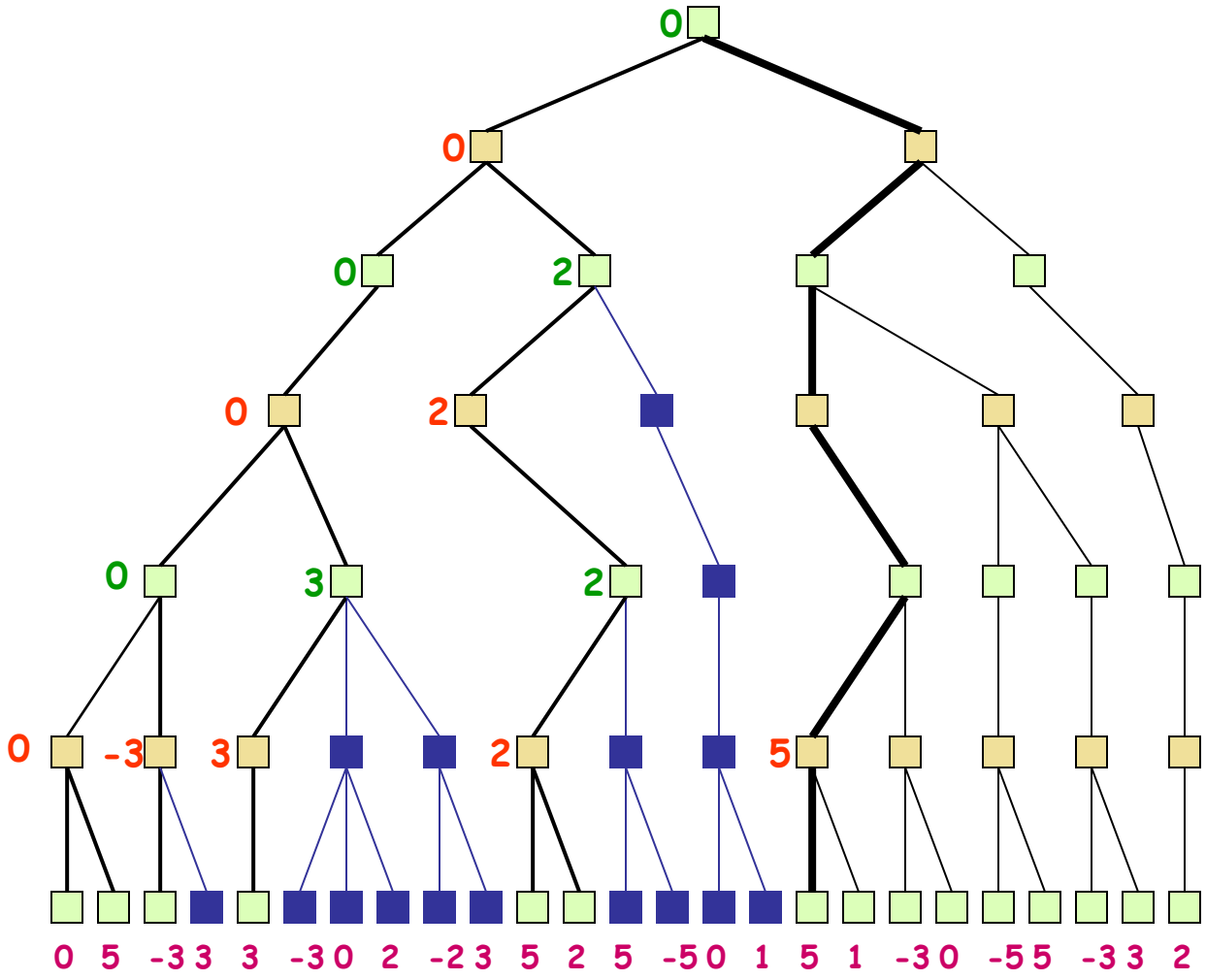
Example



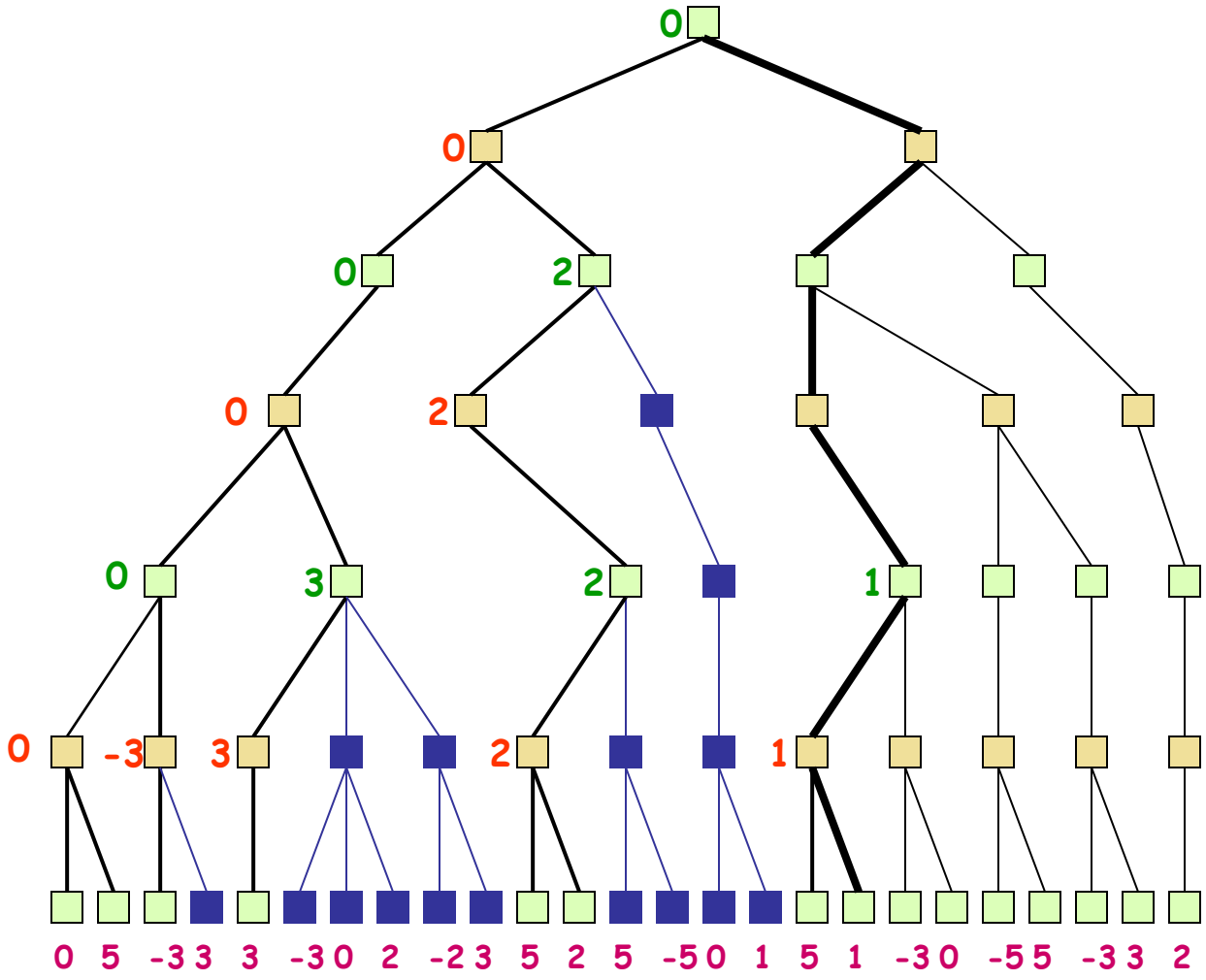
Example



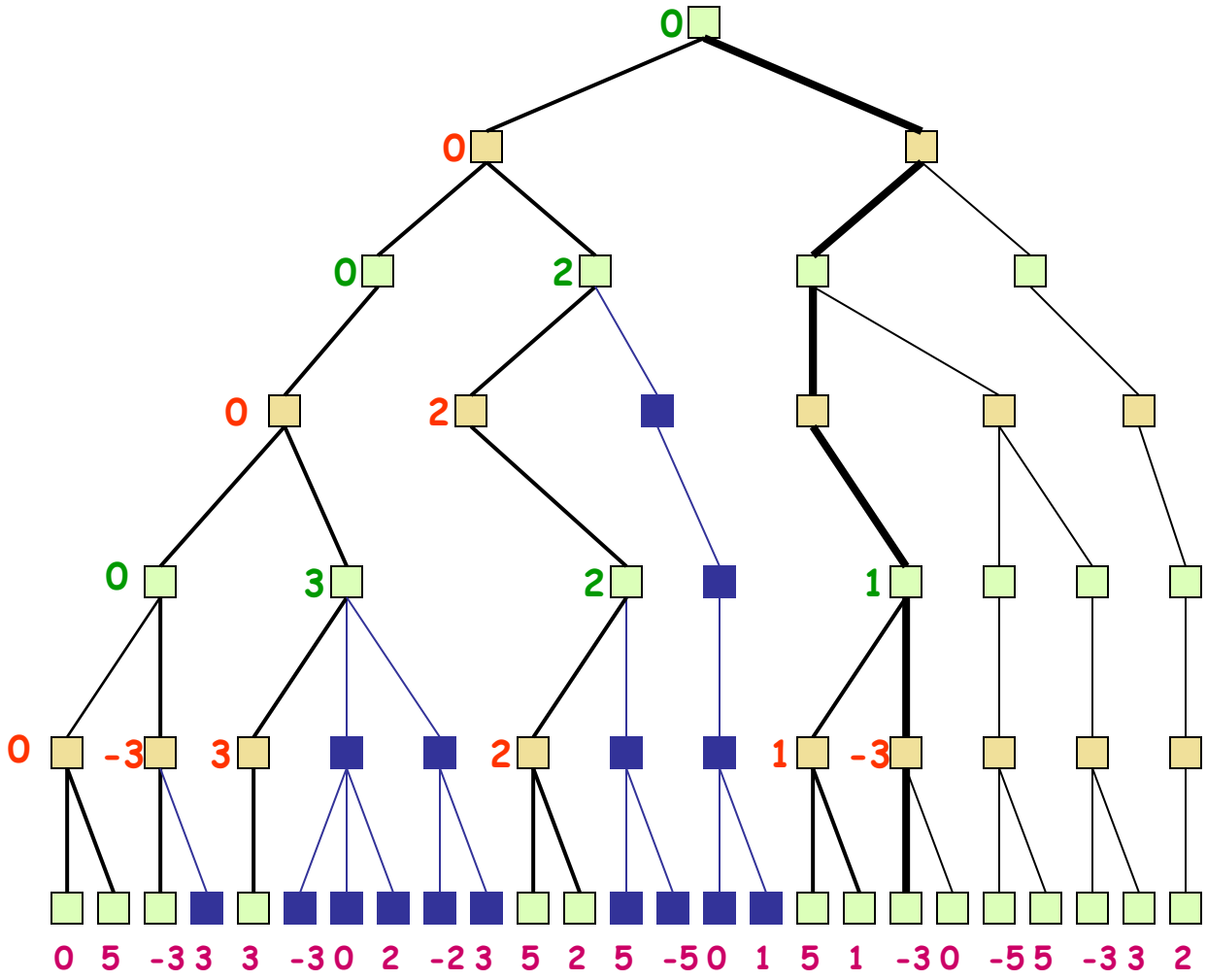
Example



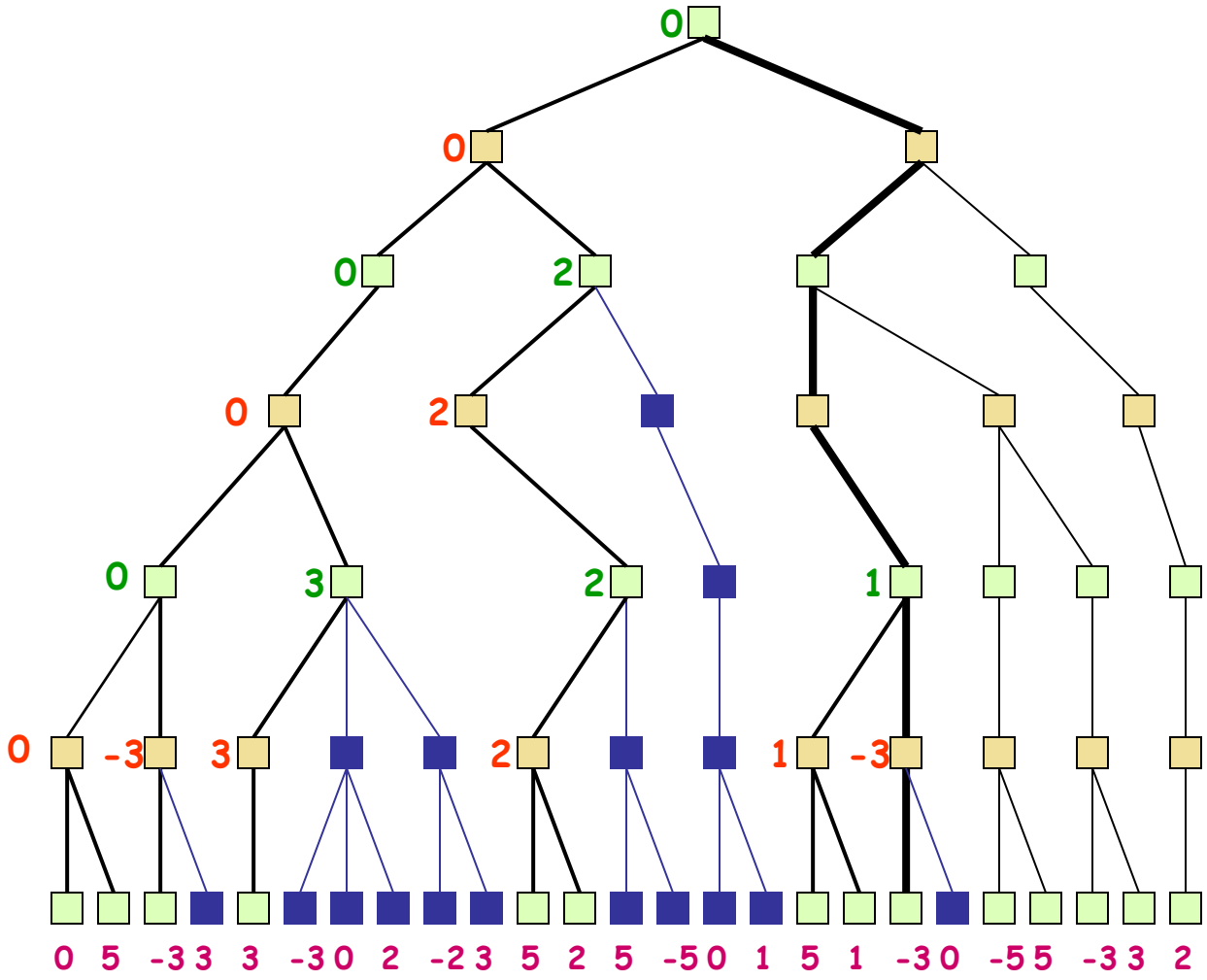
Example



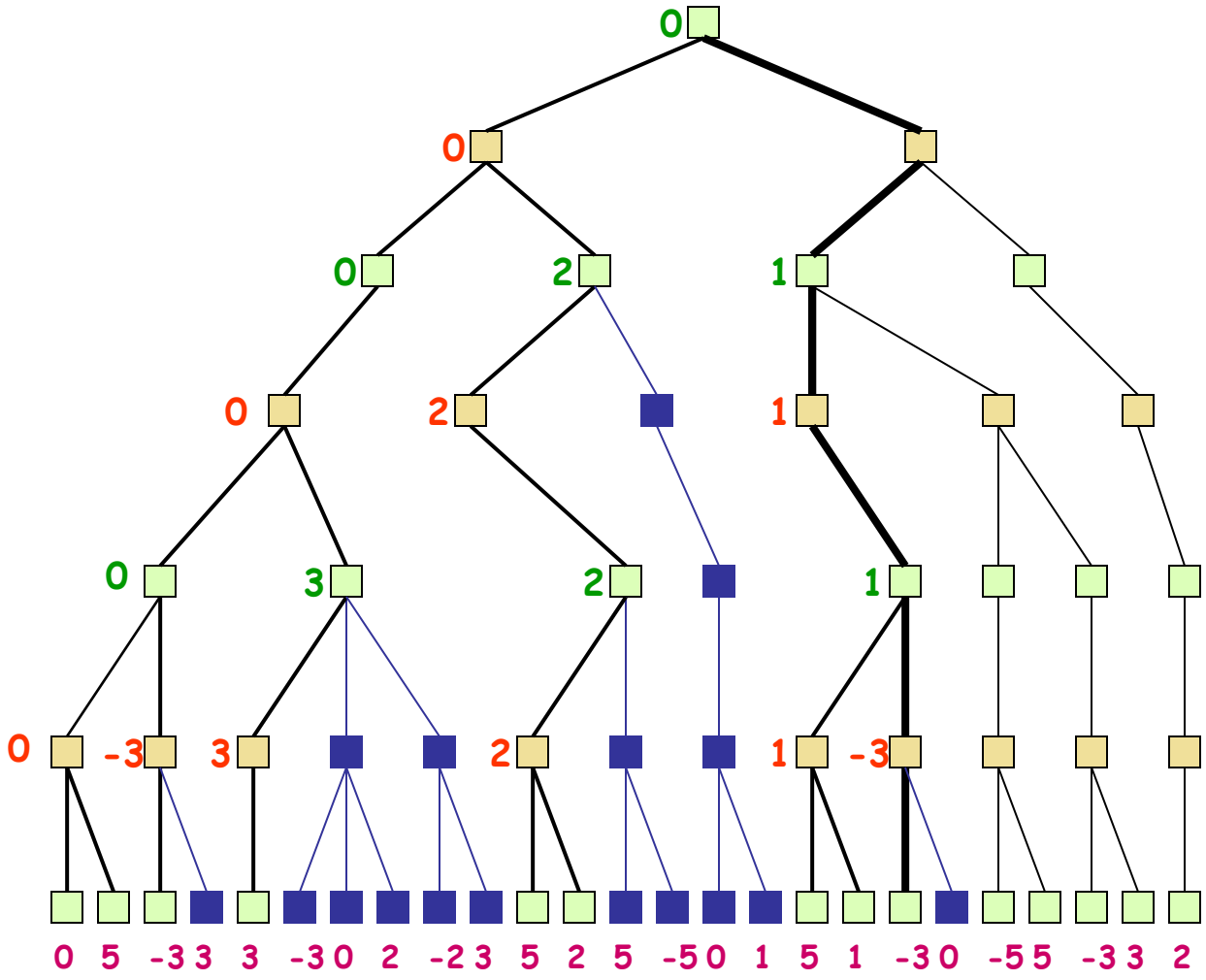
Example



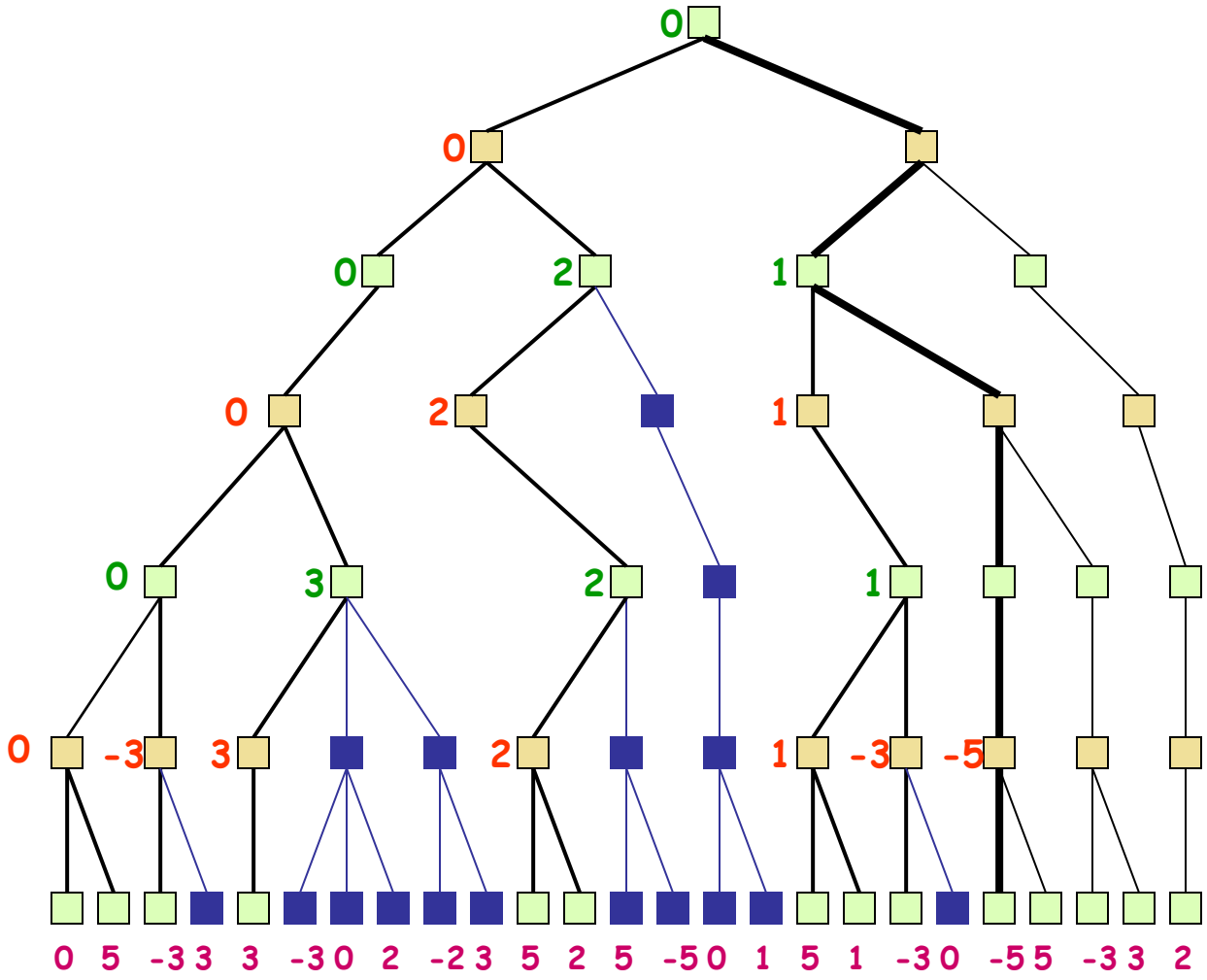
Example



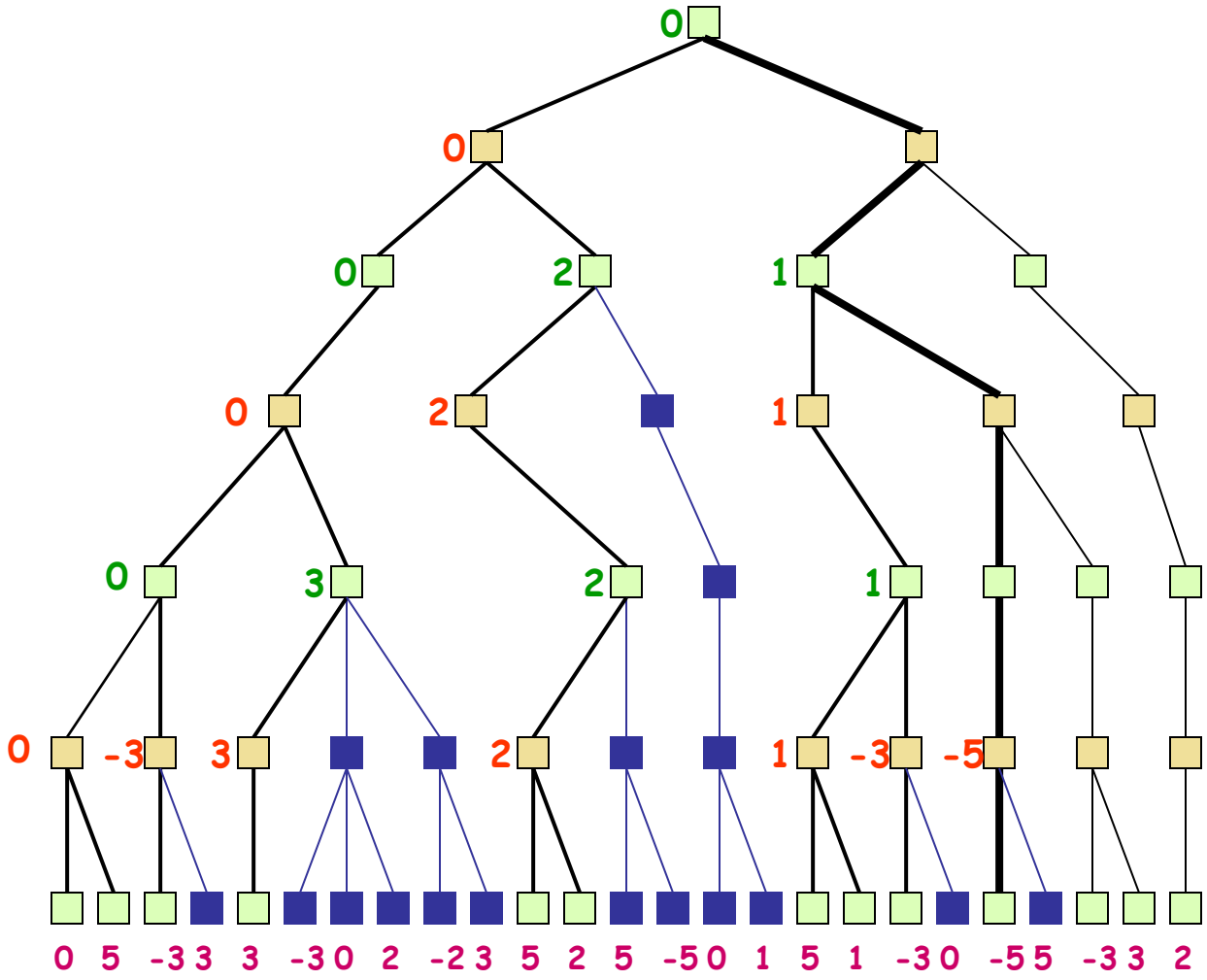
Example



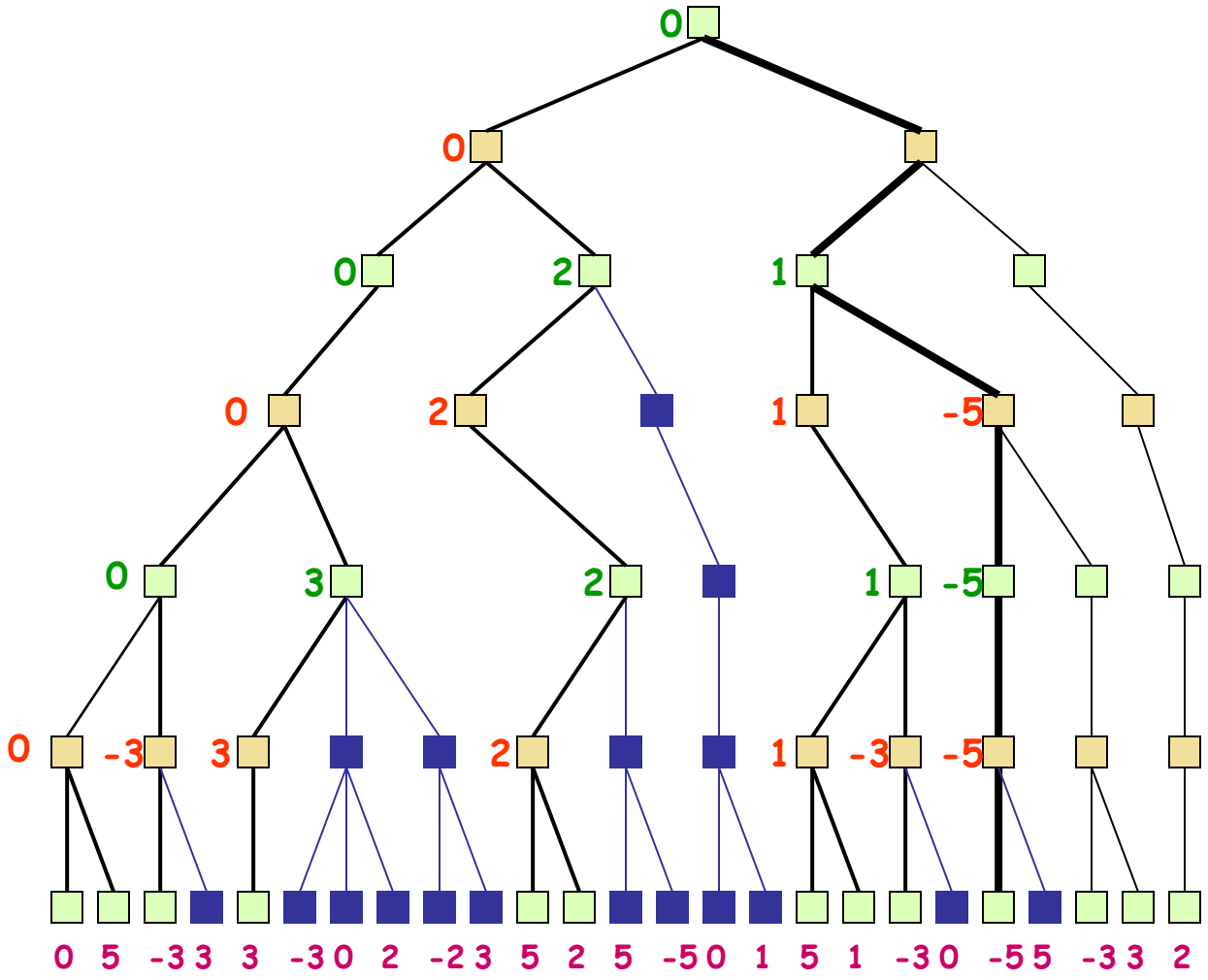
Example



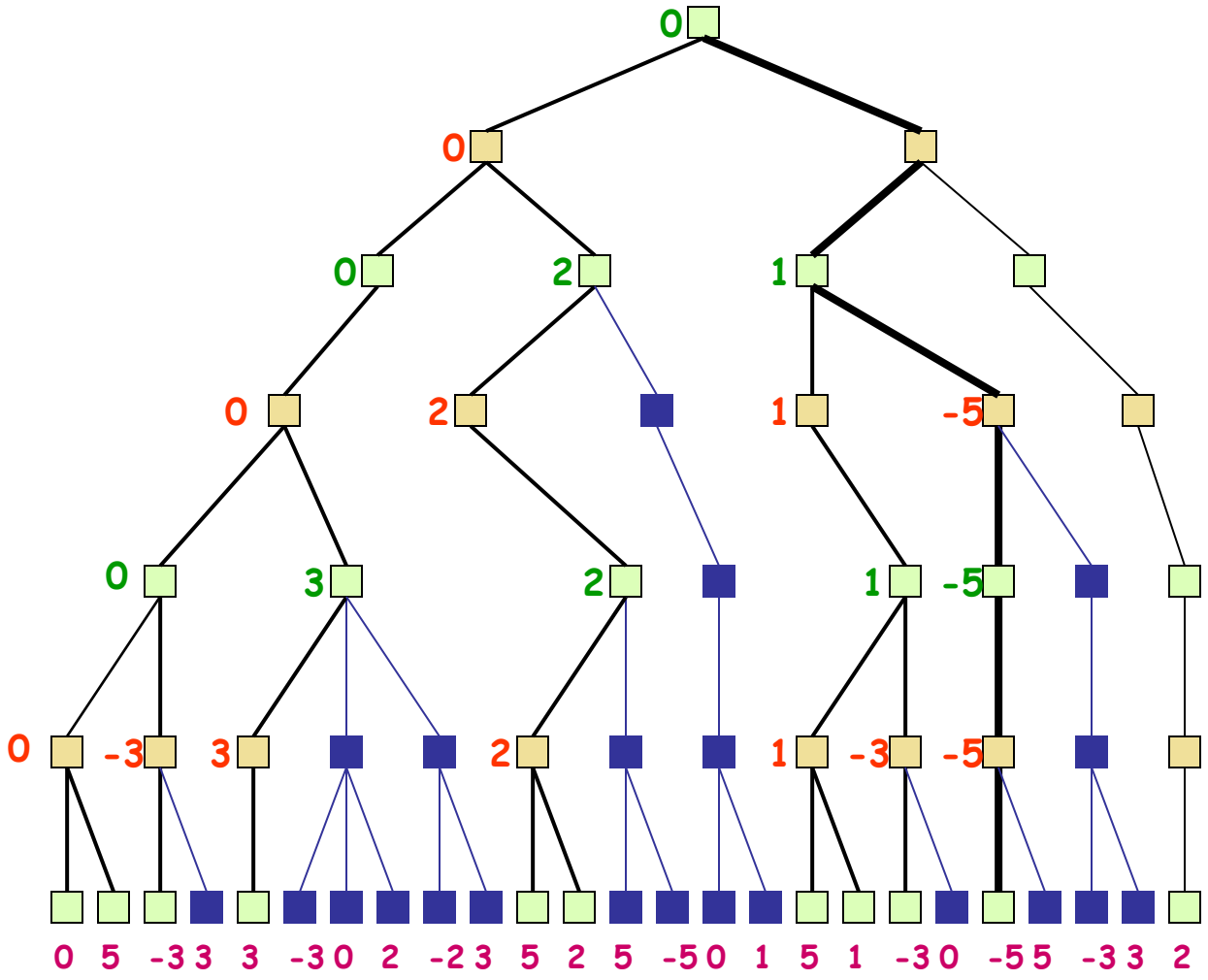
Example



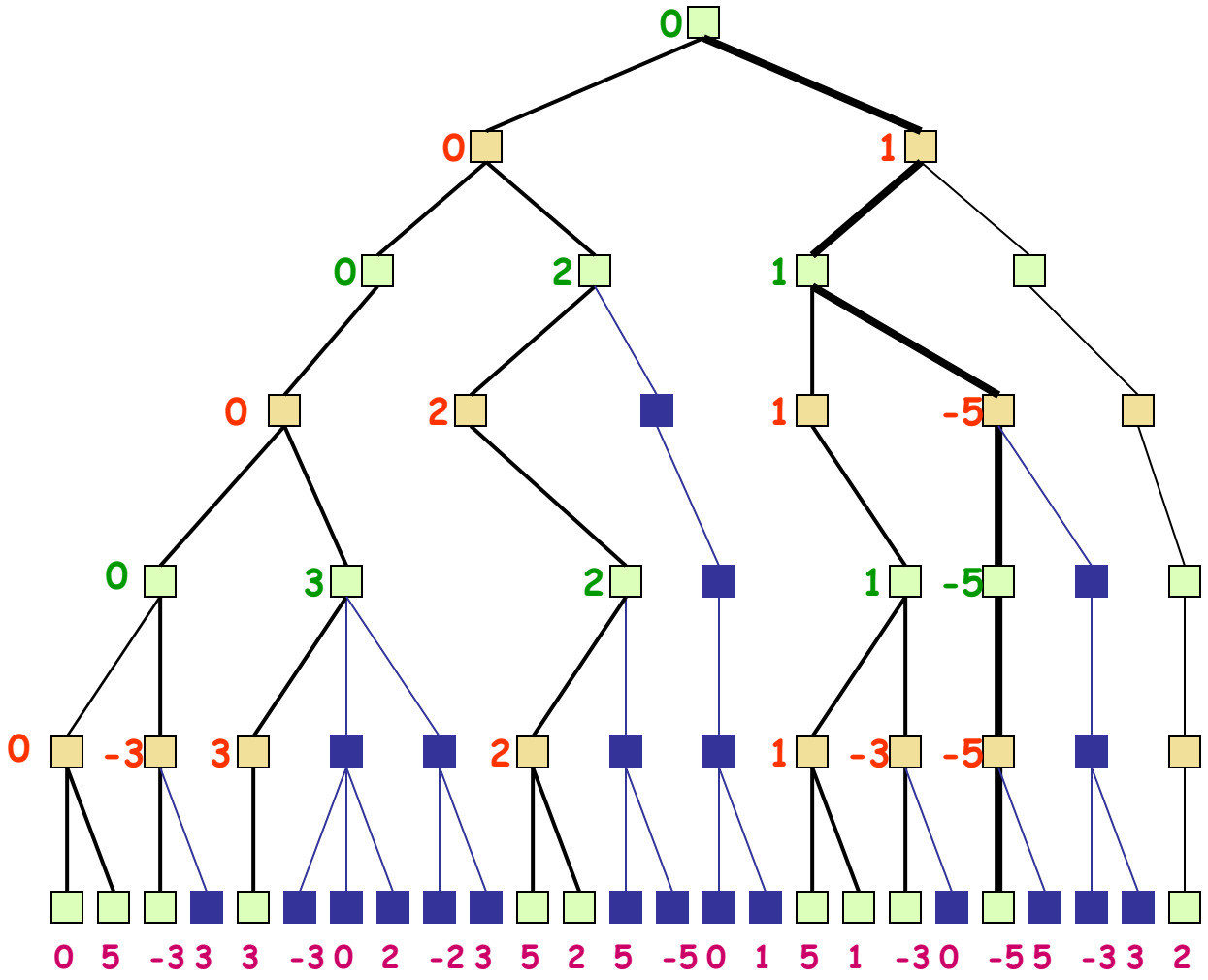
Example



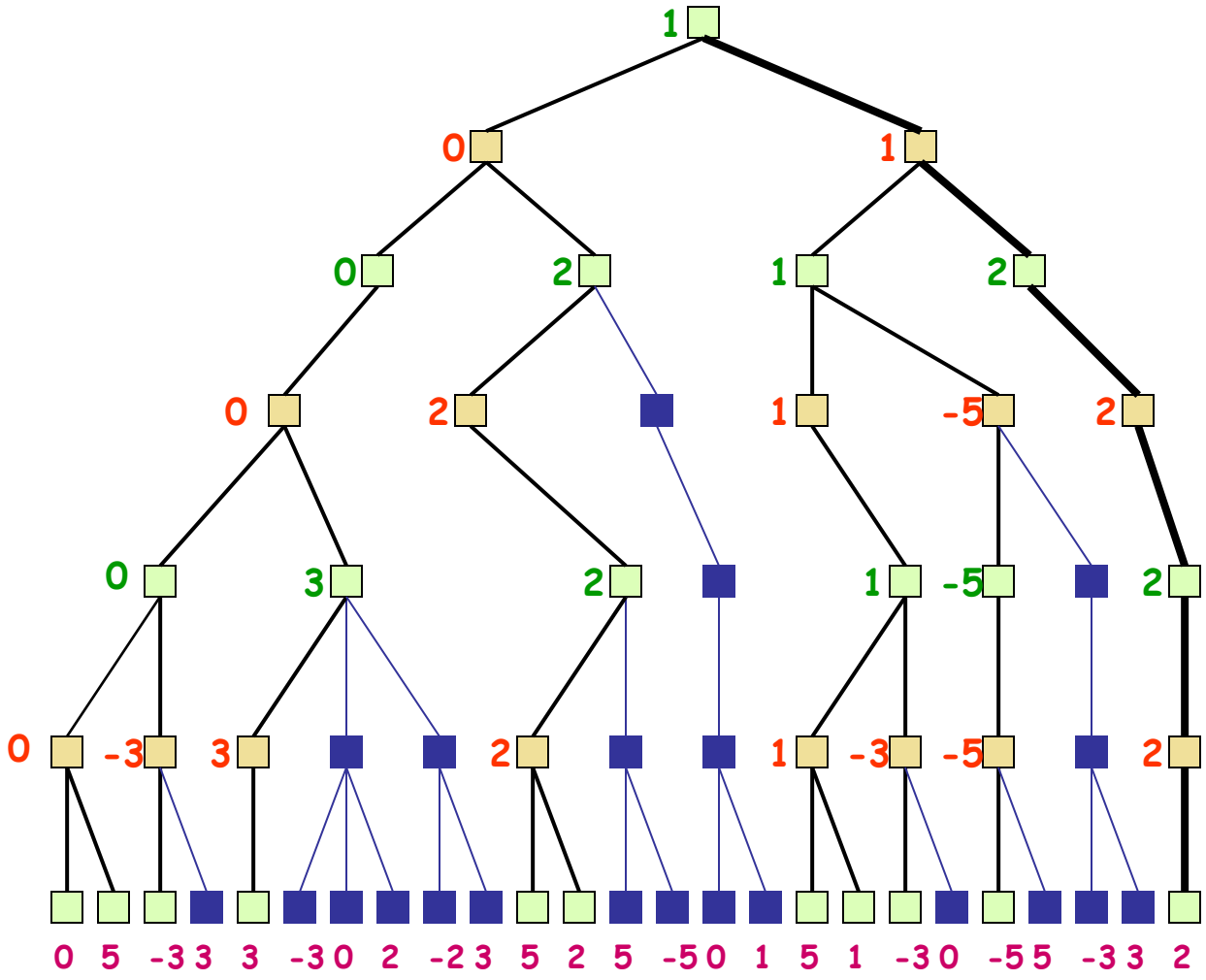
Example



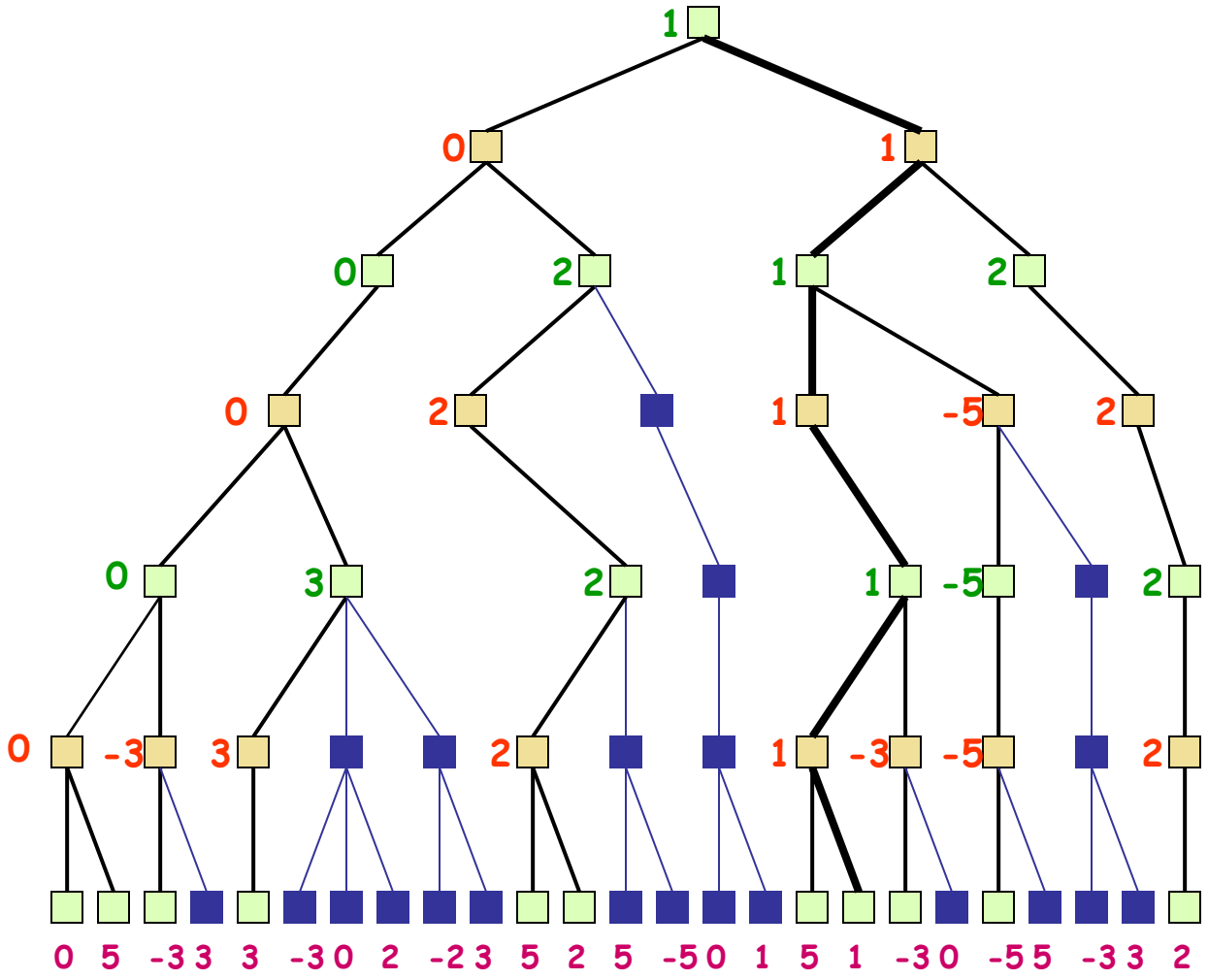
Example



Example

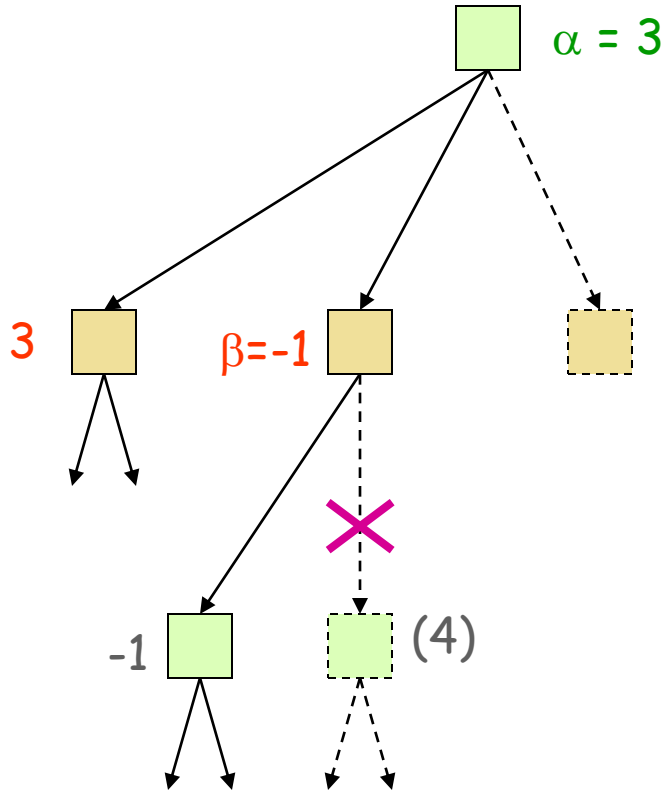

1
0 1
0 2 1 2
0 2 1 -5 2
0 3 2 1 -5 2
0 -3 3 2 1 -3 -5 2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

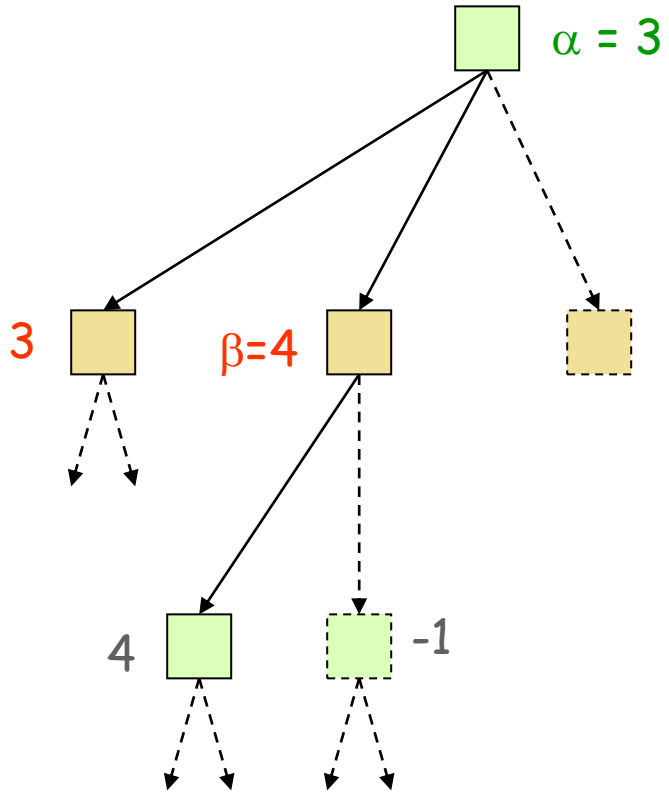
Example


1
0 1
0 2 1 2
0 2 1 -5 2
0 3 2 1 -5 2
0 -3 3 2 1 -3 -5 2
0 5 -3 3 3 -3 0 2 -2 3 5 2 5 -5 0 1 5 1 -3 0 -5 5 -3 3 2

How much do we gain?

Consider these two cases:





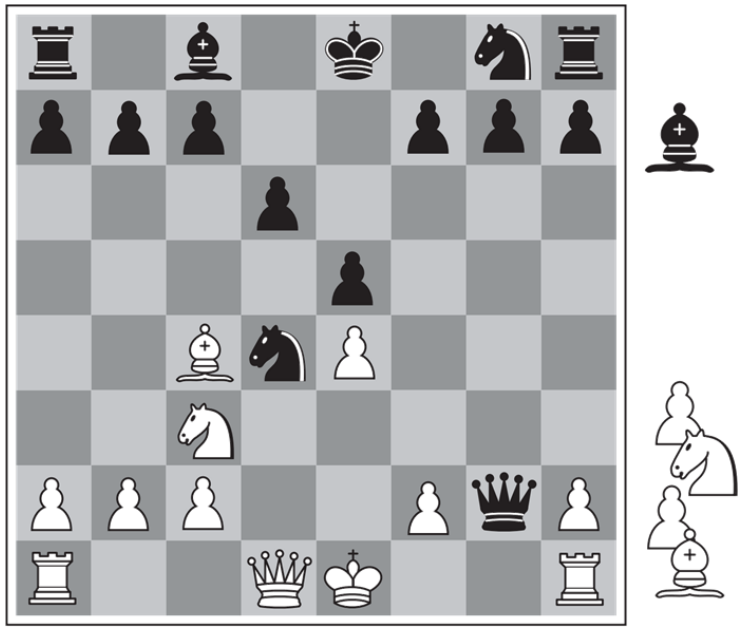
How much do we gain?

- Assume a game tree of uniform branching factor b
- Minimax examines $O(b^h)$ nodes, so does alpha-beta in the worst-case
- The gain for alpha-beta is maximum when:
  - The MIN children of a MAX node are ordered in decreasing backed up values
  - The MAX children of a MIN node are ordered in increasing backed up values
- Then alpha-beta examines $O(b^{h/2})$ nodes [Knuth and Moore, 1975]
- But this requires an oracle (if we knew how to order nodes perfectly, we would not need to search the game tree)
- If nodes are ordered at random, then the average number of nodes examined by alpha-beta is $\sim O(b^{3h/4})$

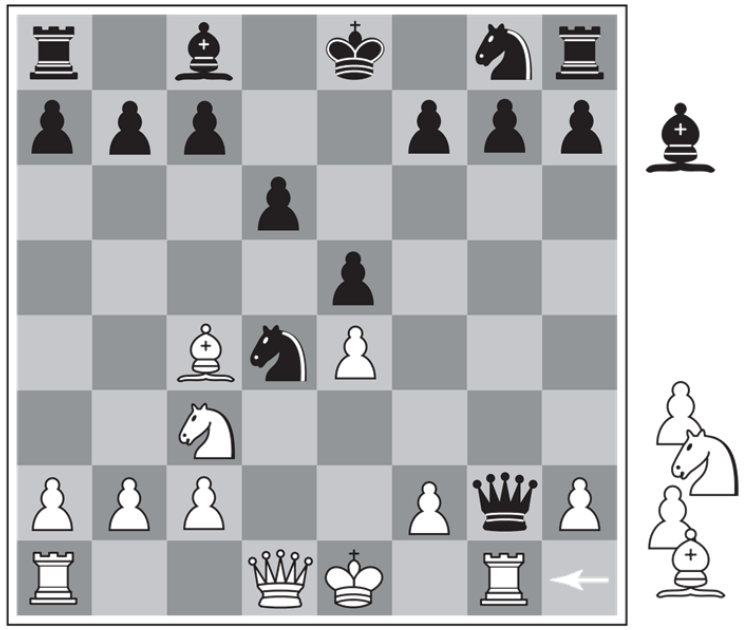
Cutting off search: simple depth limit

if CUTOFF-TEST($state$, $depth$) **then return** EVAL($state$)

- Problem1: non-quiescent positions
  - Few more plies make big difference in evaluation value



(a) White to move



(b) White to move

Cutting off search: simple depth limit

- Problem 2: horizon effect
  - Delaying tactics against opponent's move that causes serious unavoidable damage (because of pushing the damage beyond the horizon that the player can see)