Adversarial Search: Game-playing

UMaine COS 470/570 – Introduction to AI

Spring 2019

Adversarial search

Non-solipsistic search

- So far: search involves only one agent
- Many times ≥ 1:
  - Game playing
  - NLP dialogue systems
  - AI assistants
  - Multiagent systems
- Also: maybe think of "the world" as another agent
  - Your agent takes action x...
  - ...world "takes action" y
  - Why? Spontaneous events/processes, uncertainty, incomplete knowledge...

Implications

- Can't plan, then act – unpredictable world/agents $\Rightarrow$ errors
- Shouldn't plan too far ahead
- May not be able to predict every possibility
  - Uncertainty
  - Maybe too many possibilities
- Other agent(s):
  - *May* be cooperative...
  - ...or they may be out to get you!
  - Pessimistic: model real world this way
- $\Rightarrow$ *adversarial search*
- This section: game playing, some game theory
- Later: MAS, real-world planning/acting

Game playing

Game playing

- Here: Turn-taking games, not FPS, etc.
- Similar to search:
  - Initial state, operators known
  - Goal well-defined, if intensional
  - Heuristics often possible for distance to goal
- Differences:
  - Adversarial search
  - Can't take moves back (generally)
  - Can't usually plan/search to goal

Game playing

- Here: Turn-taking games, not FPS, etc.
- Similar to search:
  - Initial state, operators known
  - Goal well-defined, if intensional
  - Heuristics often possible for distance to goal
- Differences:
  - Adversarial search
  - Can't take moves back (generally)
  - Can't usually plan/search to goal – e.g.,
    - ~10^{40} nodes in chess' search space (Nilsson)

Game playing

- Here: Turn-taking games, not FPS, etc.
- Similar to search:
  - Initial state, operators known
  - Goal well-defined, if intensional
  - Heuristics often possible for distance to goal
- Differences:
  - Adversarial search
  - Can't take moves back (generally)
  - Can't usually plan/search to goal – e.g.,
    - ~10^{40} nodes in chess' search space (Nilsson)
    - 1 expansion/ns $\Rightarrow$ 10^{22} centuries (universe: 10^8 centuries old)

General approach

- Current state: your move
- Apply all possible operators $\Rightarrow$ child states
- Child state: opponent's move
- Apply all opponent's operators
- Continue to some *depth cut-off*
- At cut-off: apply heuristic function to leaves
- Pick initial move to maximize worth of move

Game assumptions

- Agent wants to maximize value
- Opponent wants to maximize board's value to *itself*
  - Assume *zero-sum game*
  - ∴ opponent wants to *minimize* worth of board
- Every turn: player's options based only on current board

Static evaluation function

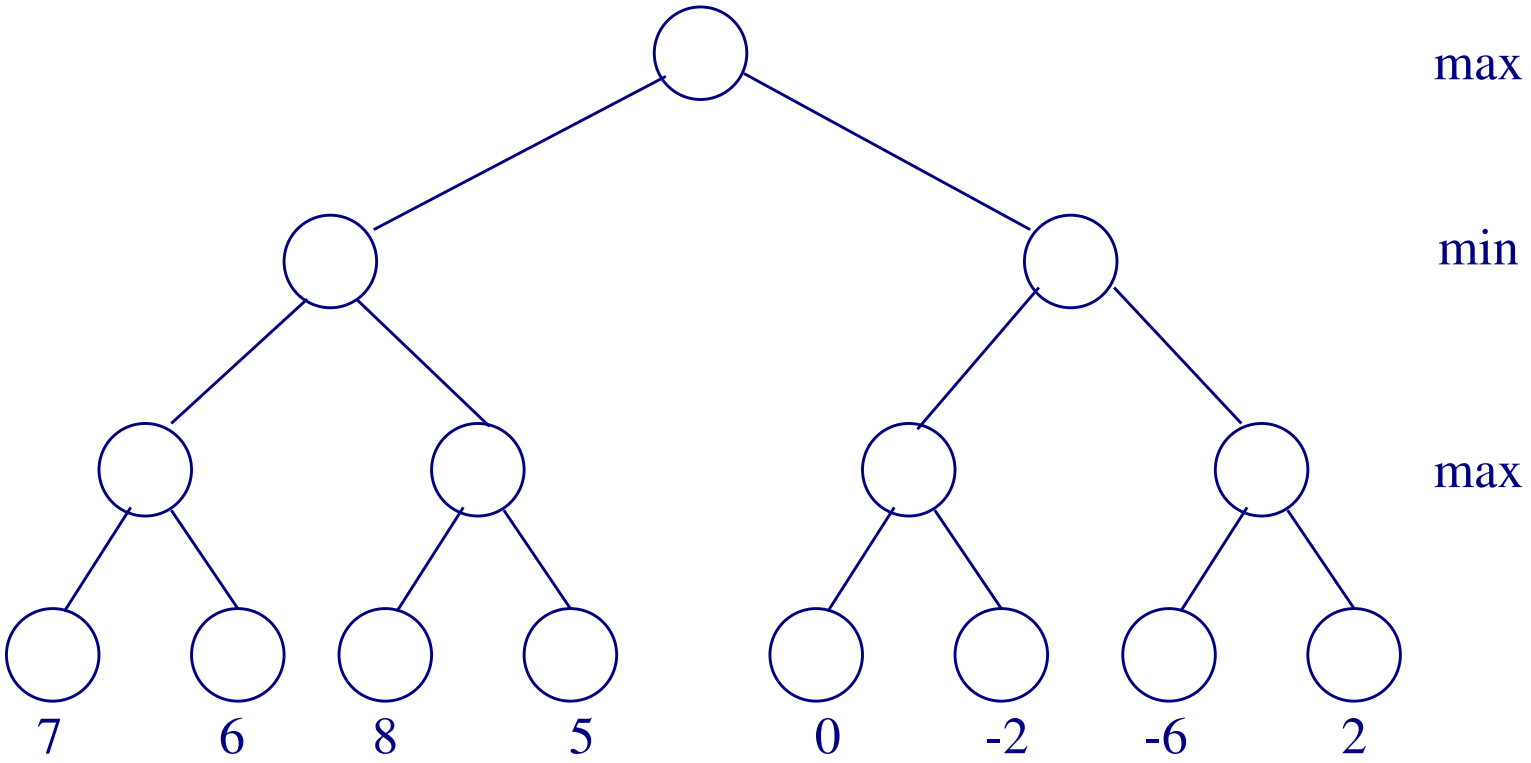
- Heuristic for games = *static evaluation function*
  - Win: $+\infty$, loss $-\infty$
  - Various heuristics for other states
- Don't care about cost to get there (usually)
- Player: want $\uparrow$SE, opponent $\downarrow$SE

Minimax Search

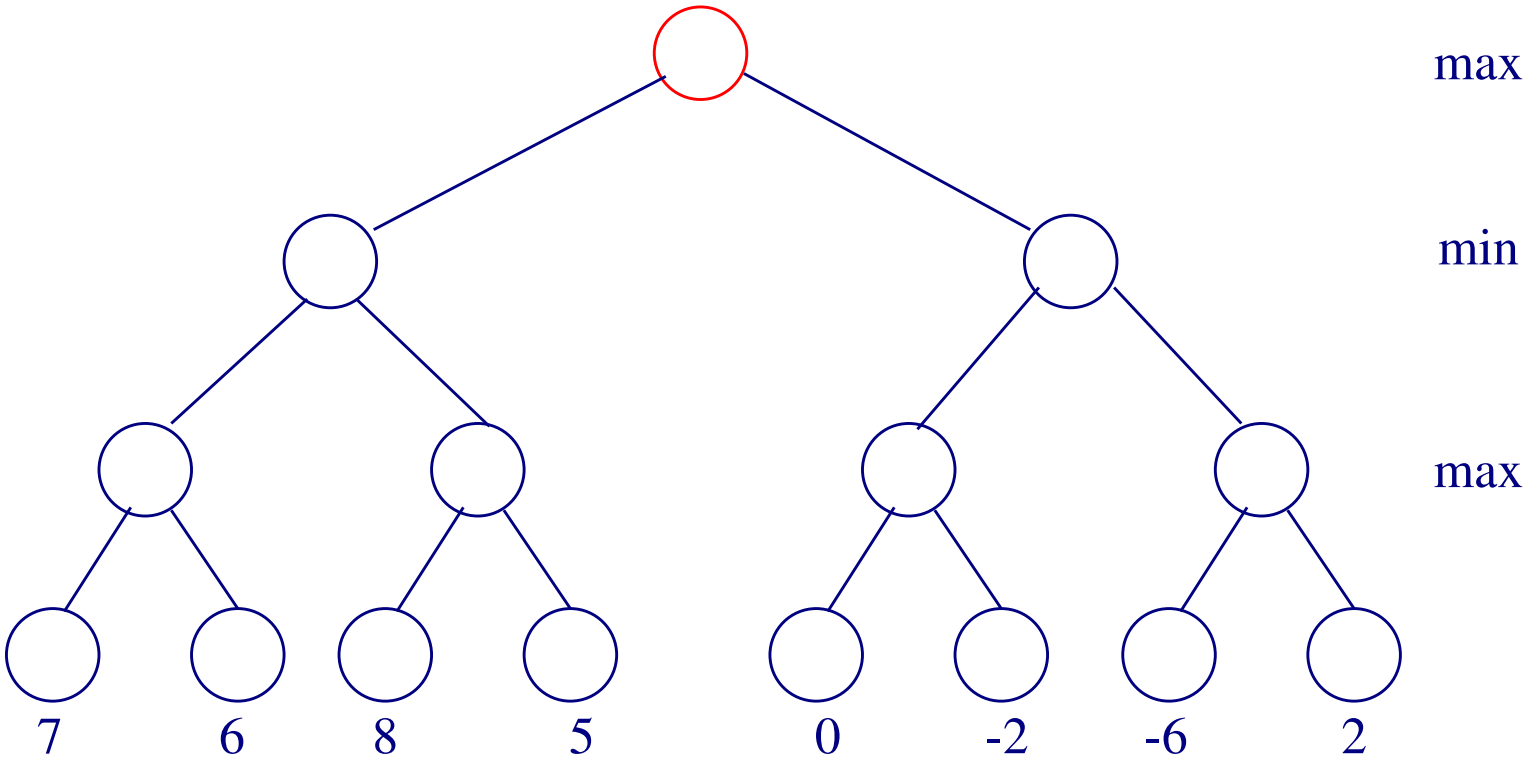
Minimax search procedure

- Apply all operators to generate tree down to some level
- Use static eval function for each leaf
- Propagate values up the tree
  - Select min/max depending on level
- Select best move at top node
- Use depth-first search:
  - Game trees can be huge: don't want to store nodes
  - Board doesn't depend on other boards $\Rightarrow$ don't need others in memory

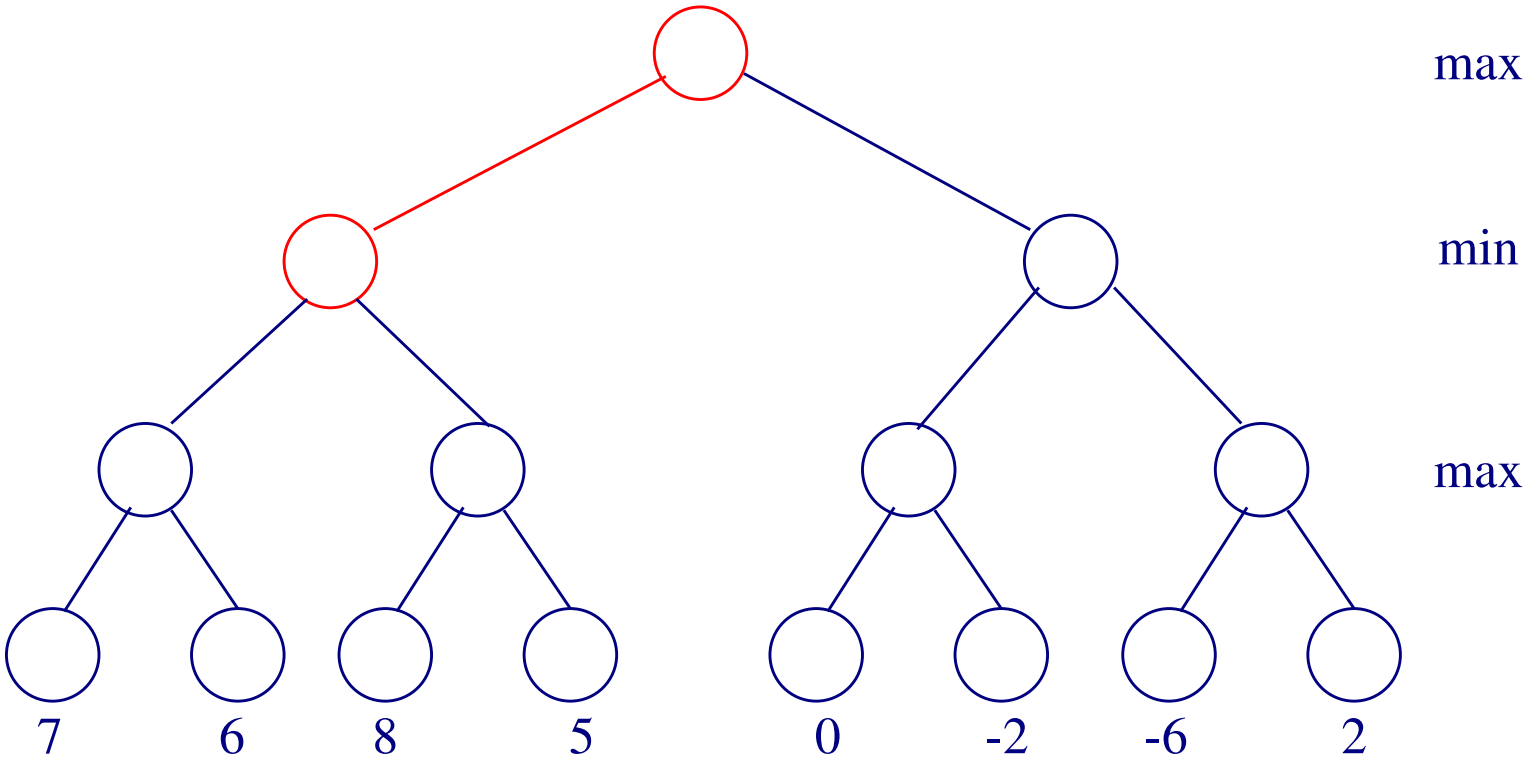
Example 1



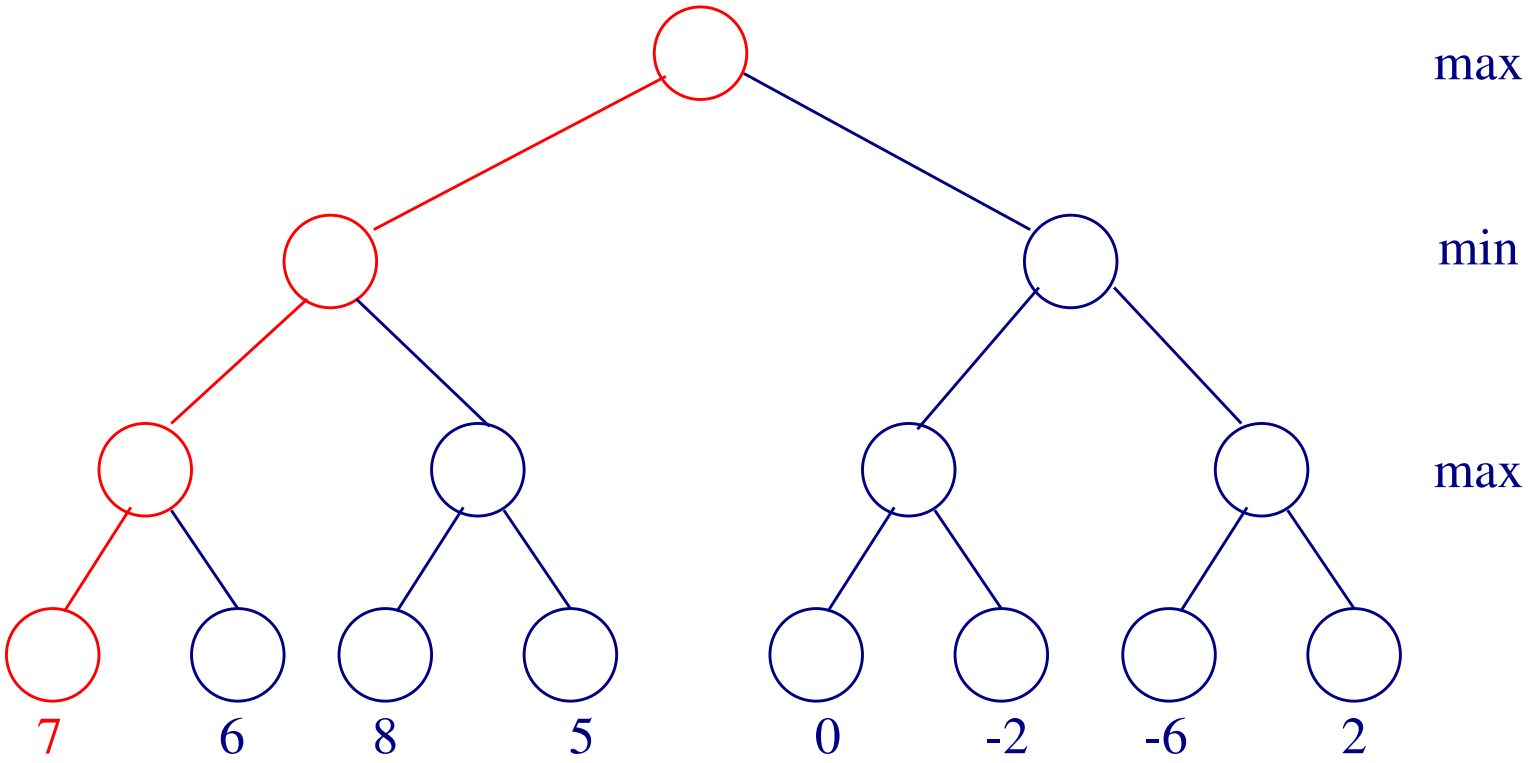
Example 1



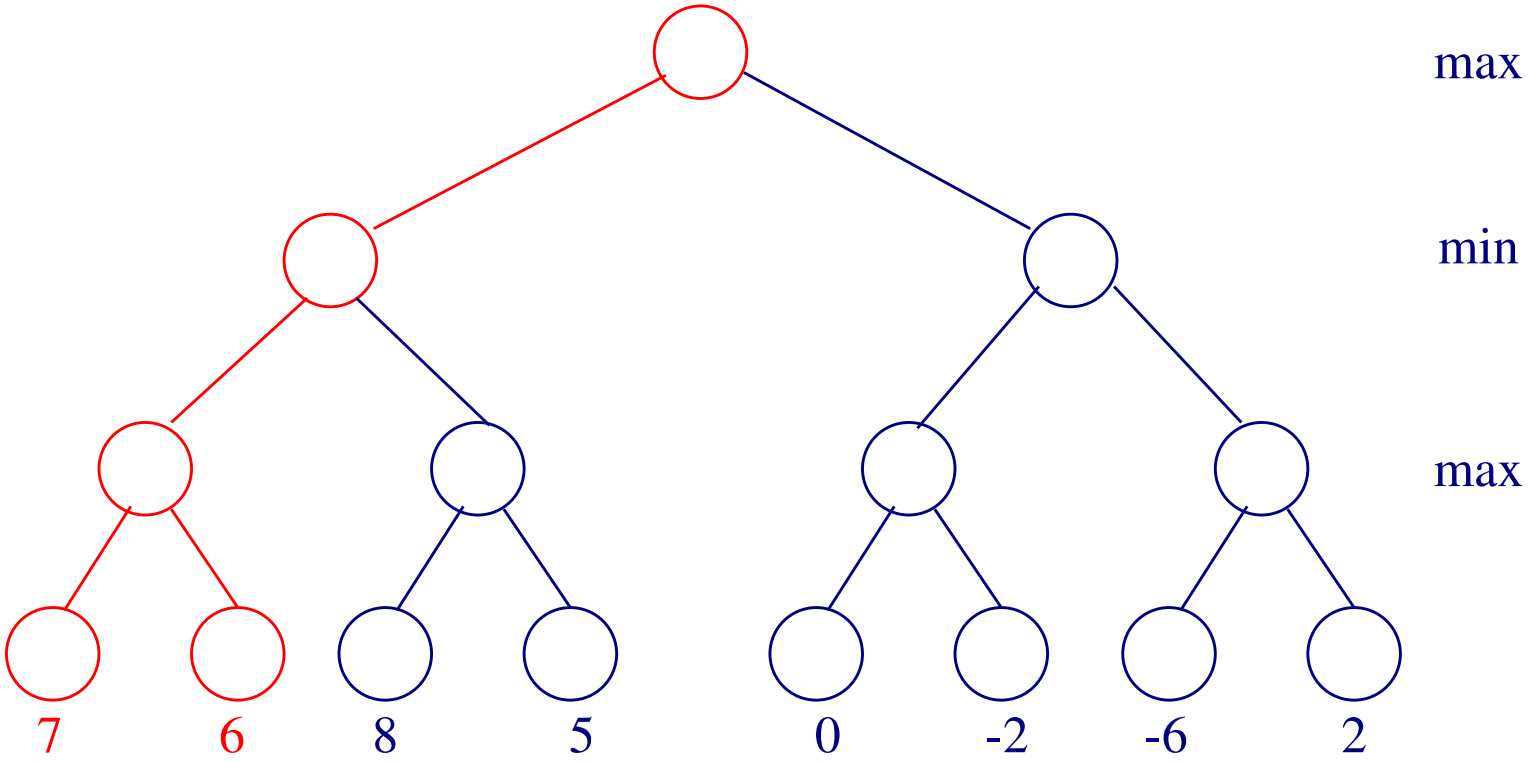
Example 1



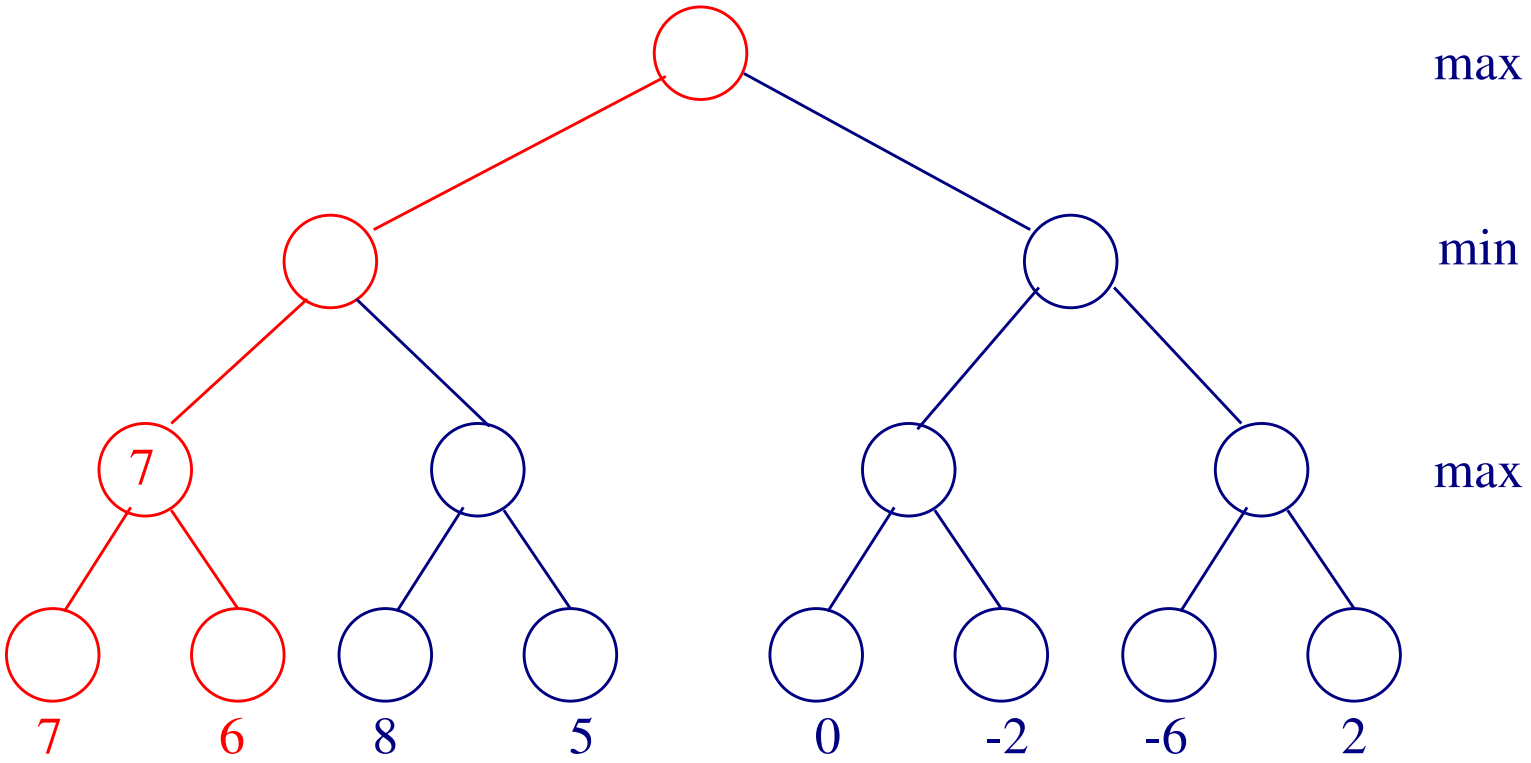
Example 1



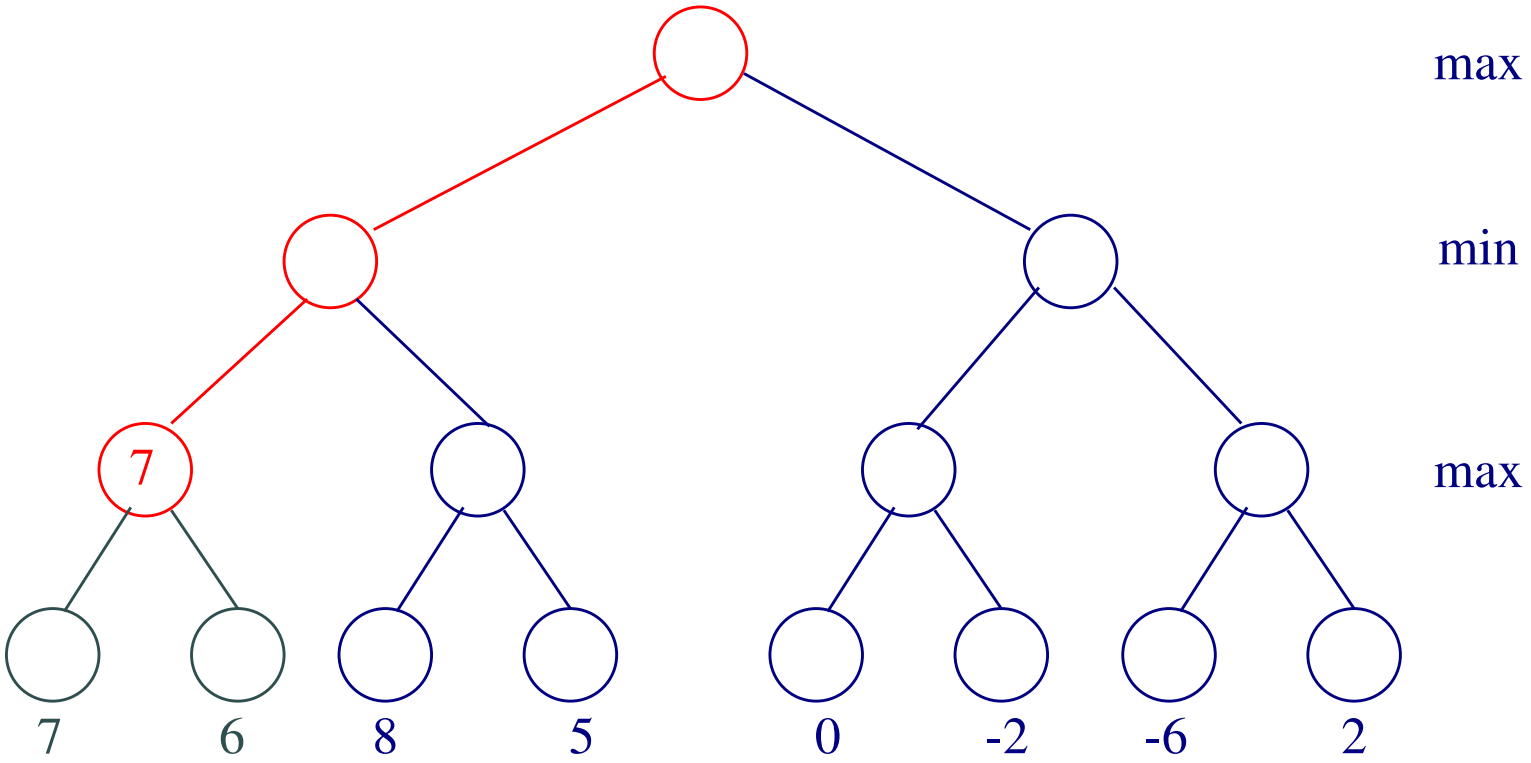
Example 1



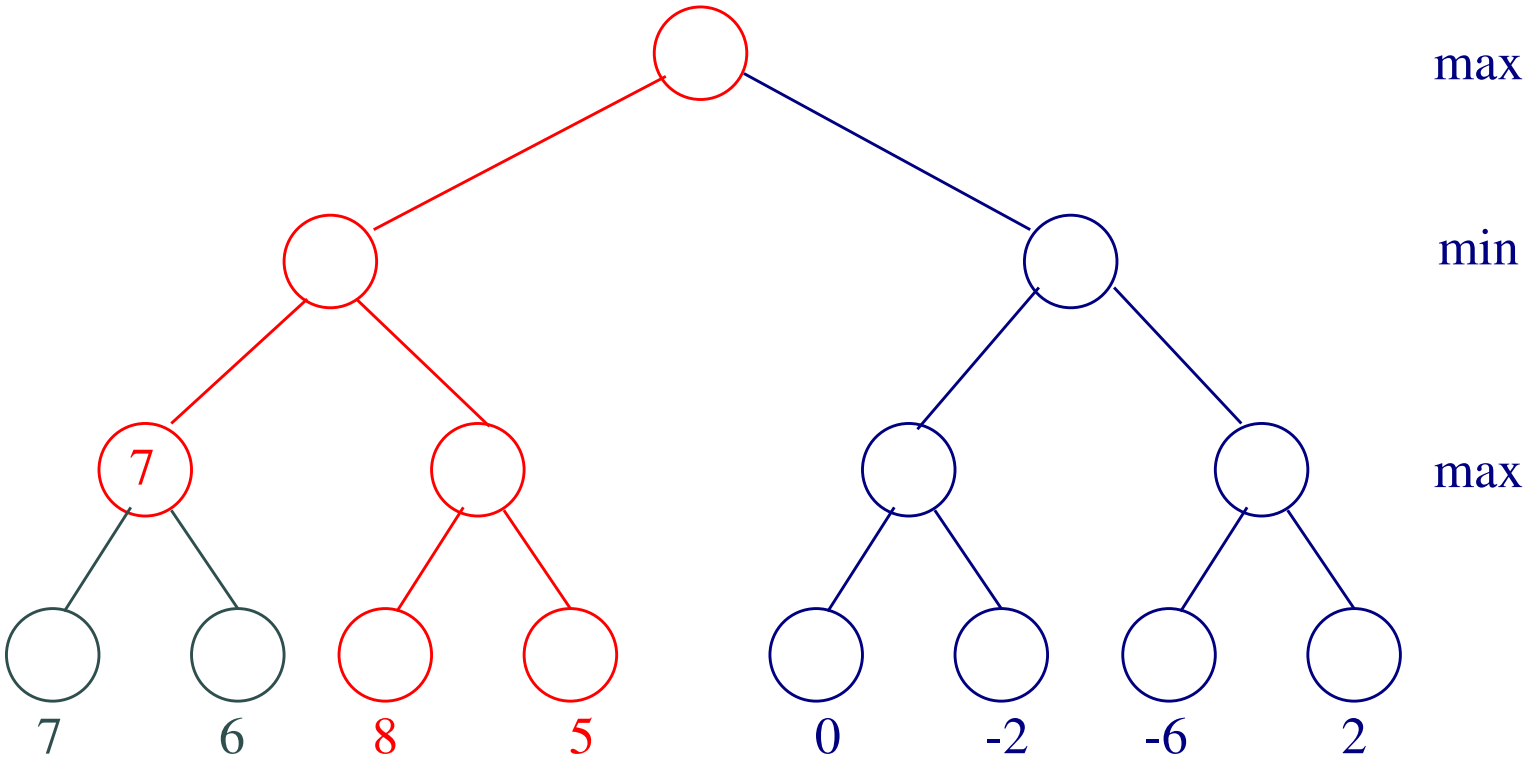
Example 1



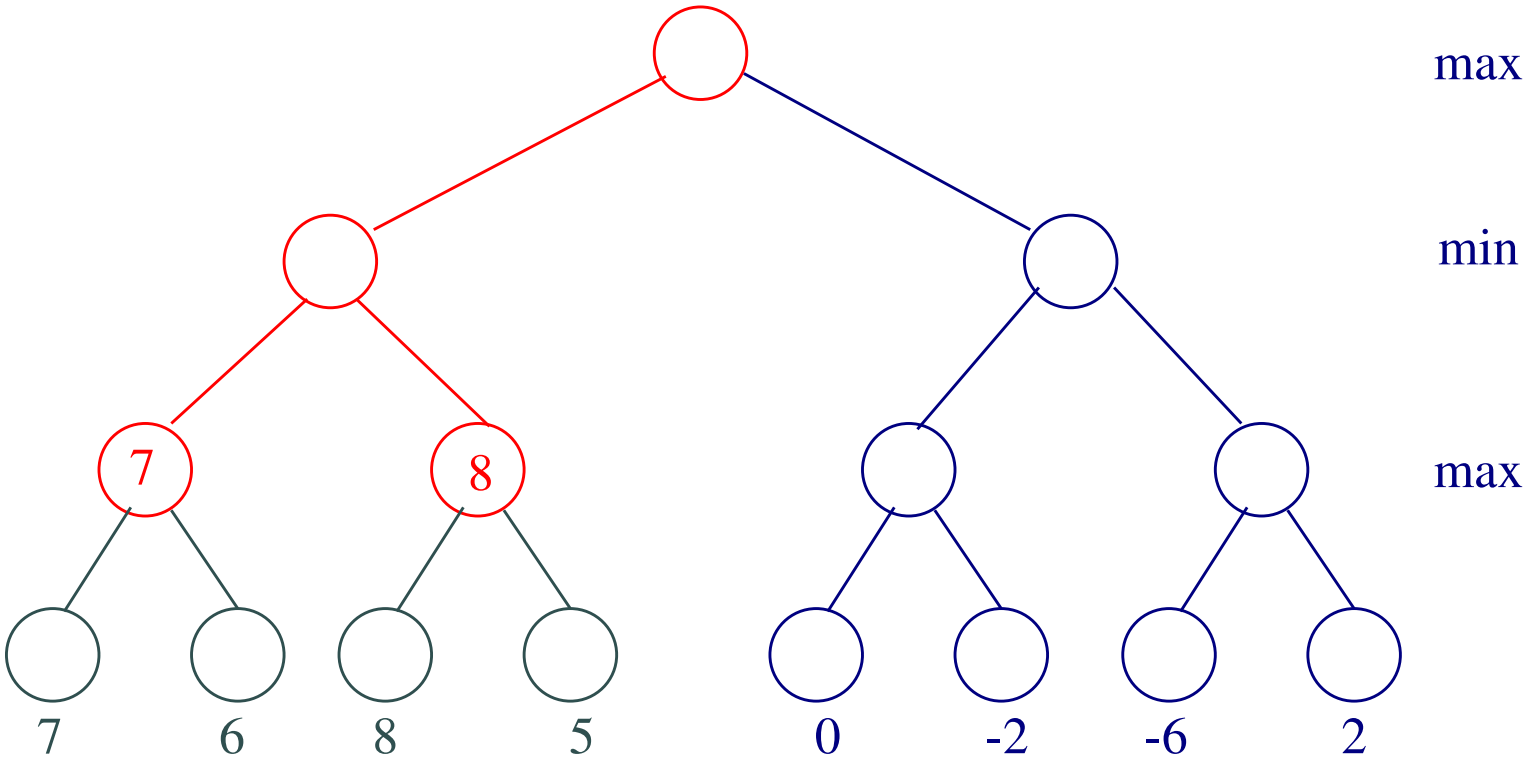
Example 1



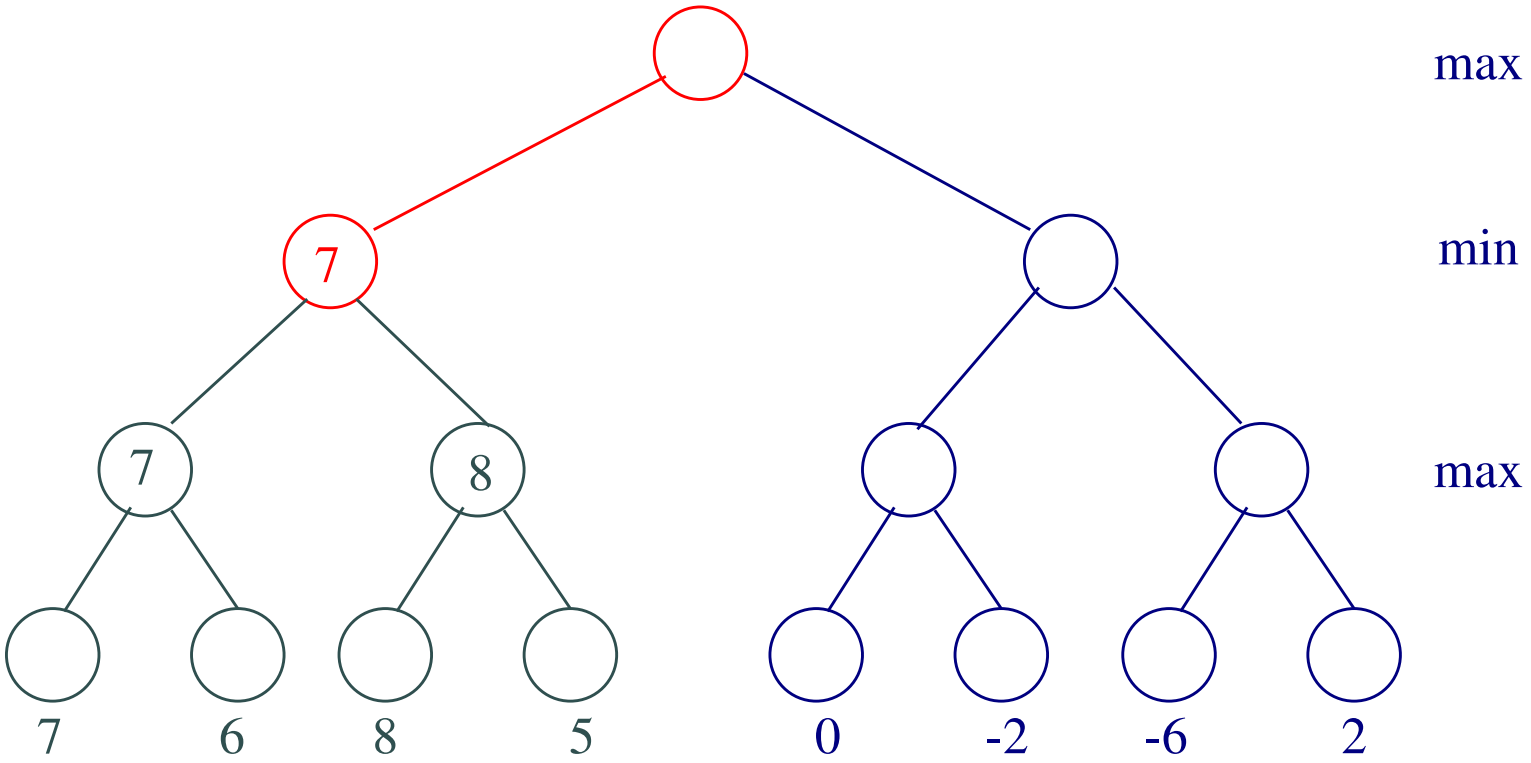
Example 1



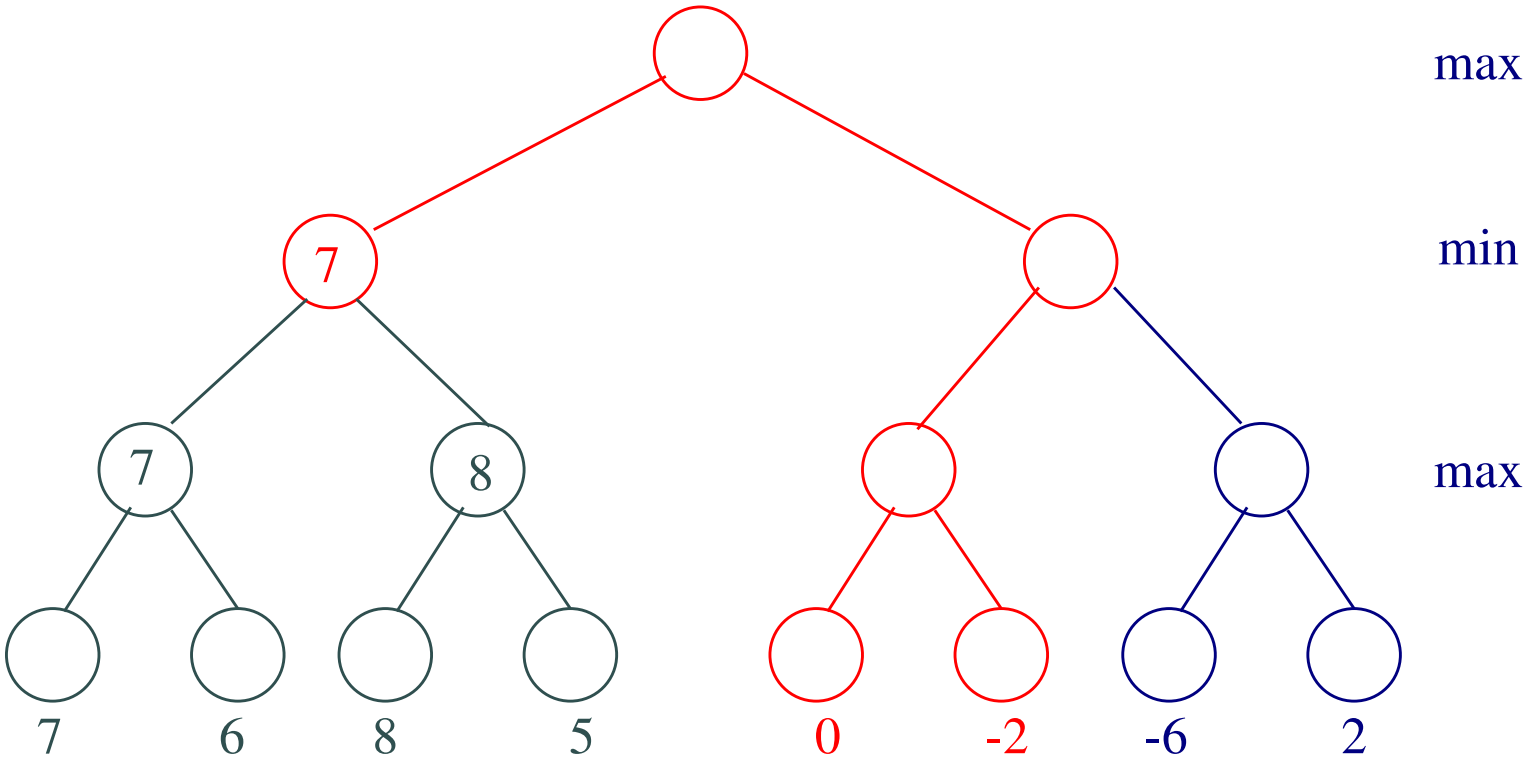
Example 1



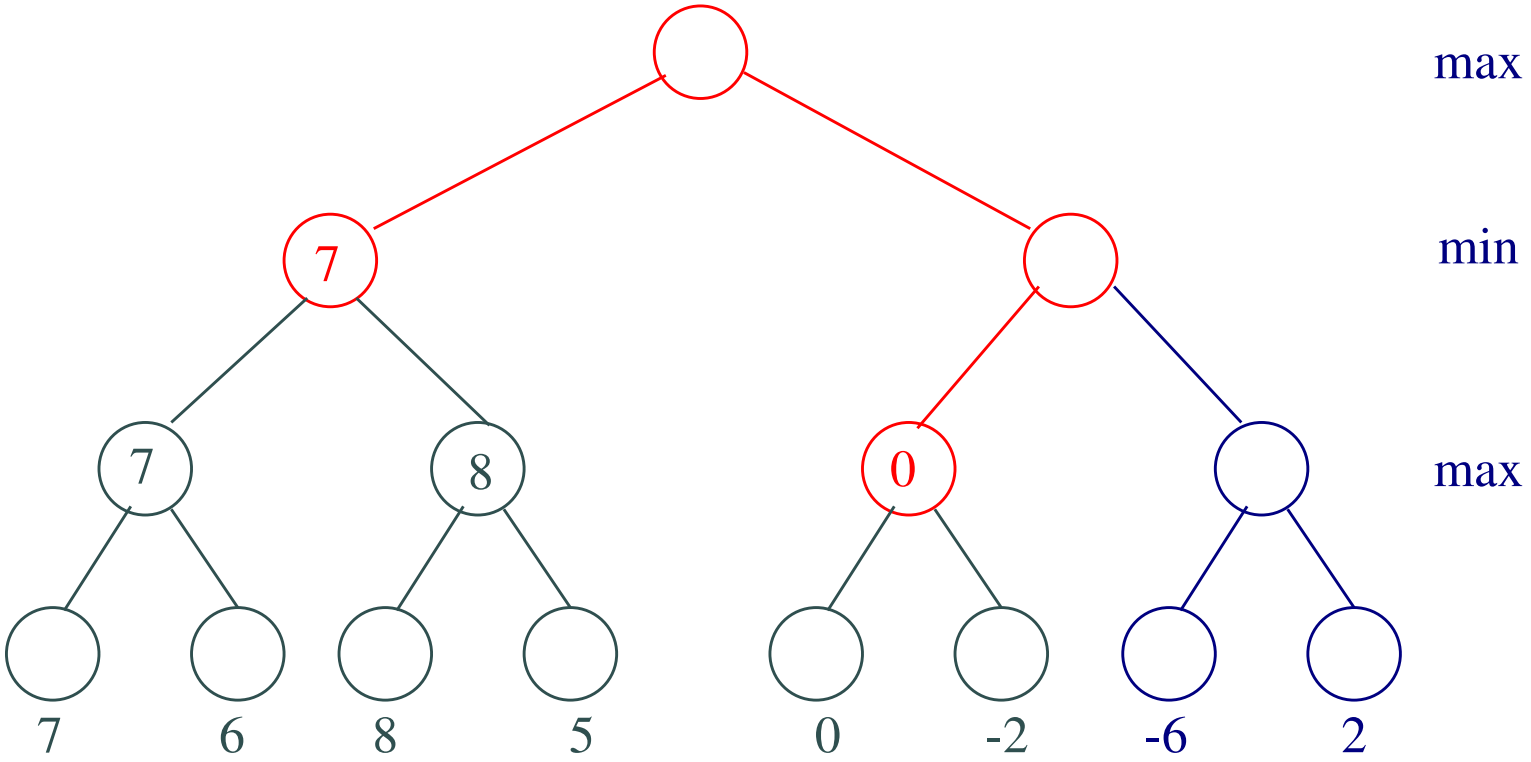
Example 1



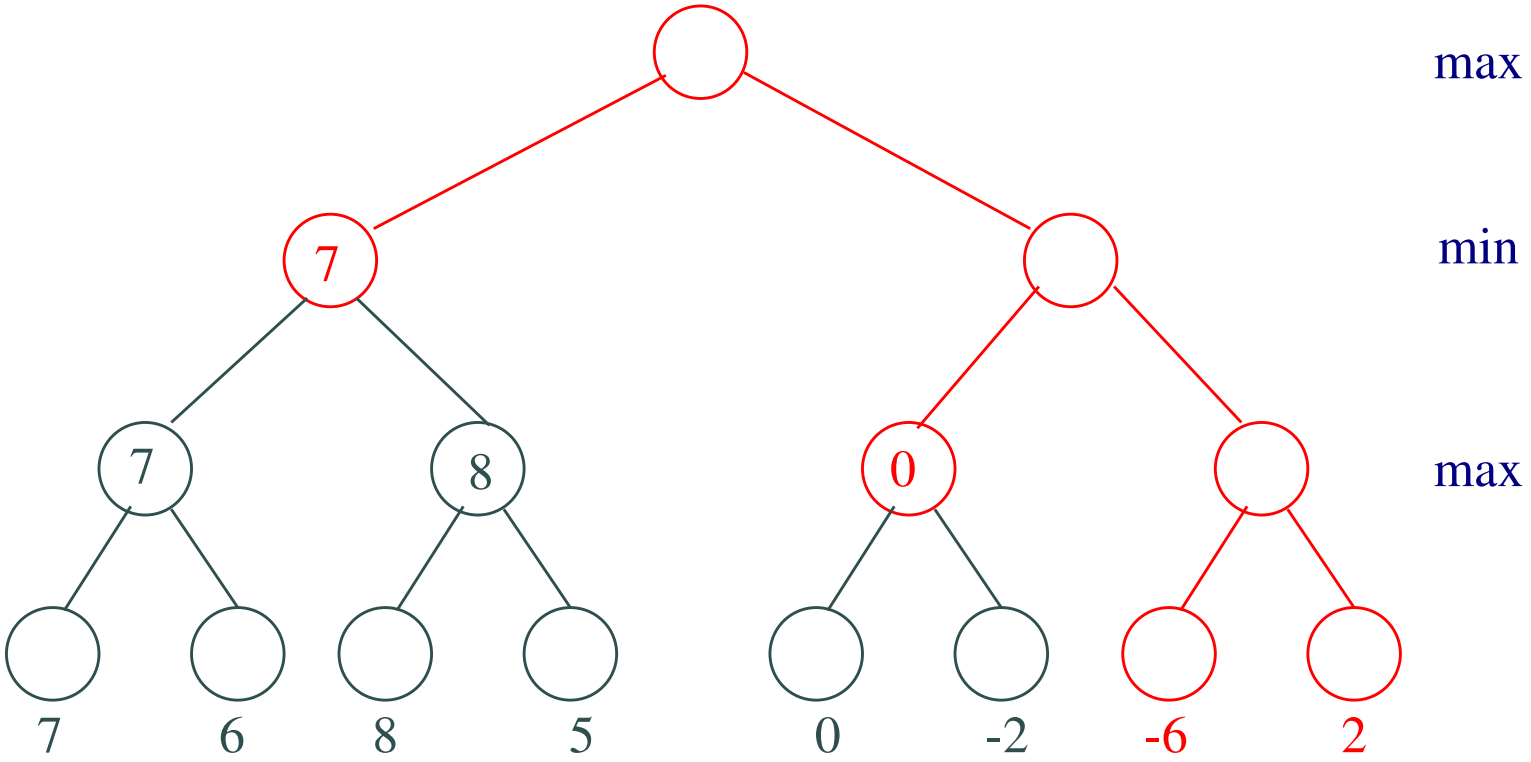
Example 1



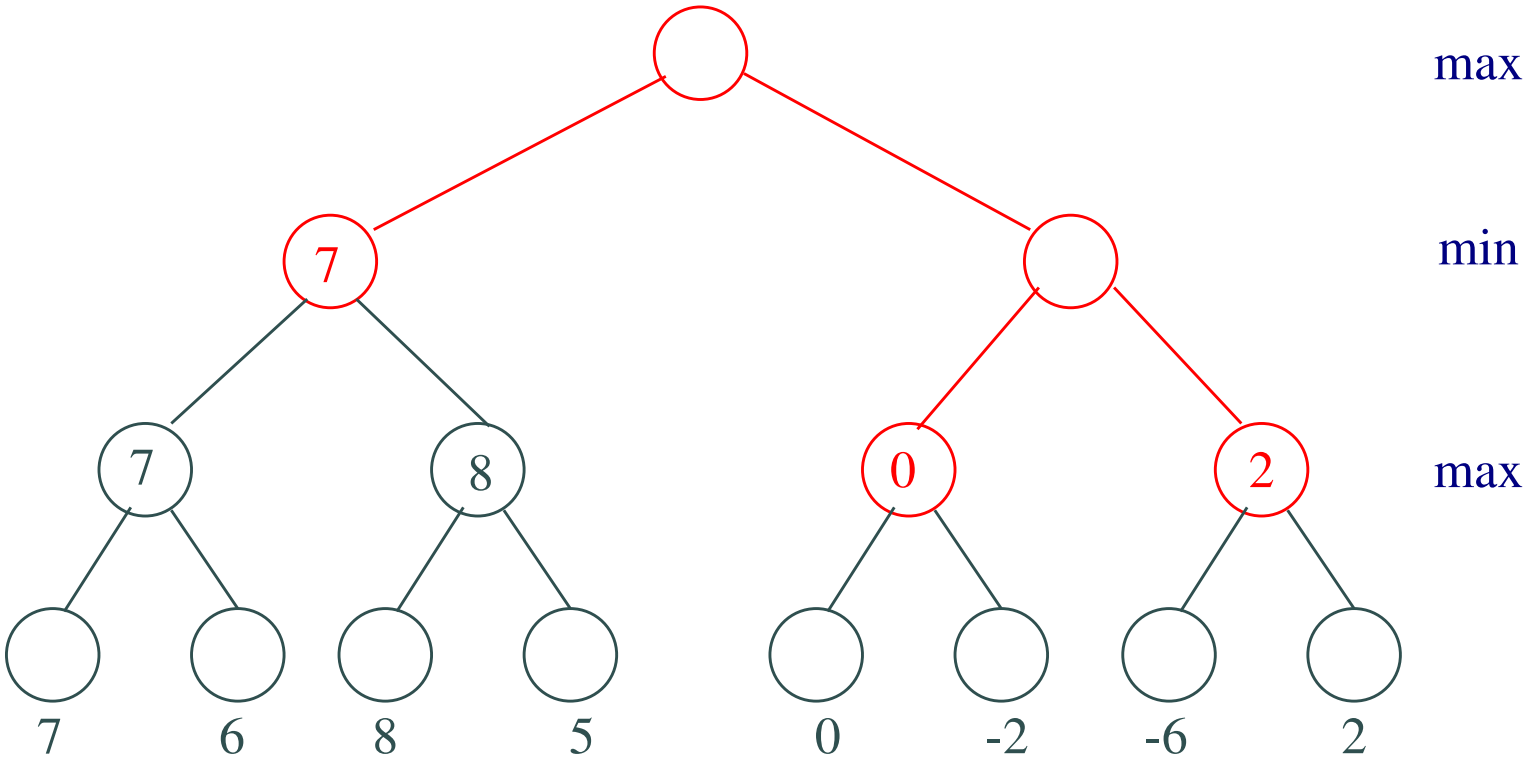
Example 1



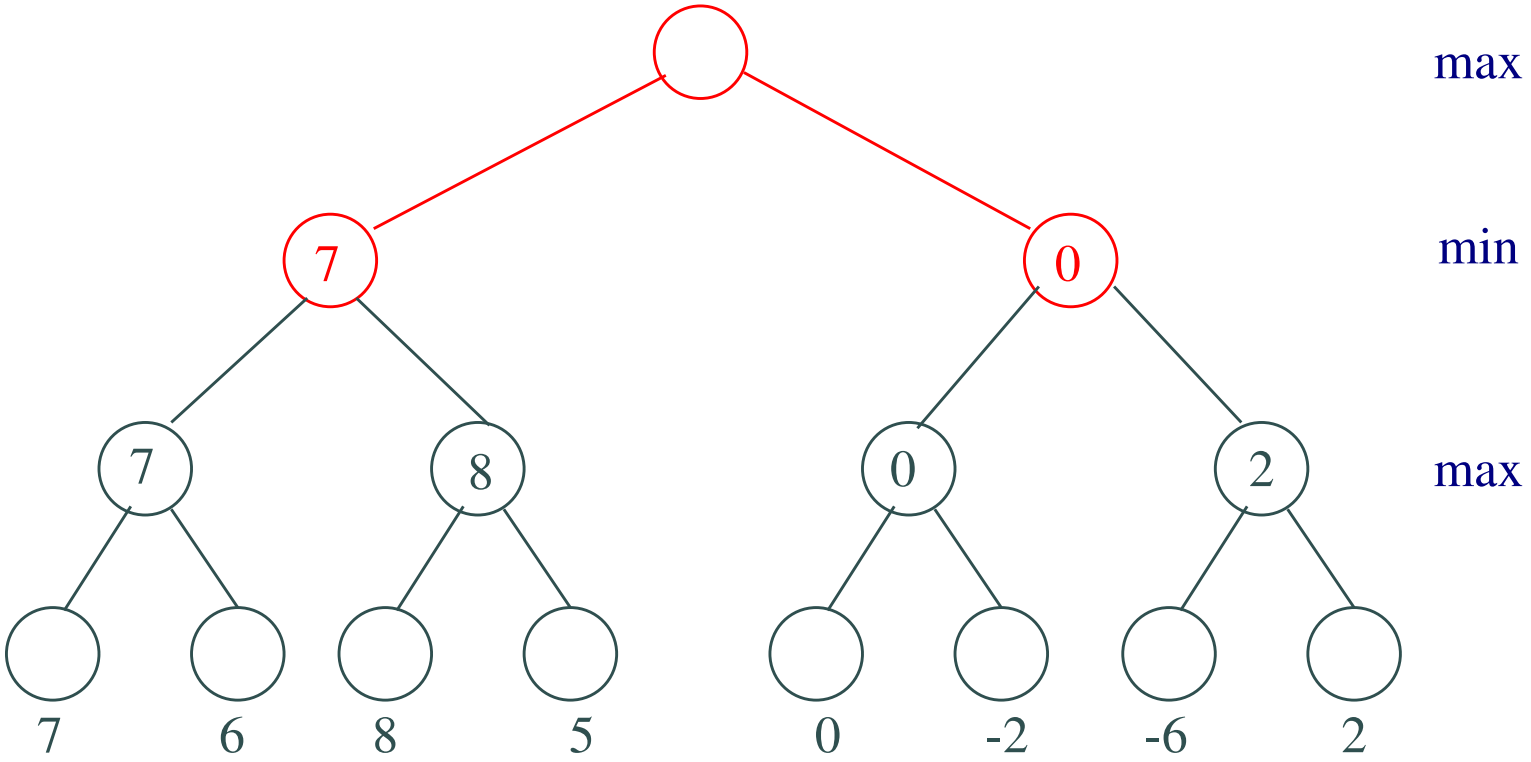
Example 1



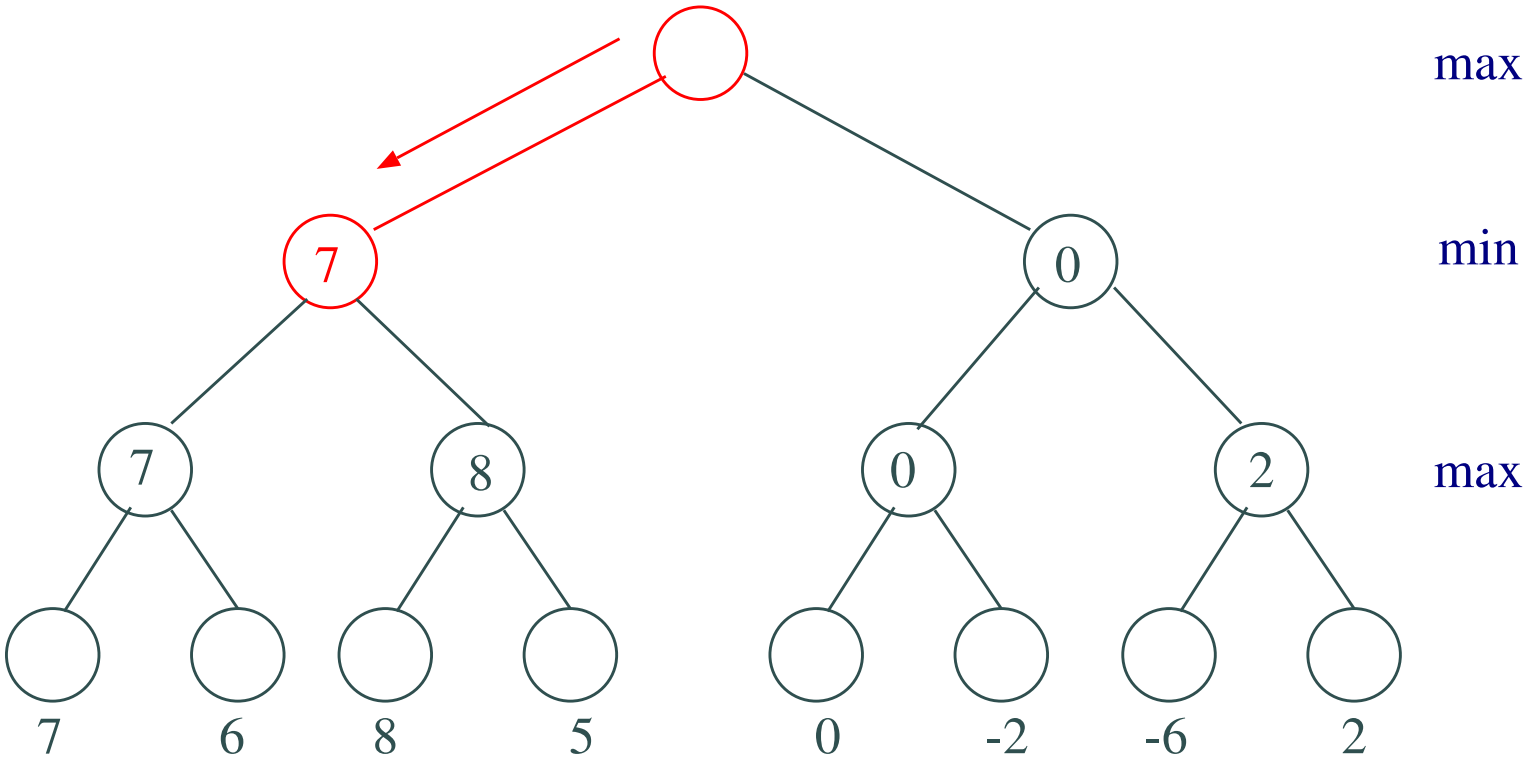
Example 1



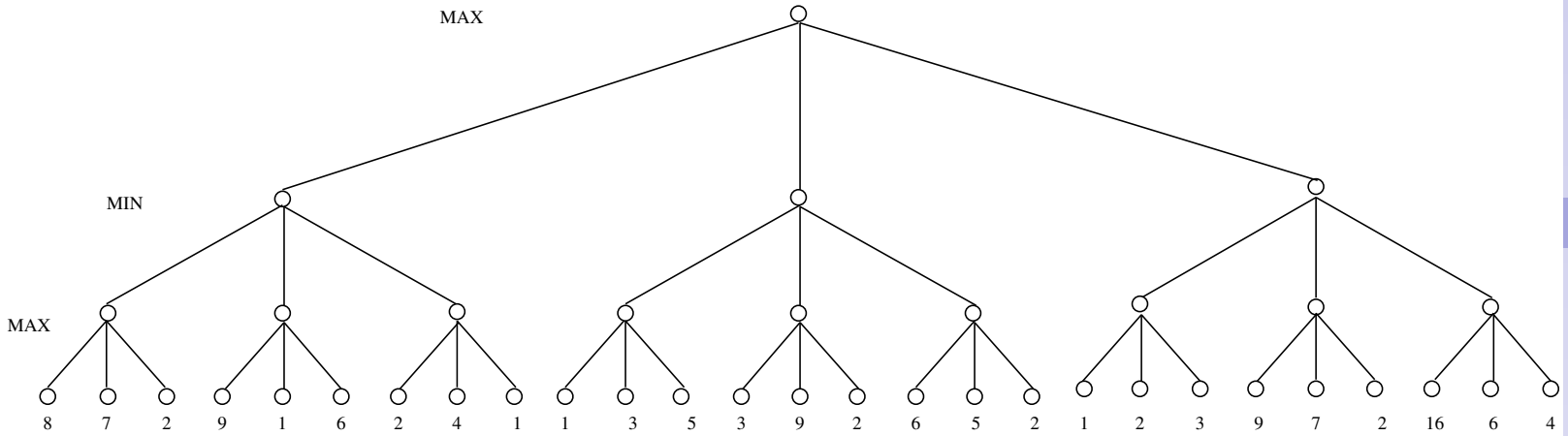
Example 1



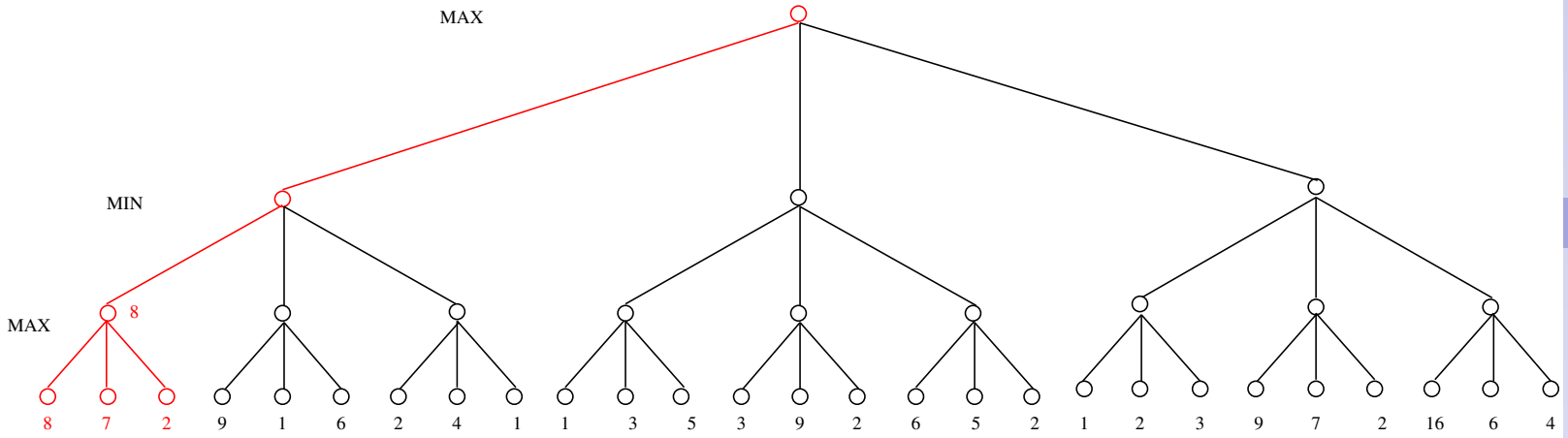
Example 1



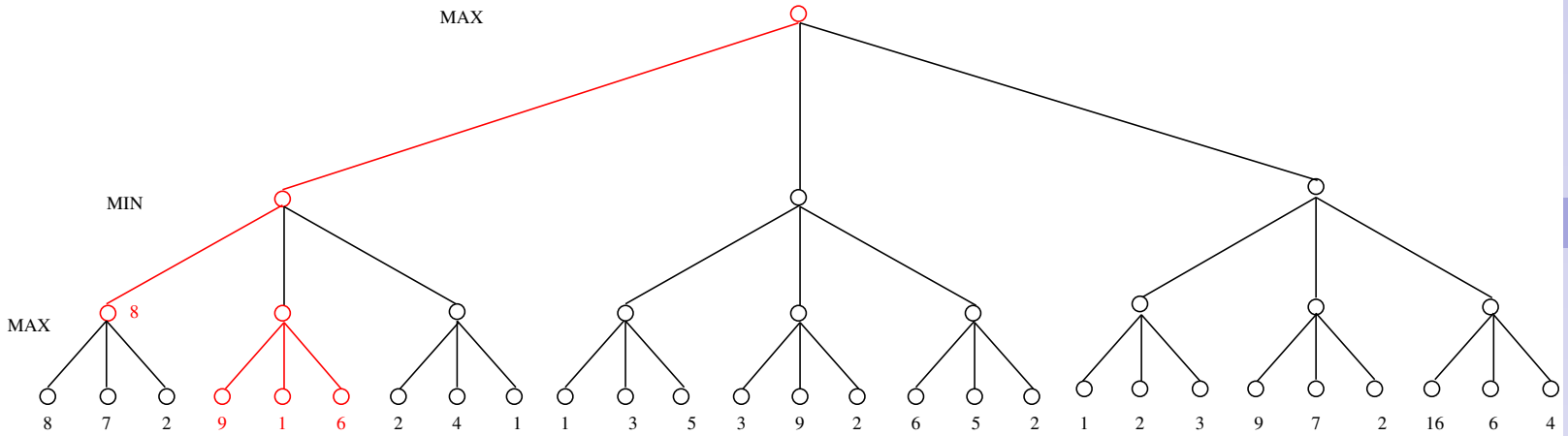
Example 2



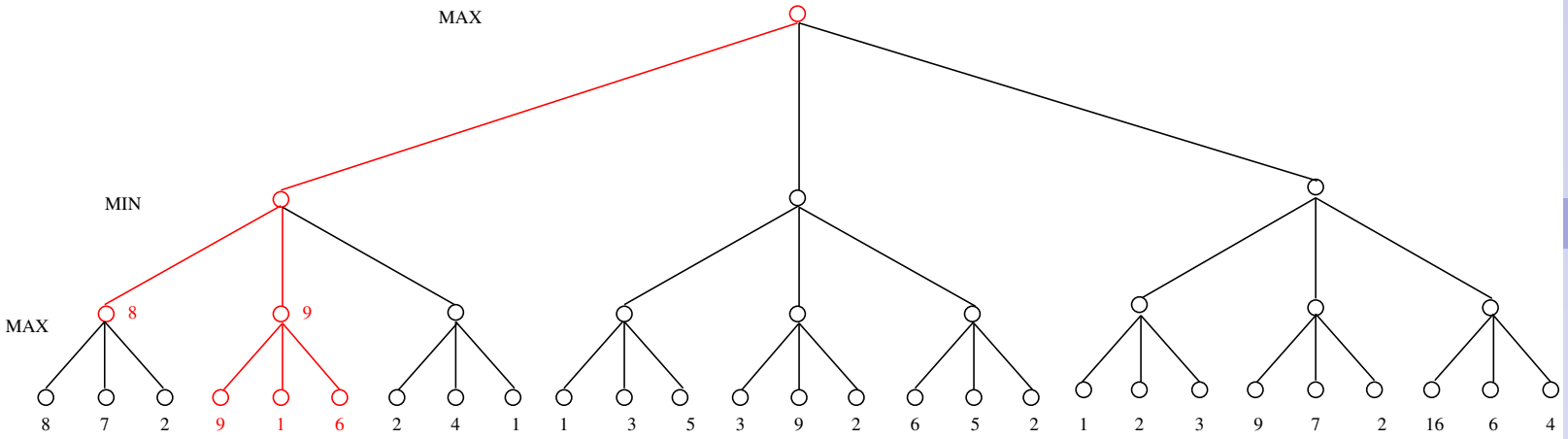
Example 2



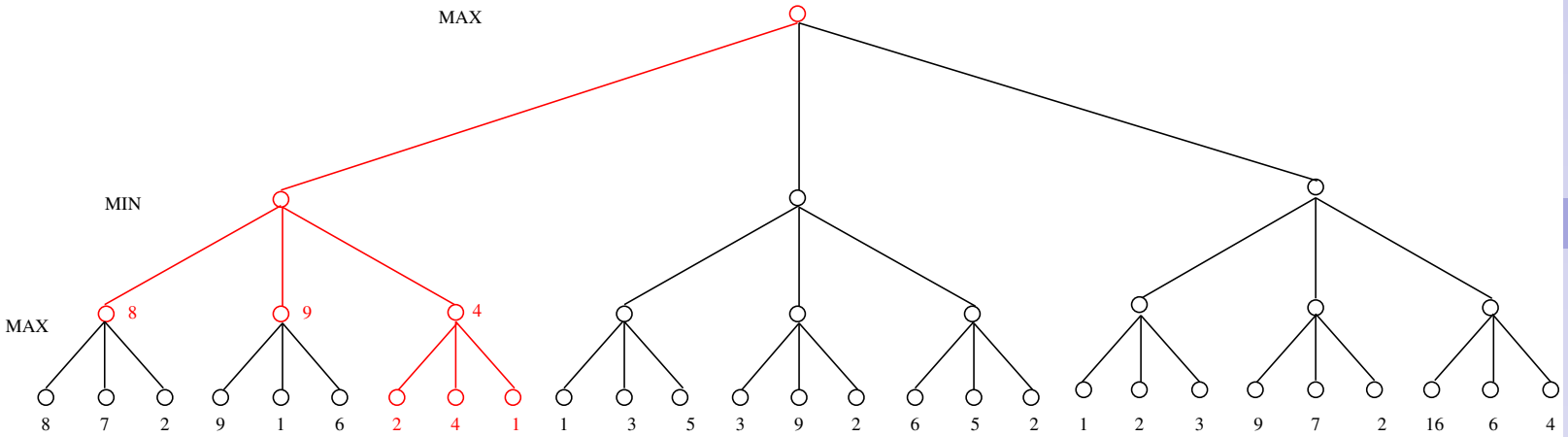
Example 2



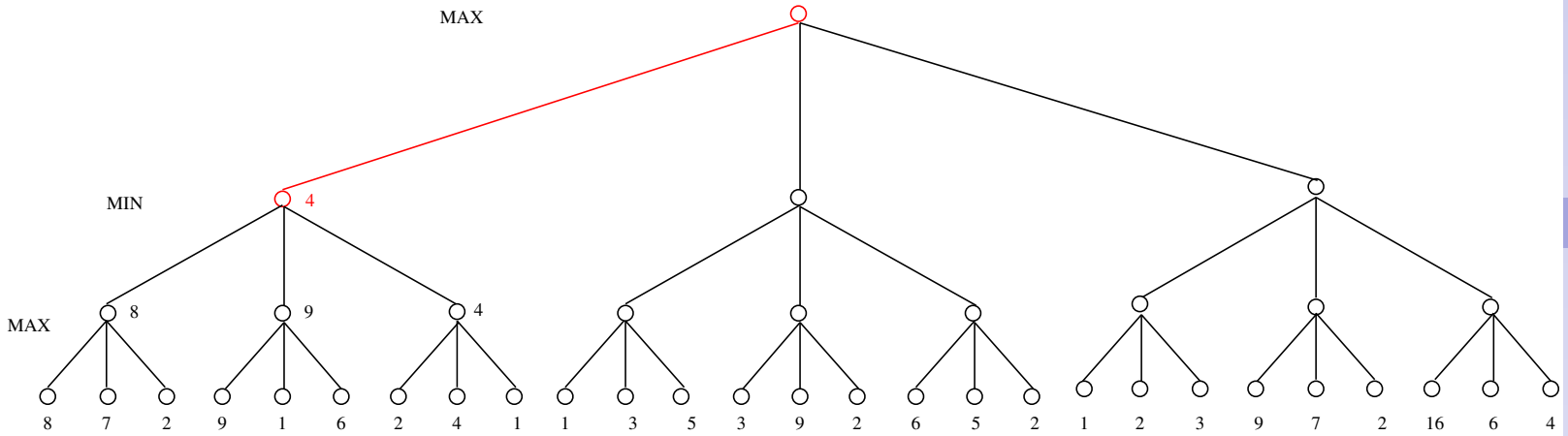
Example 2



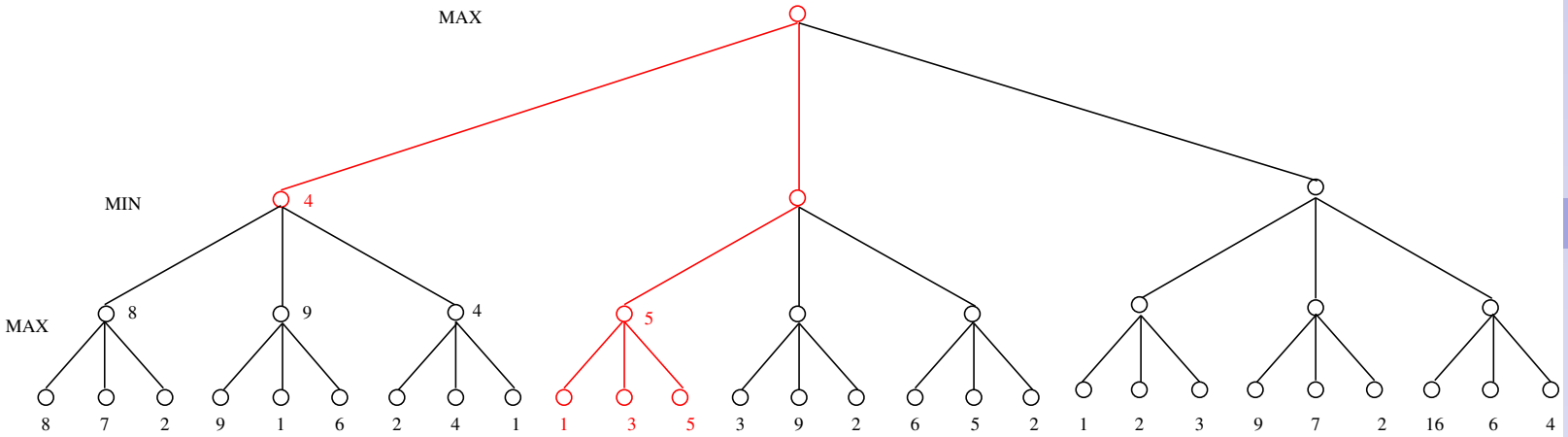
Example 2



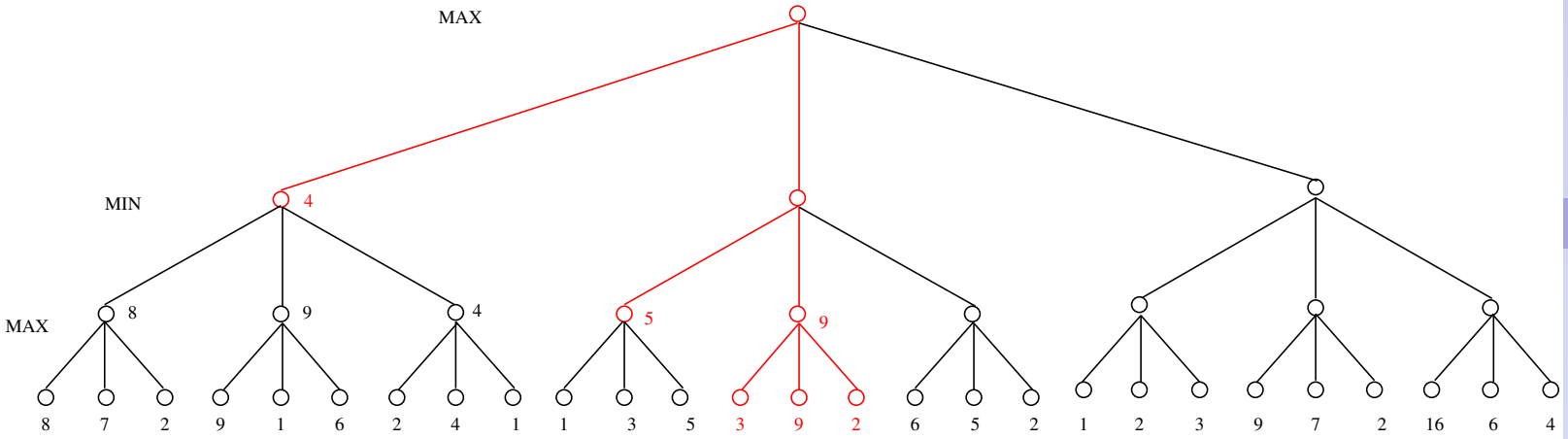
Example 2



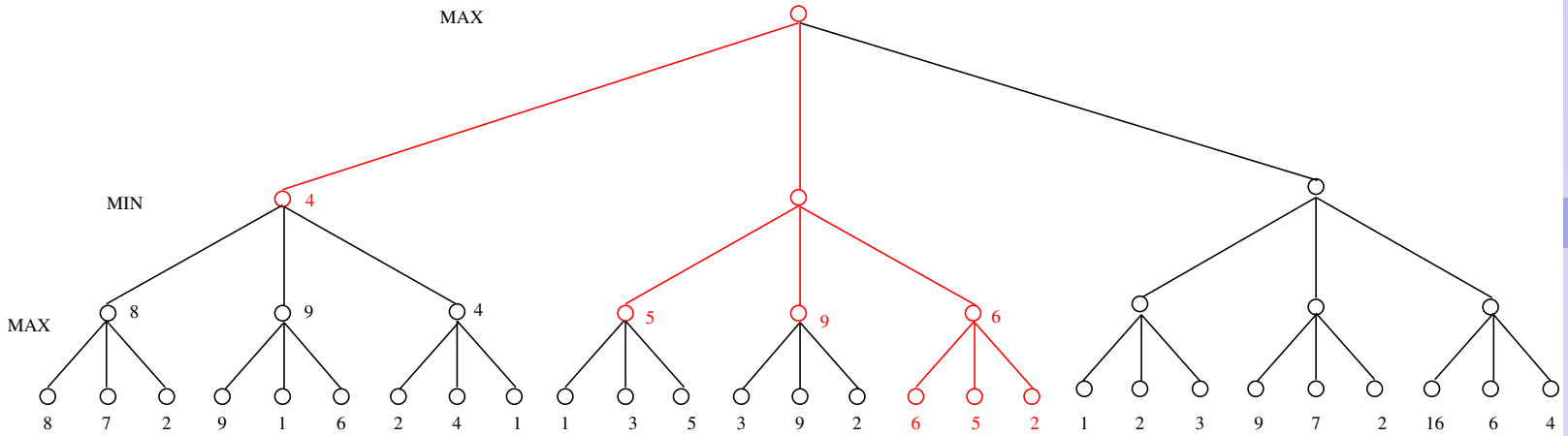
Example 2



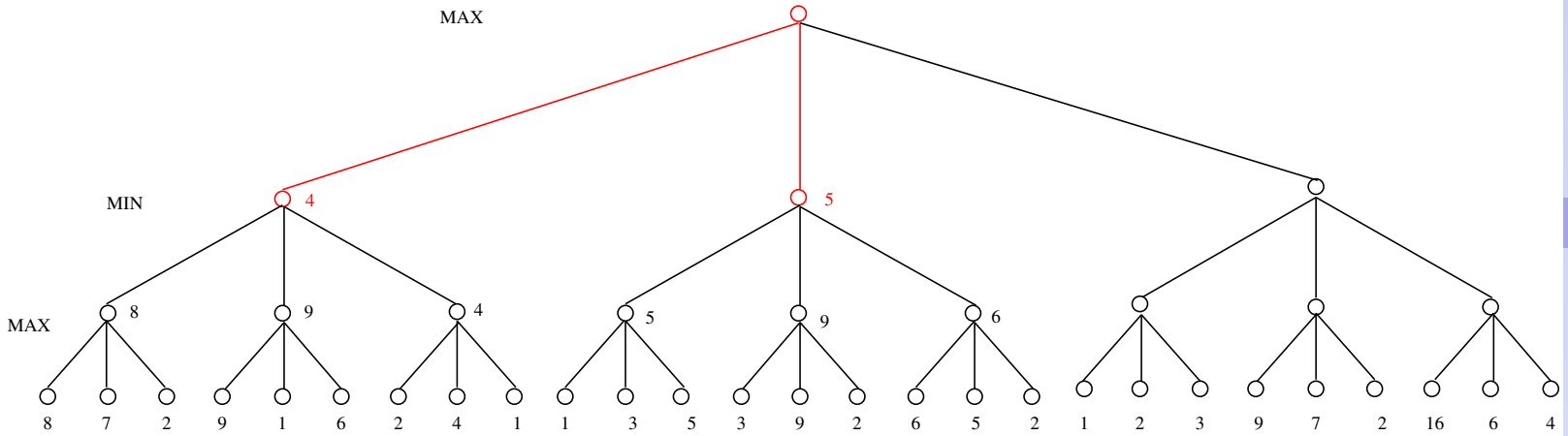
Example 2



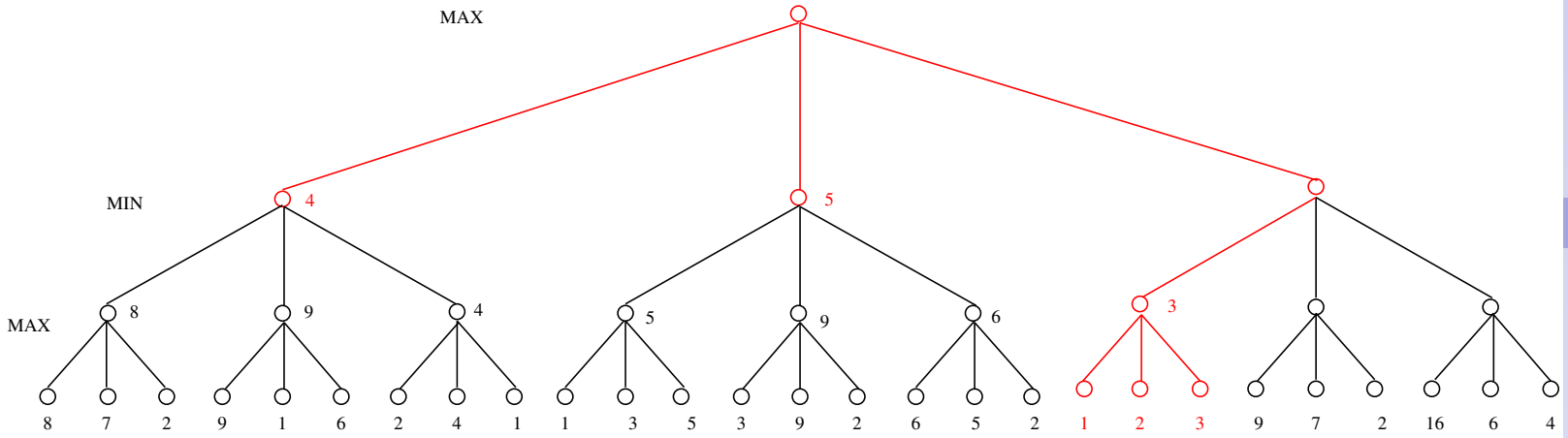
Example 2



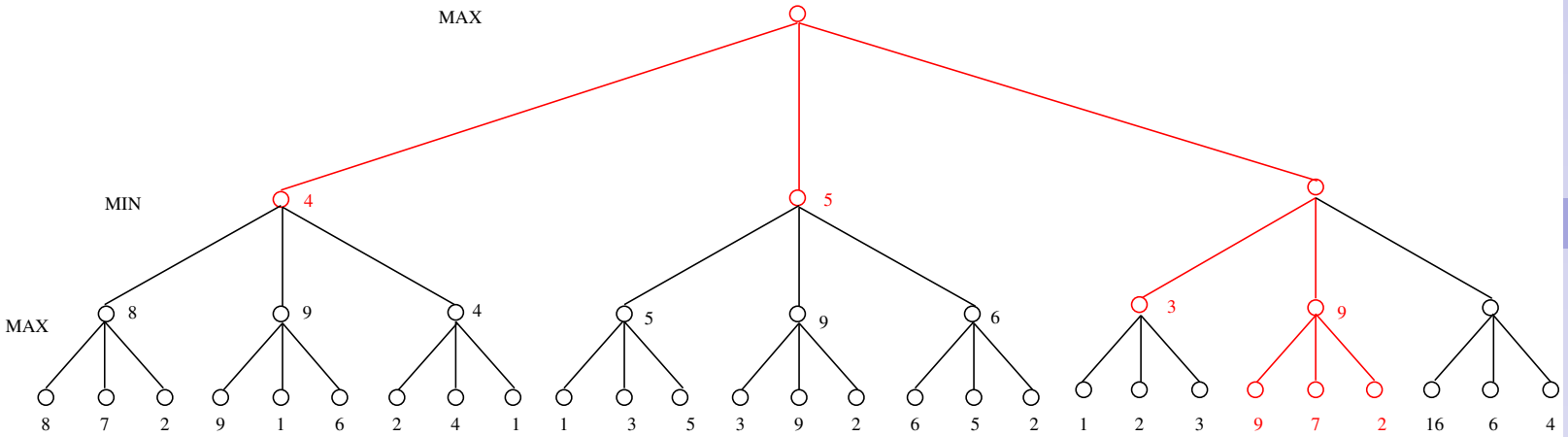
Example 2



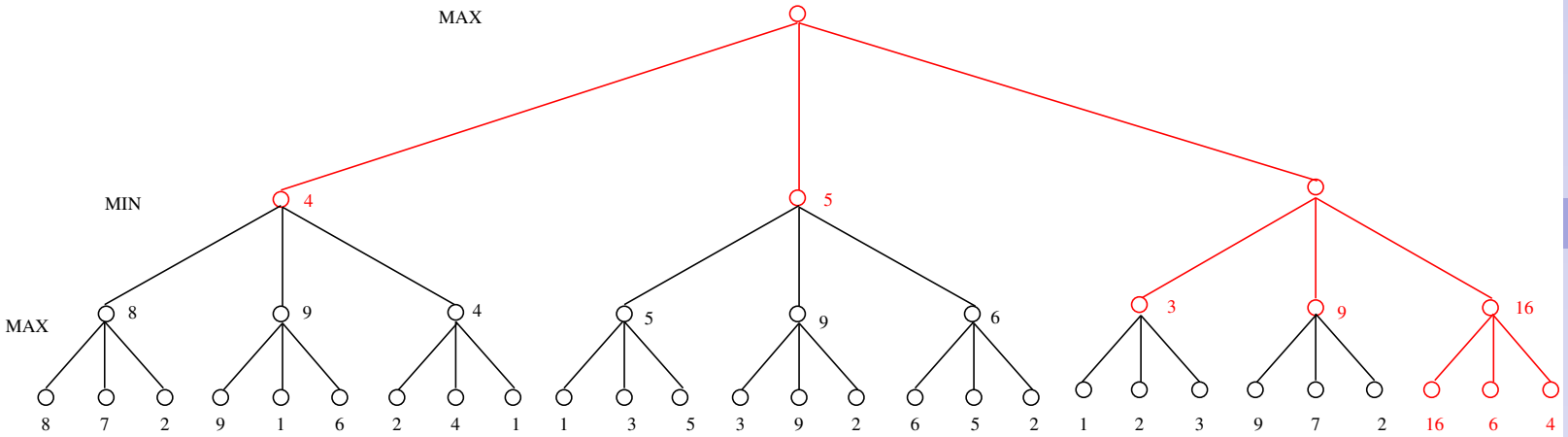
Example 2



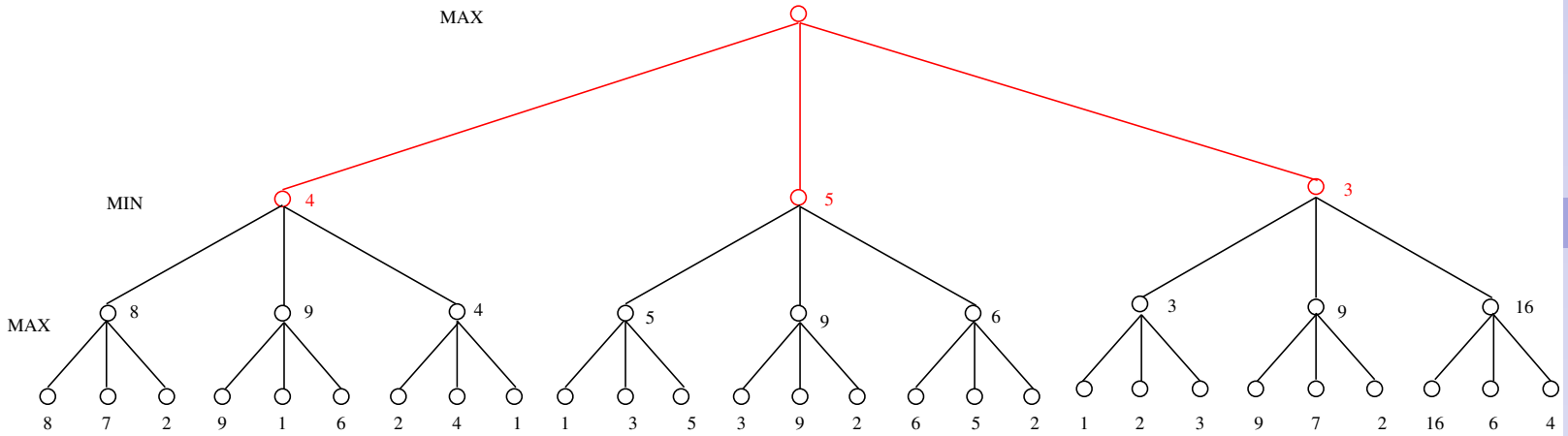
Example 2



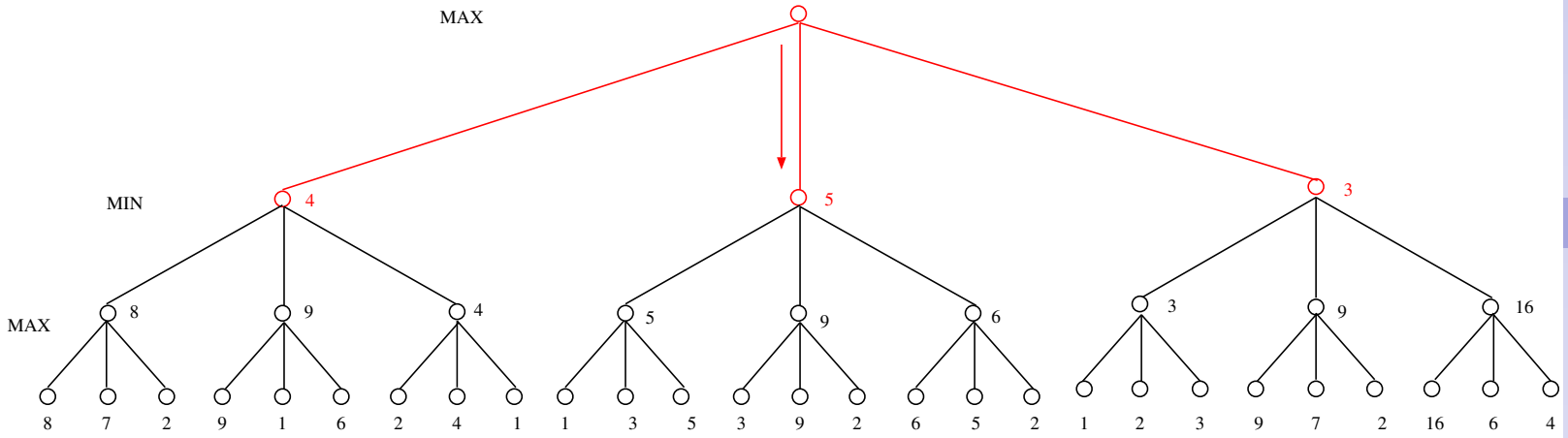
Example 2



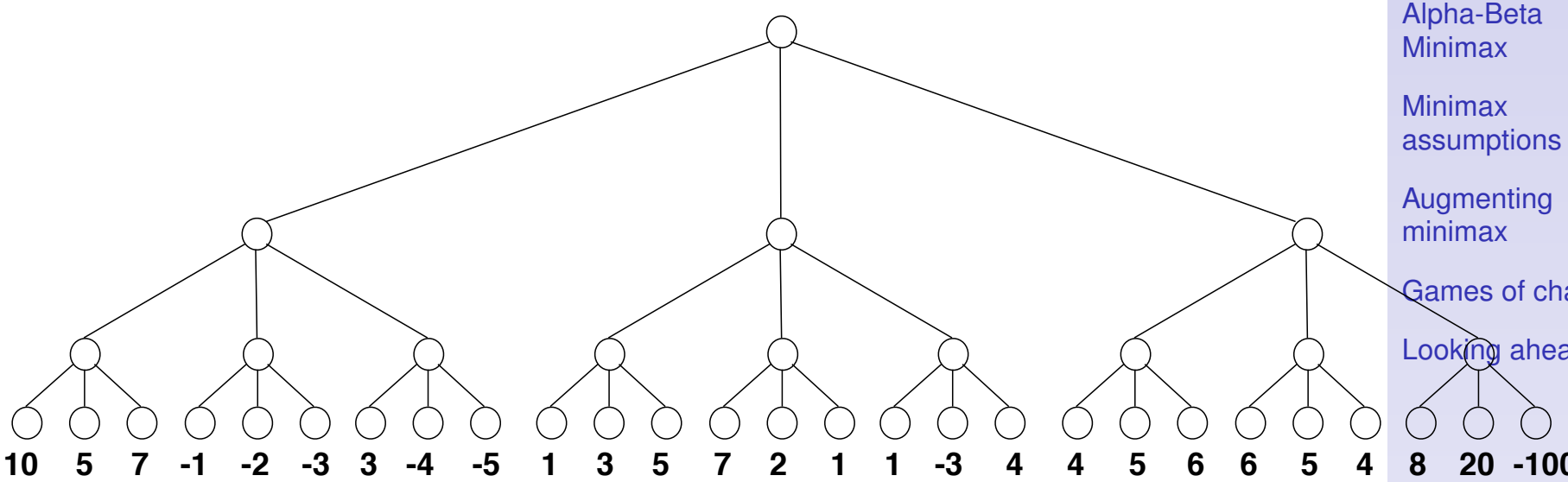
Example 2



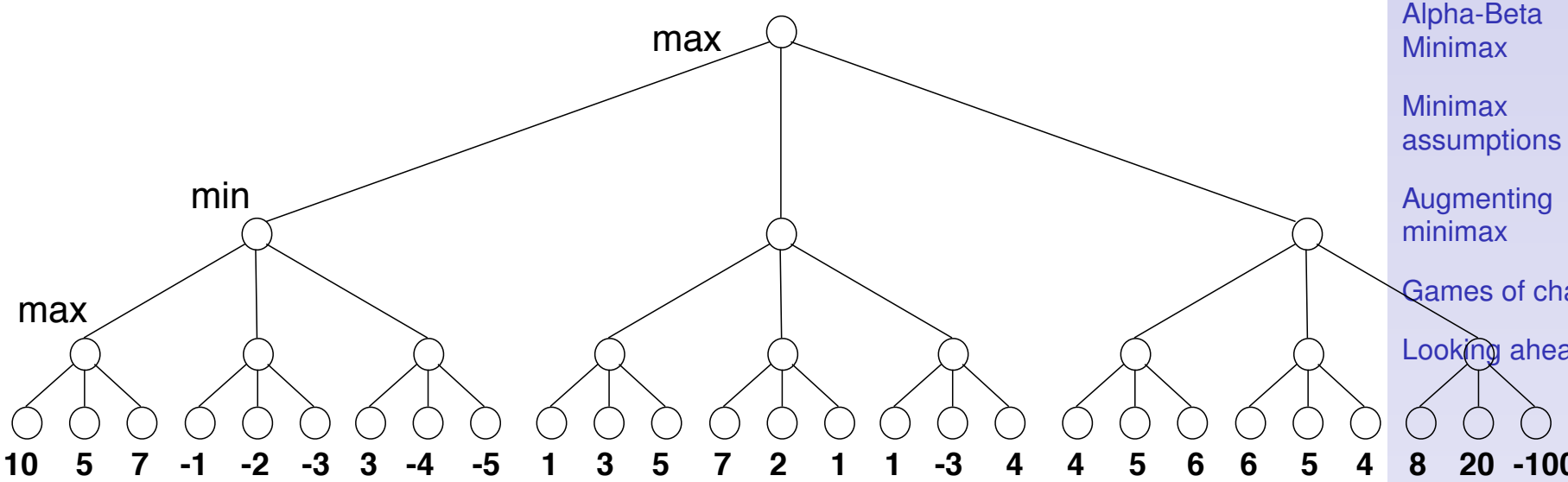
Example 2



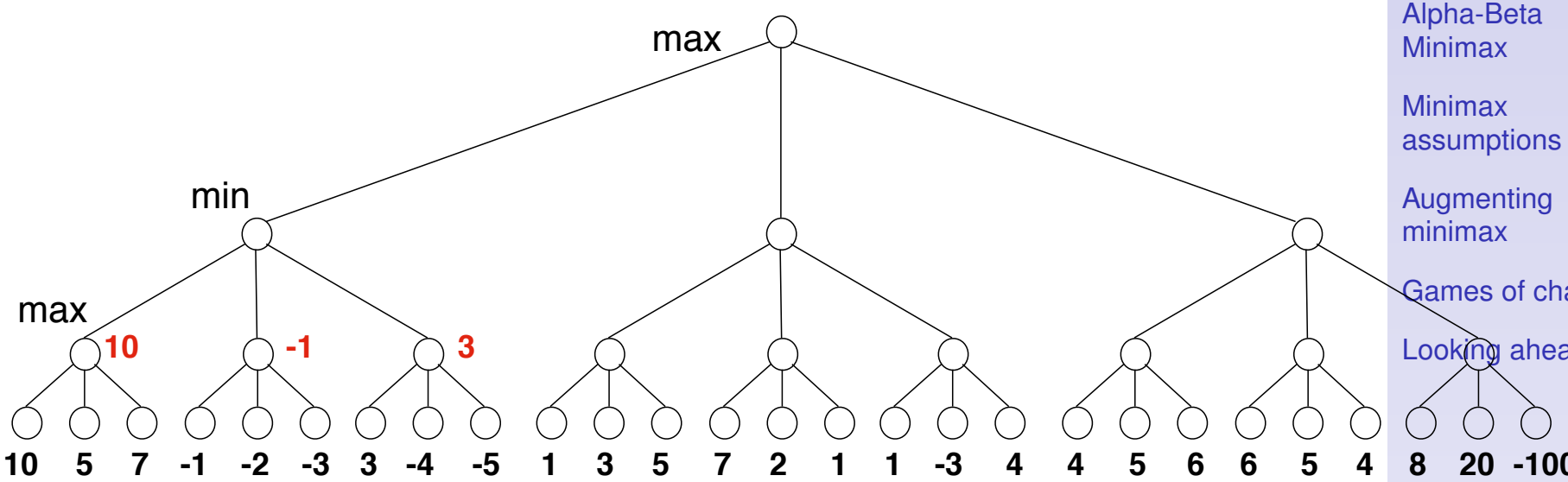
Minimax: Your turn!



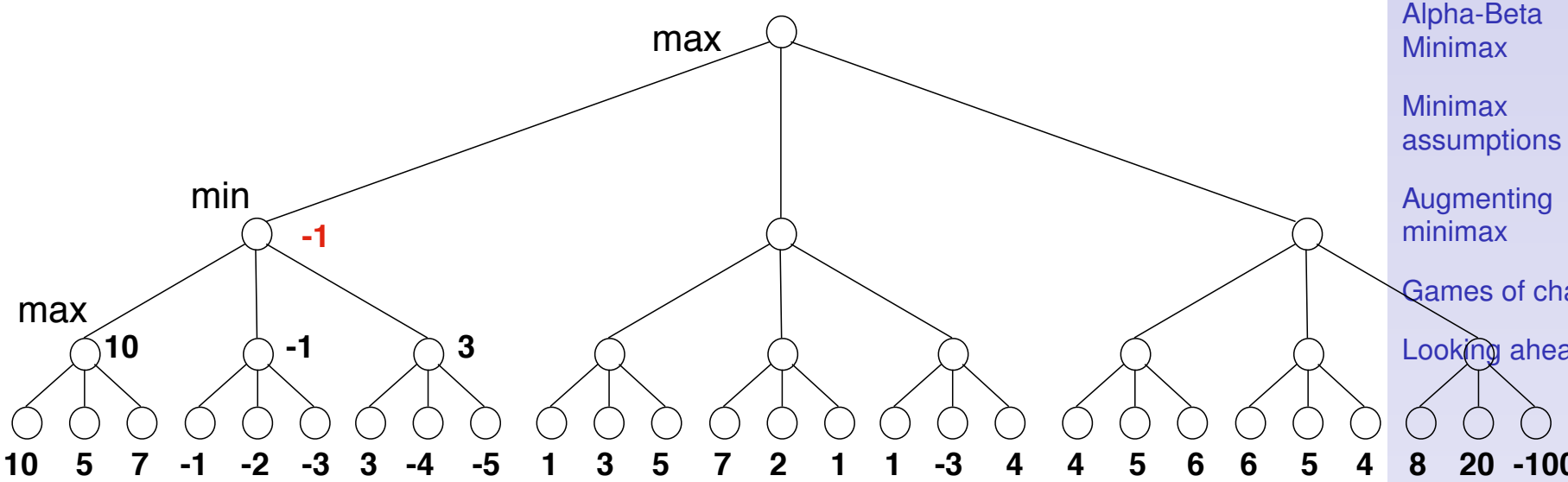
Minimax: Your turn!



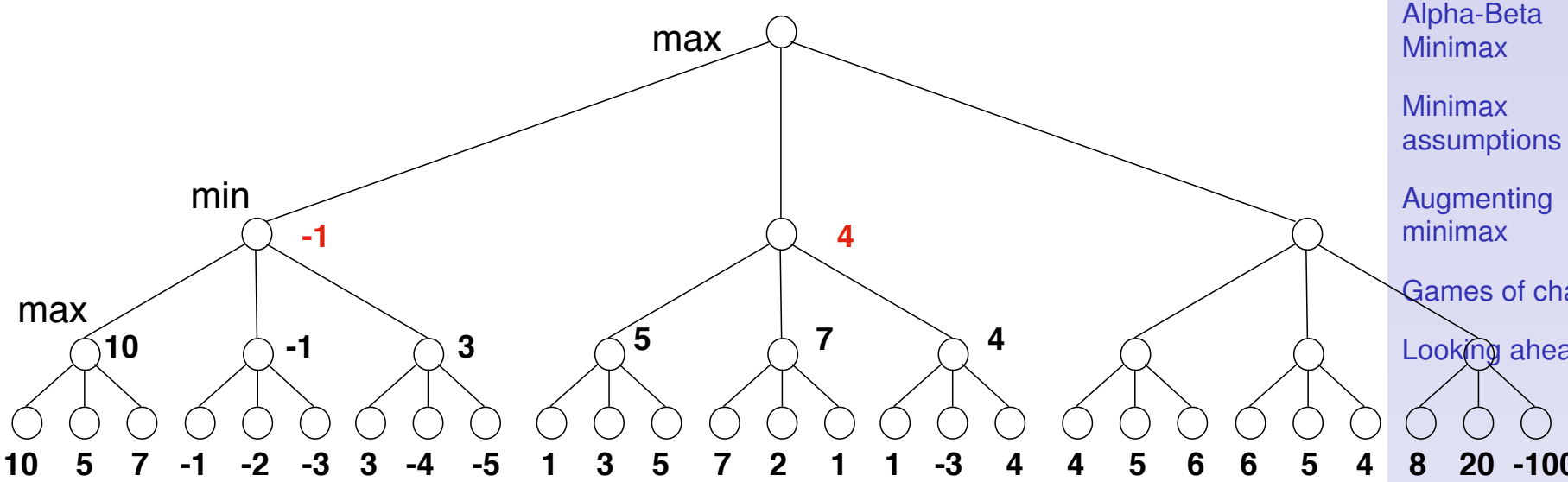
Minimax: Your turn!



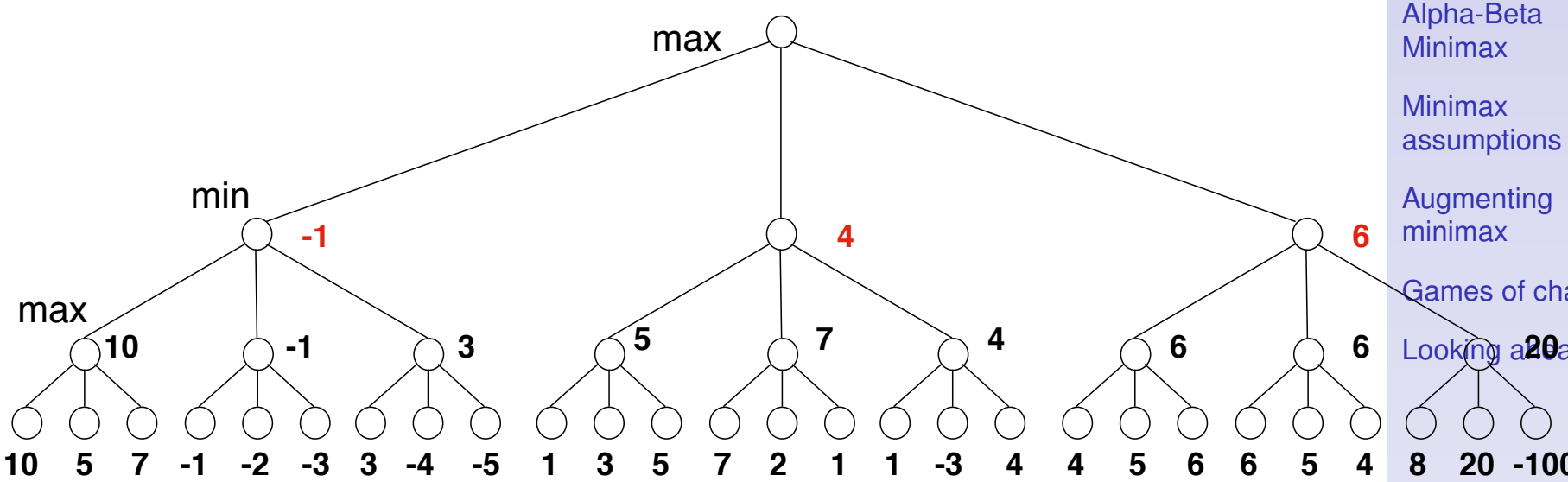
Minimax: Your turn!



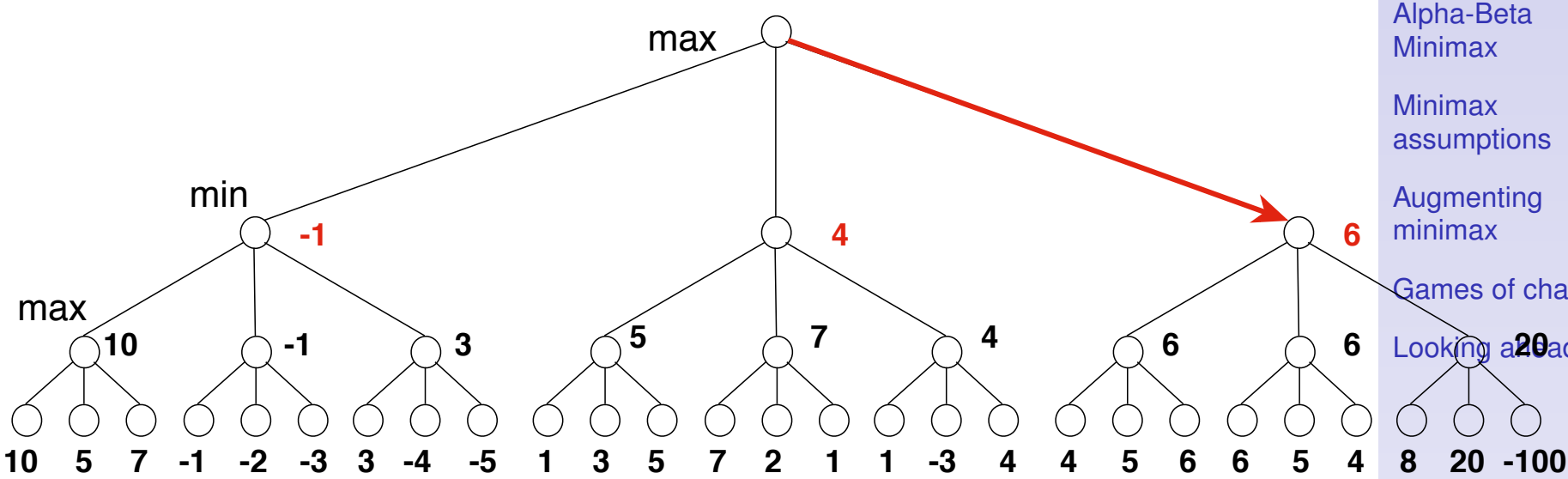
Minimax: Your turn!



Minimax: Your turn!



Minimax: Your turn!



Minimax algorithm

function MINIMAX(board,player,cutoff,depth=0)
  Inputs: a board + T/F (player/opponent) +
    cutoff + current depth
  Returns: move
  if depth = cutoff **then**
    Return STATIC-EVAL(board)
  else if player = T **then**
    Return argmax(STATIC-EVAL,CHILDREN(board))
  else
    Return argmin(STATIC-EVAL,CHILDREN(board))

Minimax program 1

```
(defun minimax (node &key (depth 5) (which :maximizer))
  (cond
   ((zerop depth)                          ;at max ply
    (static-eval node))
   (t
    (let (best-value best-child
          current-value)
      (loop for child in (children node)
          do
            ;; call minimax on each child:
            (setq current-value
              (minimax child :depth (1- depth)
                       :which (if (eql which :maximizer)
                                  :minimizer
                                  :maximizer)))
            ;; keep track of best node:
            (when (or (null best-value)
                      (funcall (if (eql which :maximizer)
                                   #'>
                                   #'<)
                               current-value best-value))
              (setq best-value current-value
                    best-child child)))
      (values best-value best-child)))))
```

Minimax program 2:

```
(defun minimax (board &key (depth 0) (cutoff 5) (player? t))
  (cond
   ((= depth cutoff) (static-eval board))
   (t (multiple-value-bind (child value)
          (if player?
            (argmax #'(lambda (b)
                        (minimax b
                                 :depth (1+ depth)
                                 :cutoff cutoff
                                 :player? (not player?)))
                    (children board))
            (argmin #'(lambda (b)
                        (minimax b
                                 :depth (1+ depth)
                                 :cutoff cutoff
                                 :player? (not player?)))
                    (children board))))
      (if (= depth 0)
        (move child)
        value))))
```

Minimax program 2 (argmax):

```
(defmacro argmax (fcn things &optional best)
  `(argcmp ,fcn ,things #'> ,best))

(defmacro argmin (fcn things &optional best)
  `(argcmp ,fcn ,things #'< ,best))

(defun argcmp (fcn things cmp &optional best)
  (if (null things)
    (values (cadr best) (car best))
    (let ((val (funcall fcn (car things))))
      (when (or (null best) (funcall cmp val (car best)))
        (setq best (list val (car things))))
      (argcmp fcn (cdr things) cmp best))))
```

Minimax program 3:

```
(defun minimax (board &key (depth 0) (cutoff 5) (player? t))
  (cond
   ((= depth cutoff) (static-eval board))
   (t (multiple-value-bind (child value)
          (argcmp #'(lambda (b)
                      (minimax b
                               :depth (1+ depth)
                               :cutoff cutoff
                               :player? (not player?)))
                  (children board)
                  (if player? #'> #'<)))
      (if (= depth 0)
        (move child)
        value))))
```

Minimax program 4:

```
(defun minimax (board &key (depth 0) (cutoff 5) (player? t))
  (when (= depth cutoff) (return (static-eval board)))
  (multiple-value-bind (child value)
      (argcmp #'(lambda (b)
                  (minimax b :depth (1+ depth)
                             :cutoff cutoff
                             :player? (not player?)))
              (children board)
              (if player? #'> #'<))
      (if (= depth 0)
        (move child)
        value)))
```

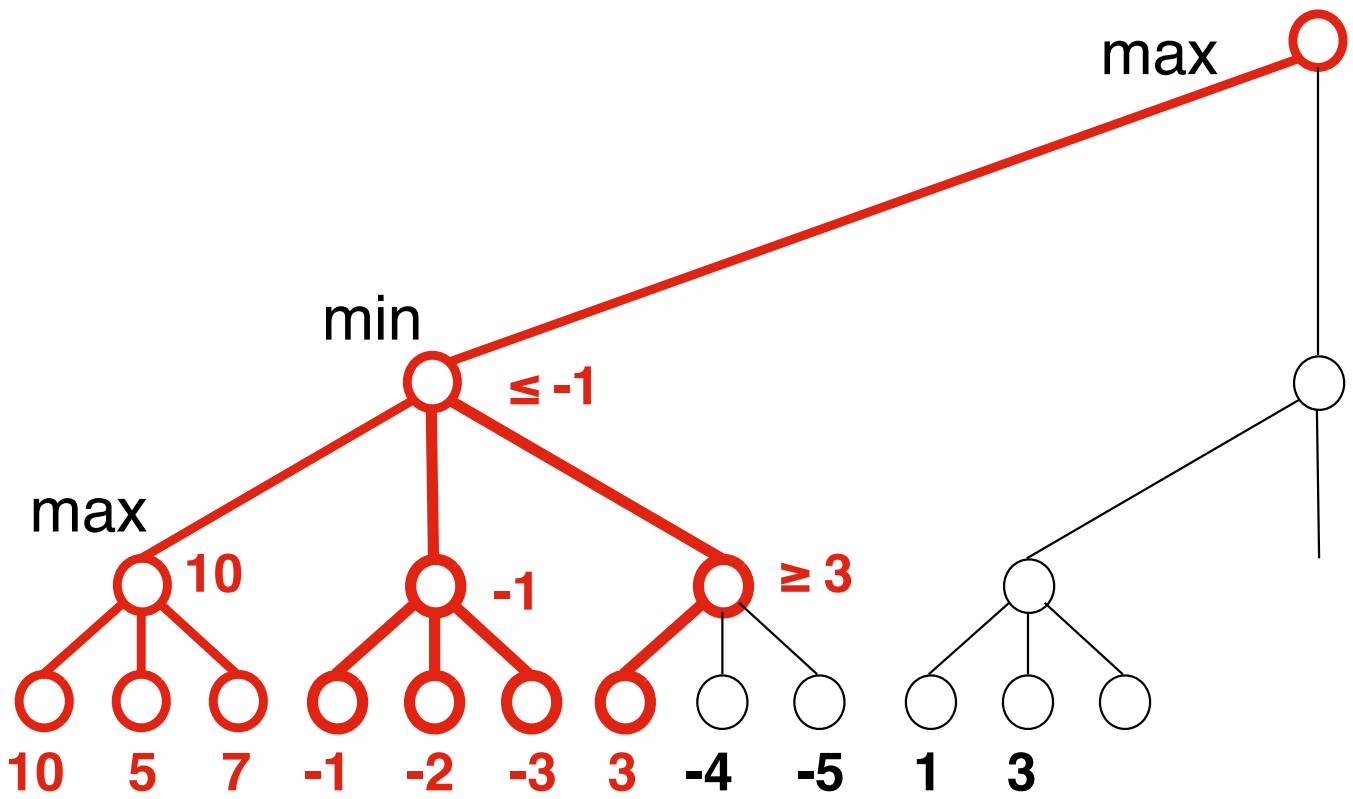
Alpha-Beta Minimax

Problem

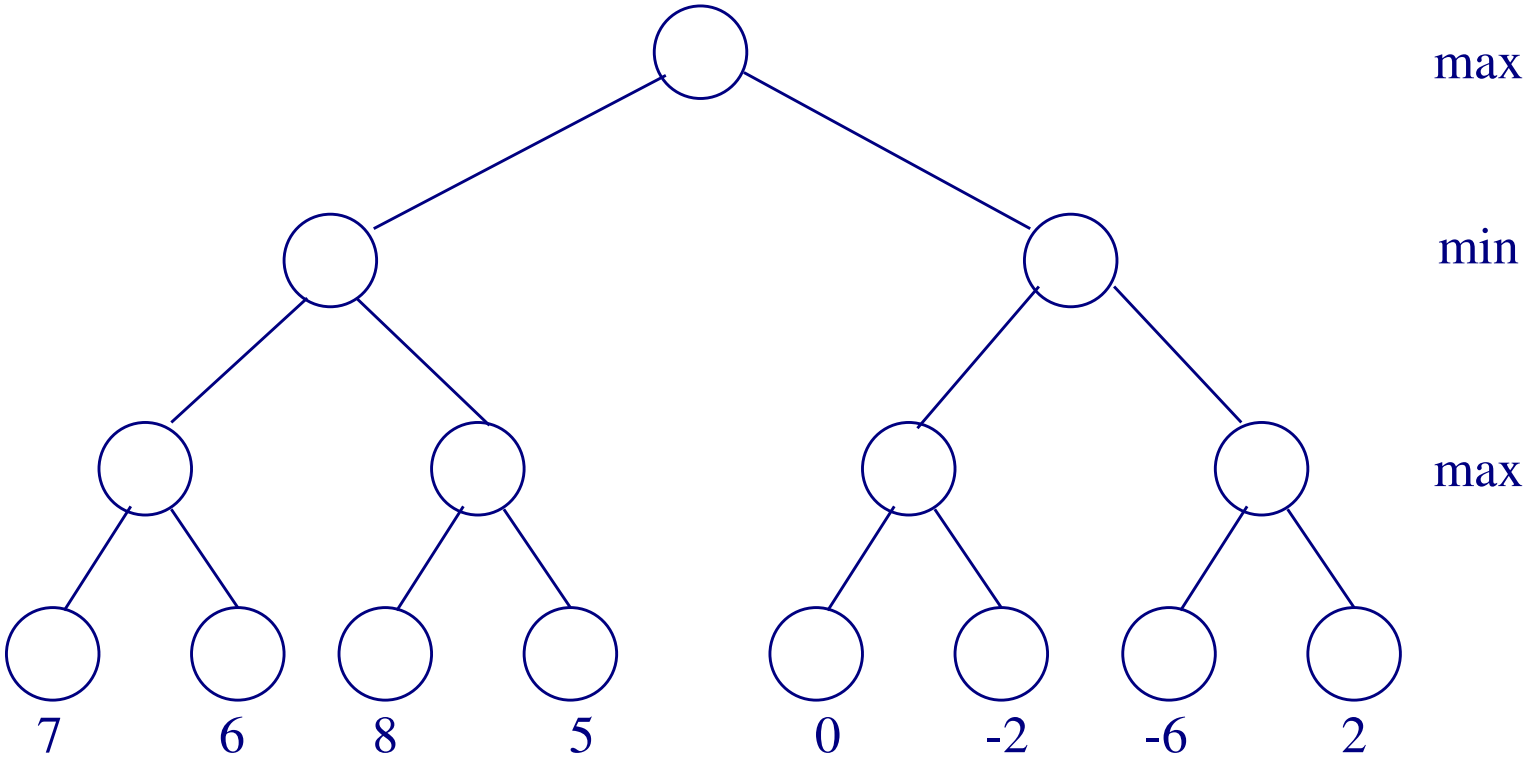
- Game trees can be *very* large
- Is there any way to prune the space?
- I.e., are there moves that *no one* will choose?

Problem

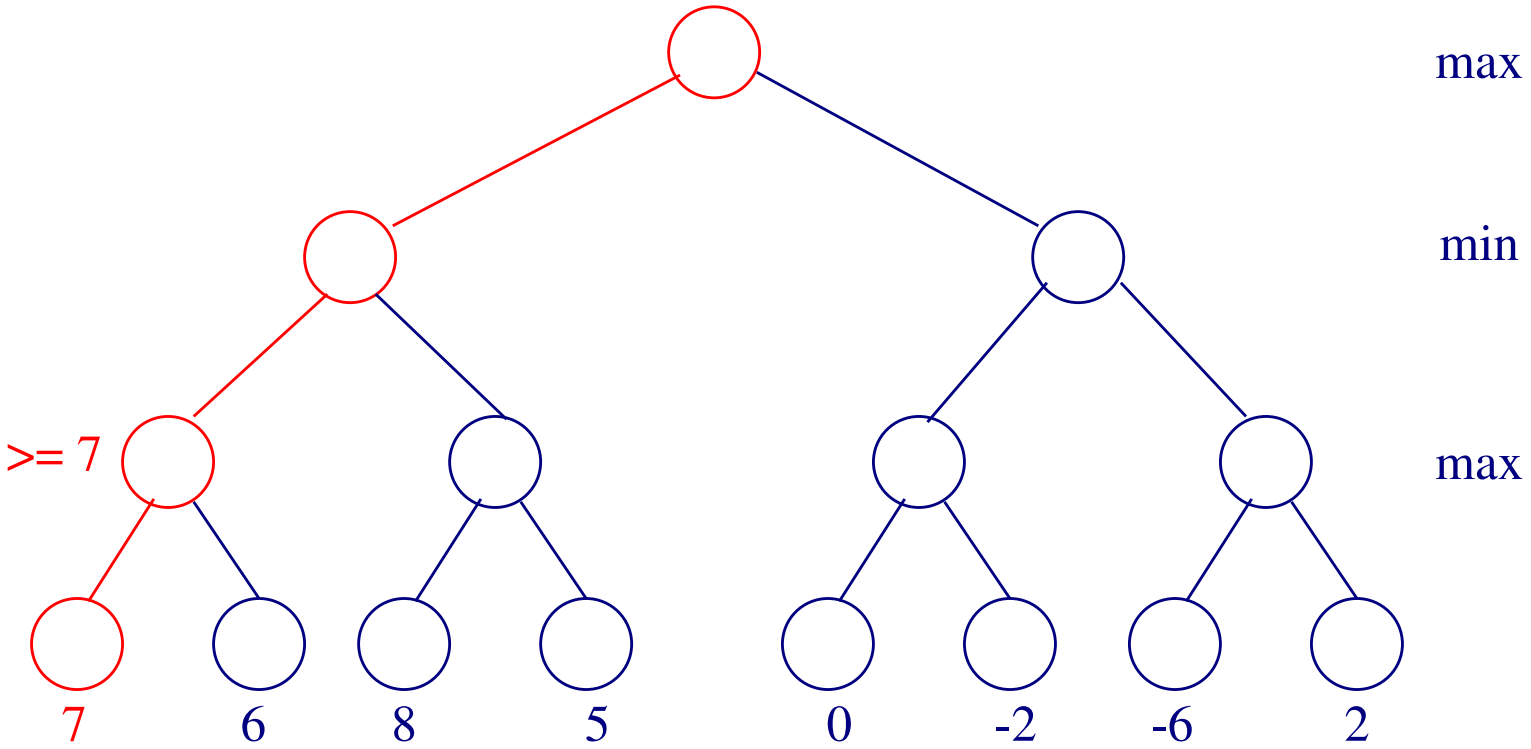
- Game trees can be *very* large
- Is there any way to prune the space?
- I.e., are there moves that *no one* will choose?



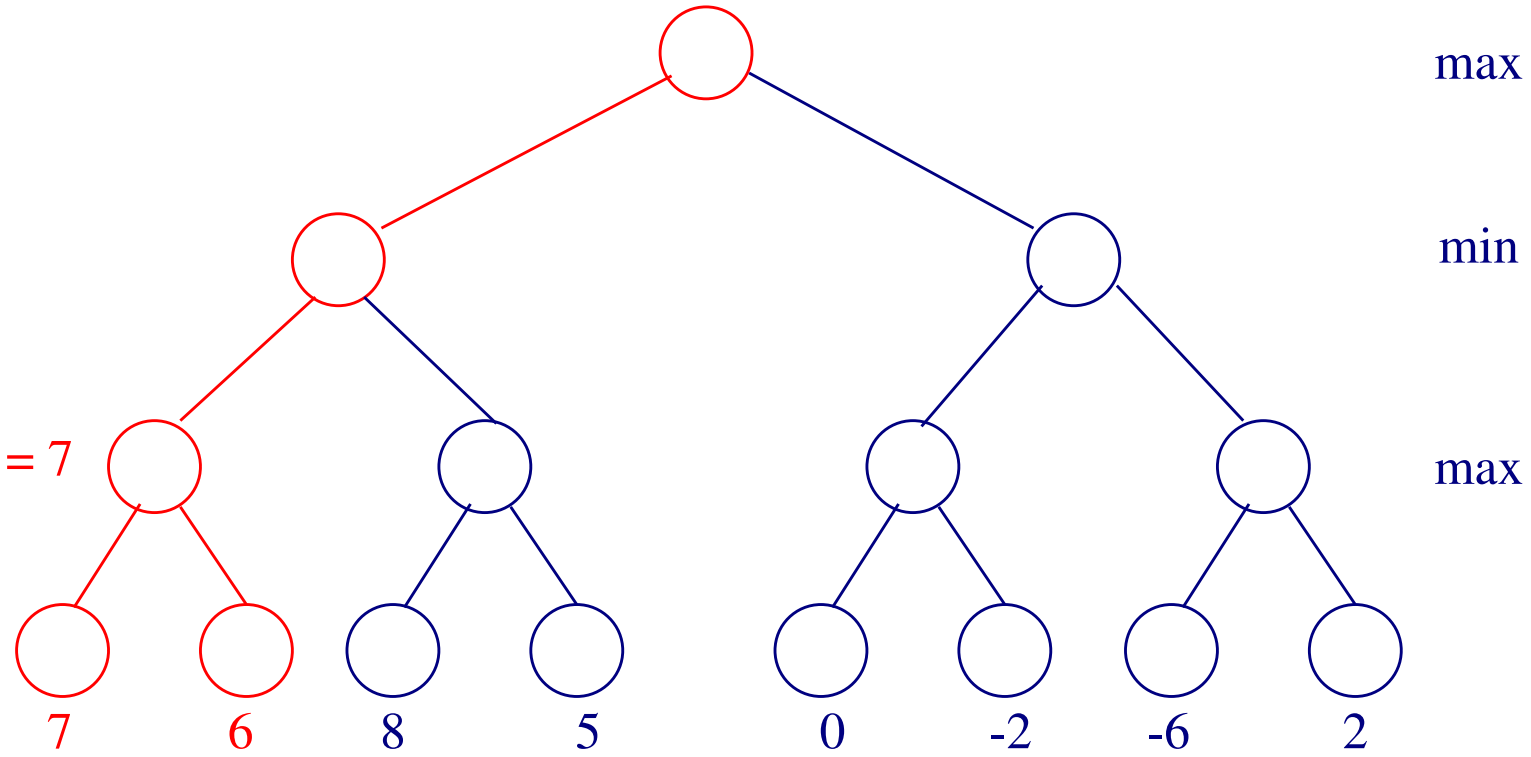
α–β cut-offs



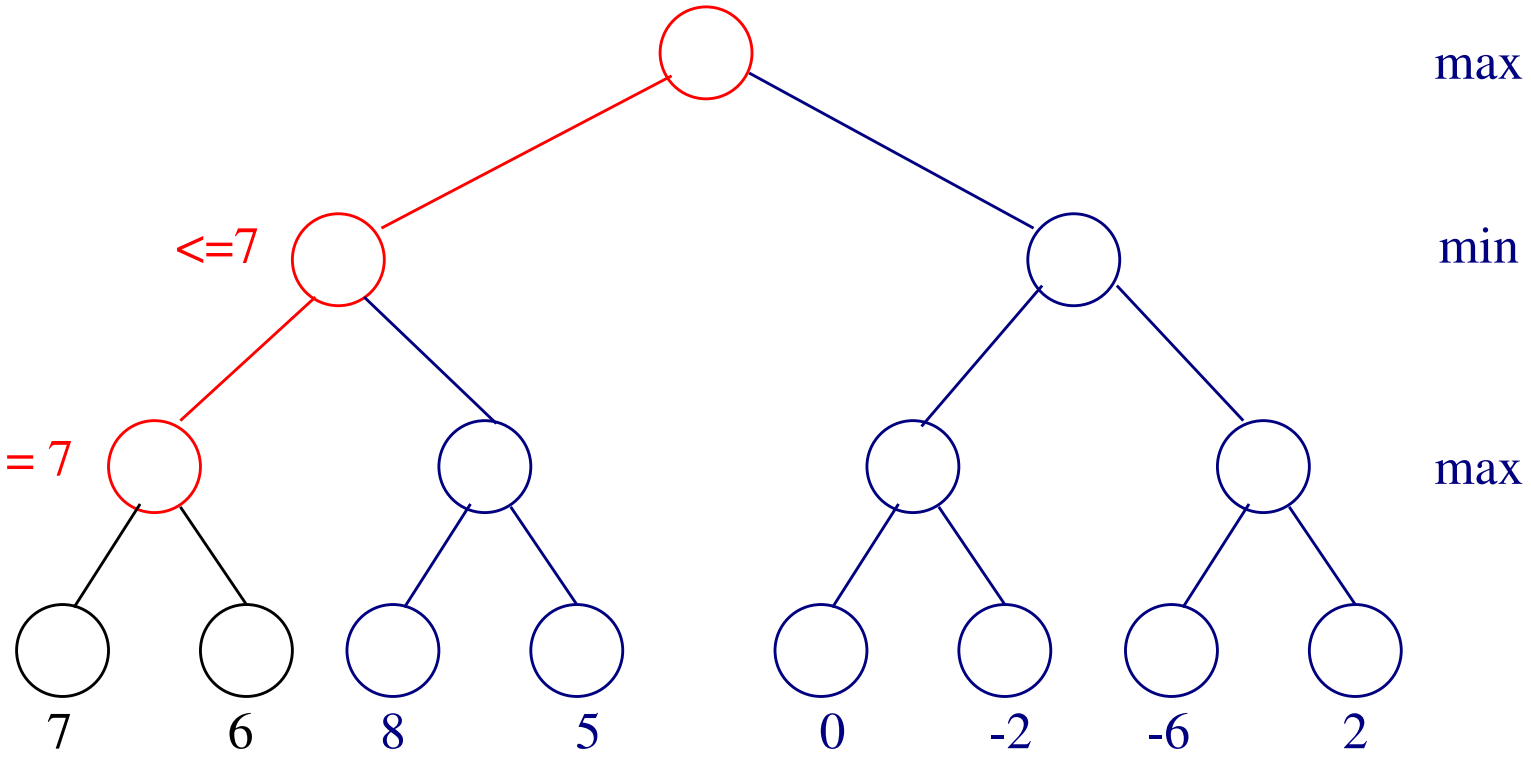
α–β cut-offs



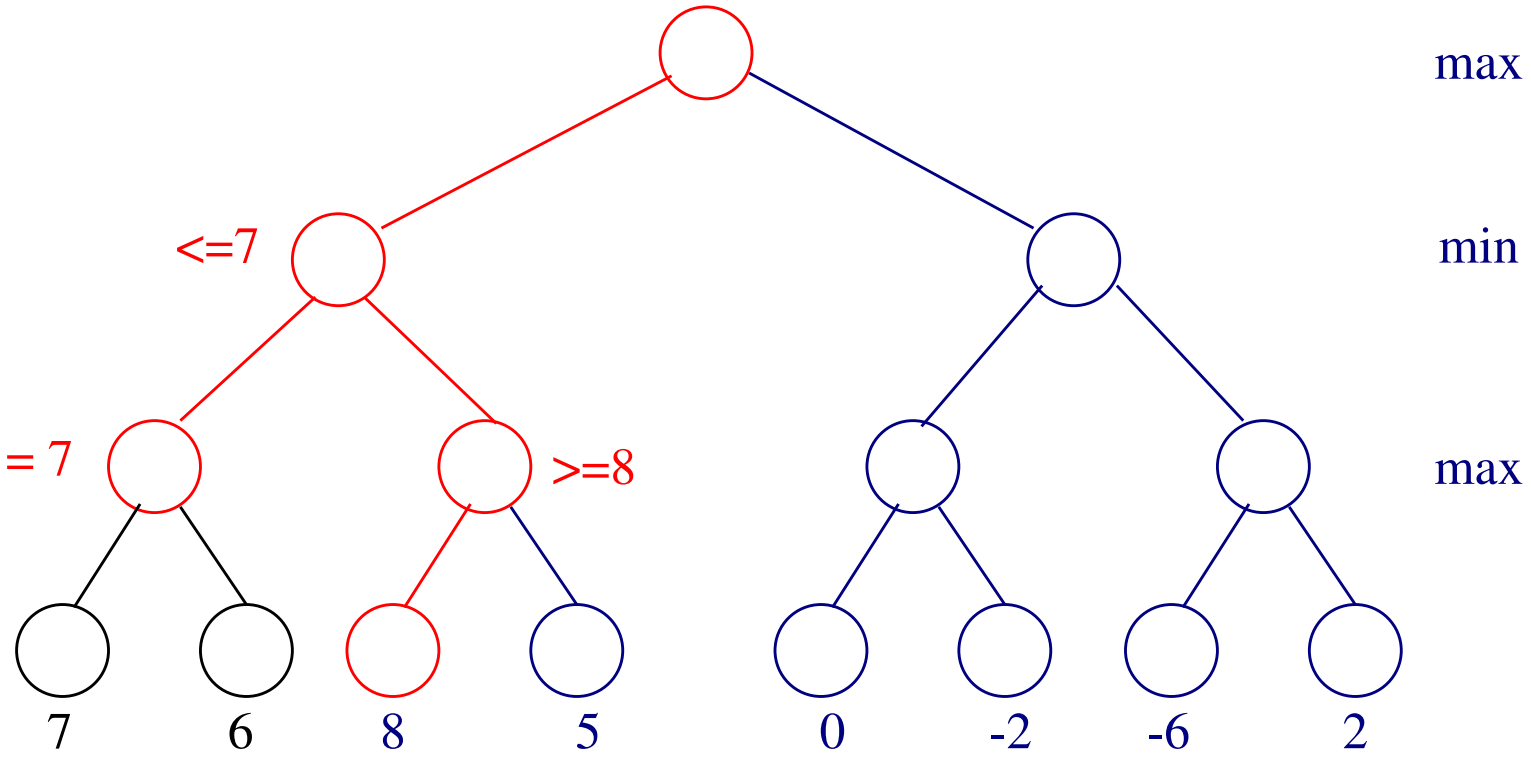
α–β cut-offs



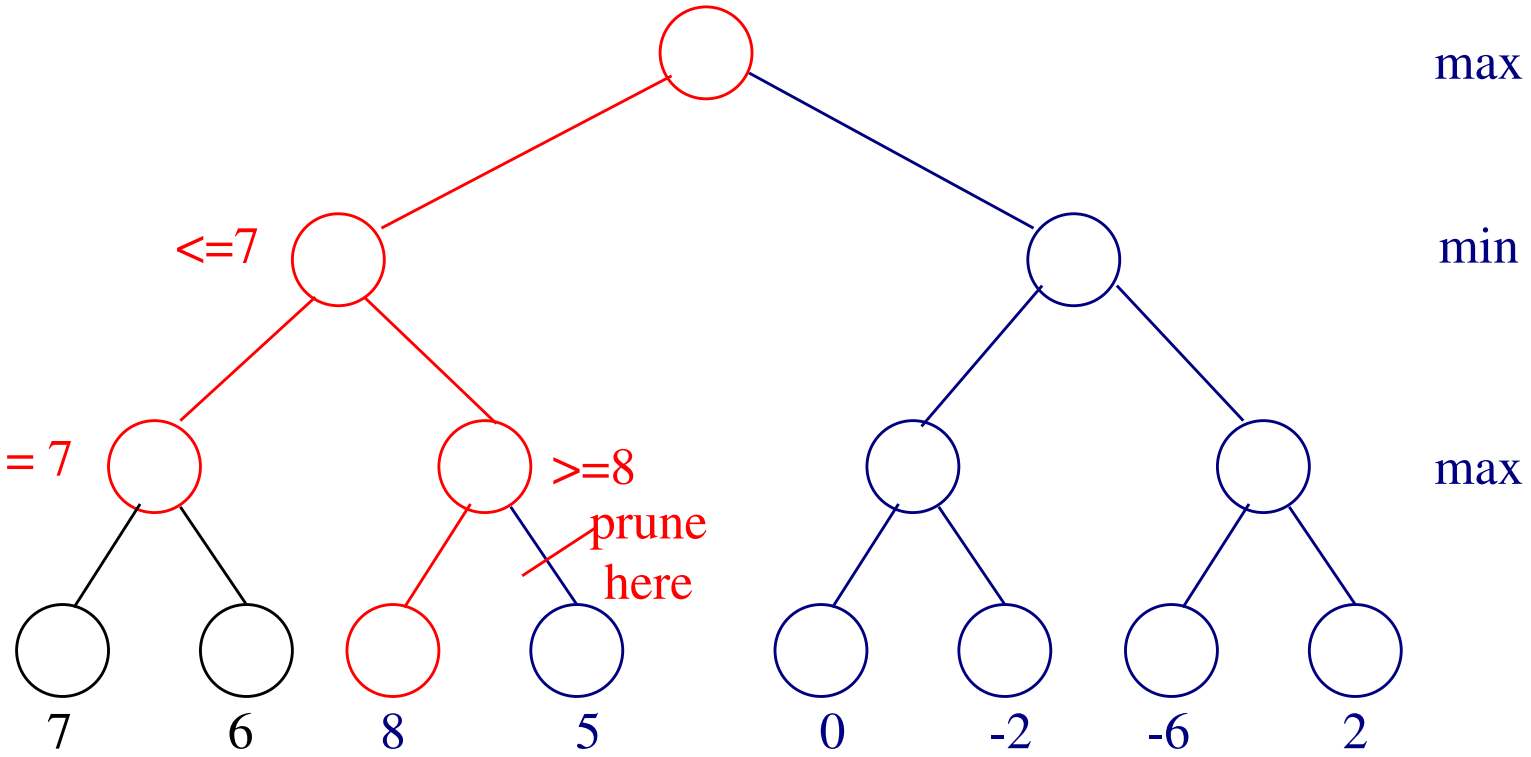
α–β cut-offs



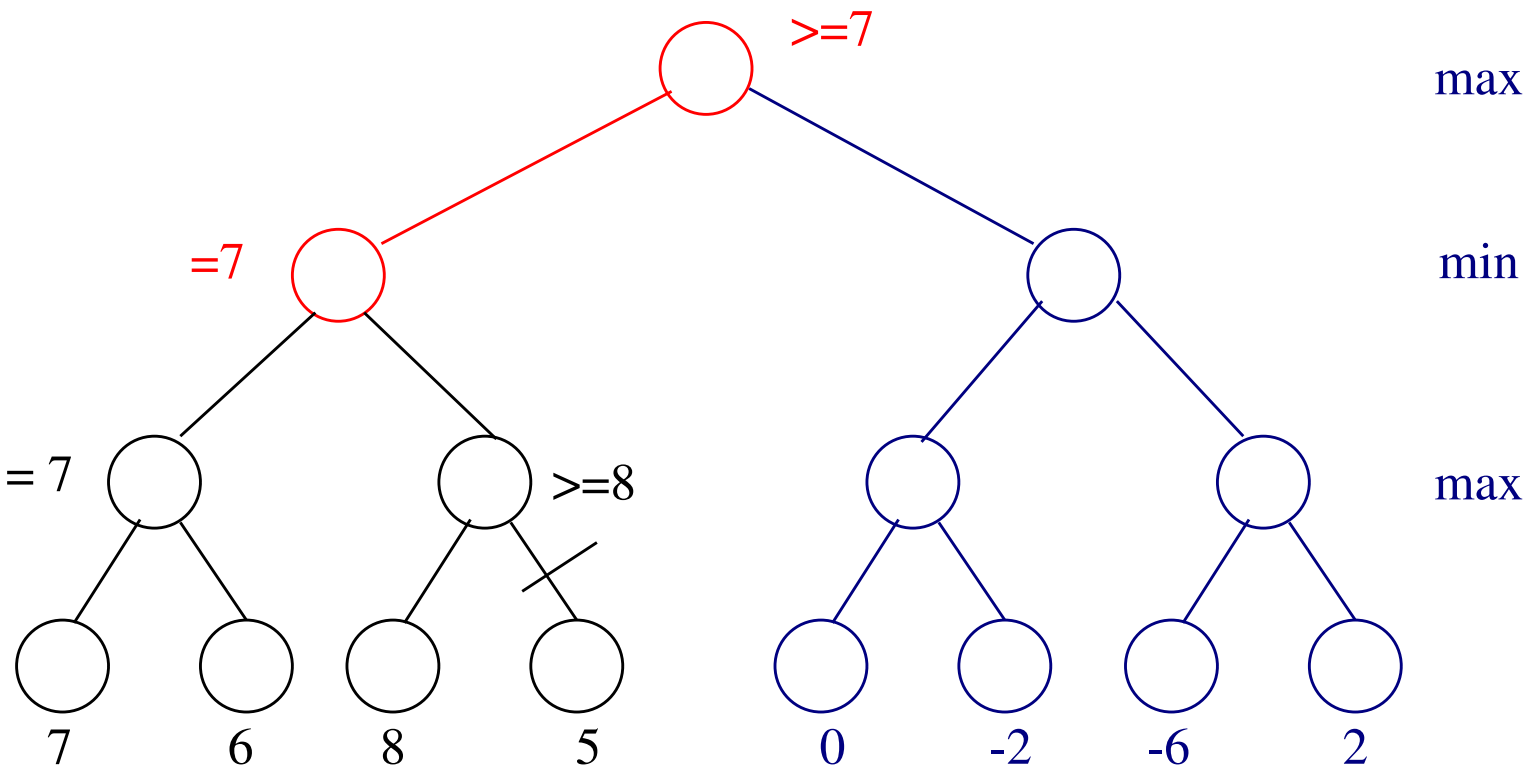
α–β cut-offs



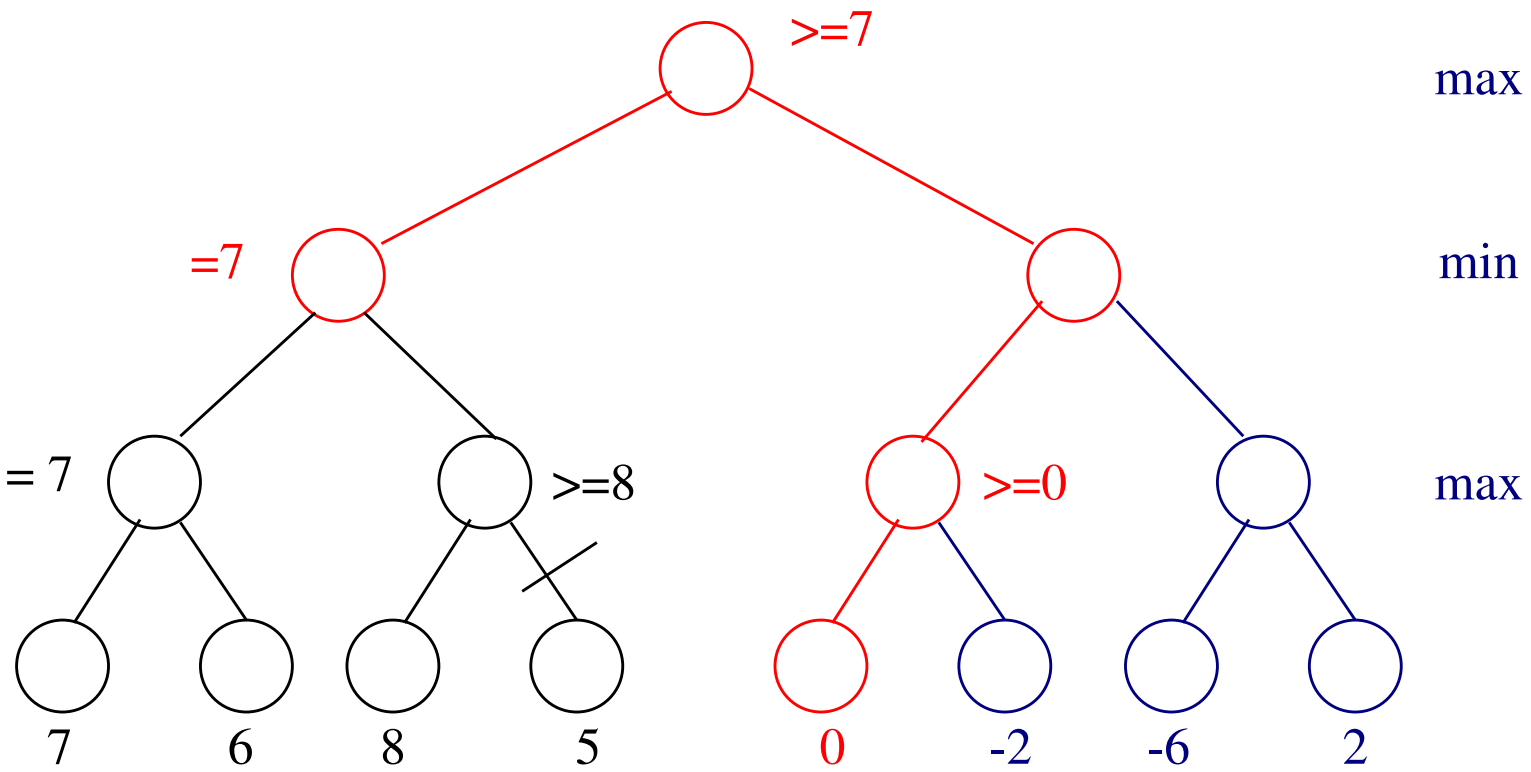
α–β cut-offs



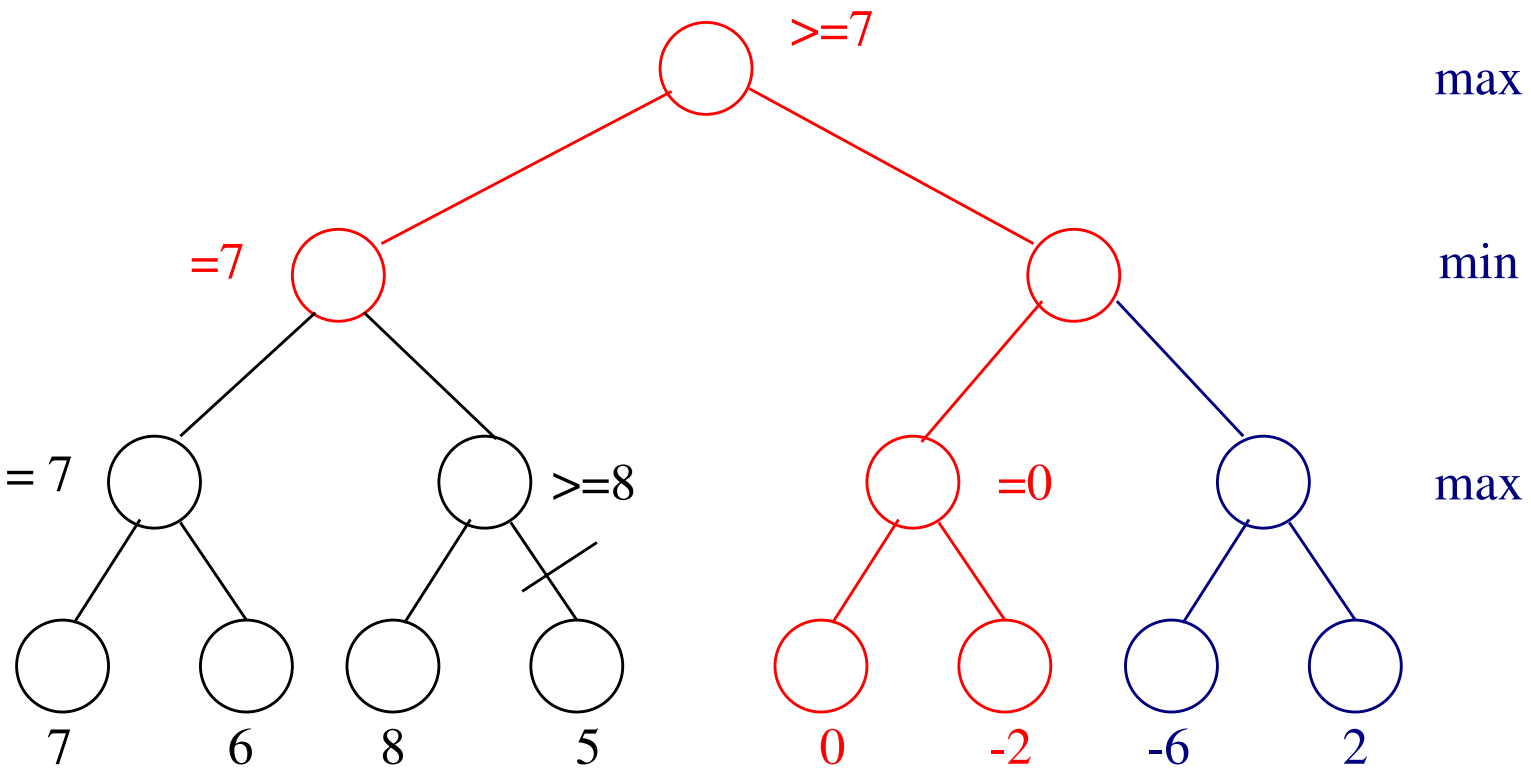
α–β cut-offs



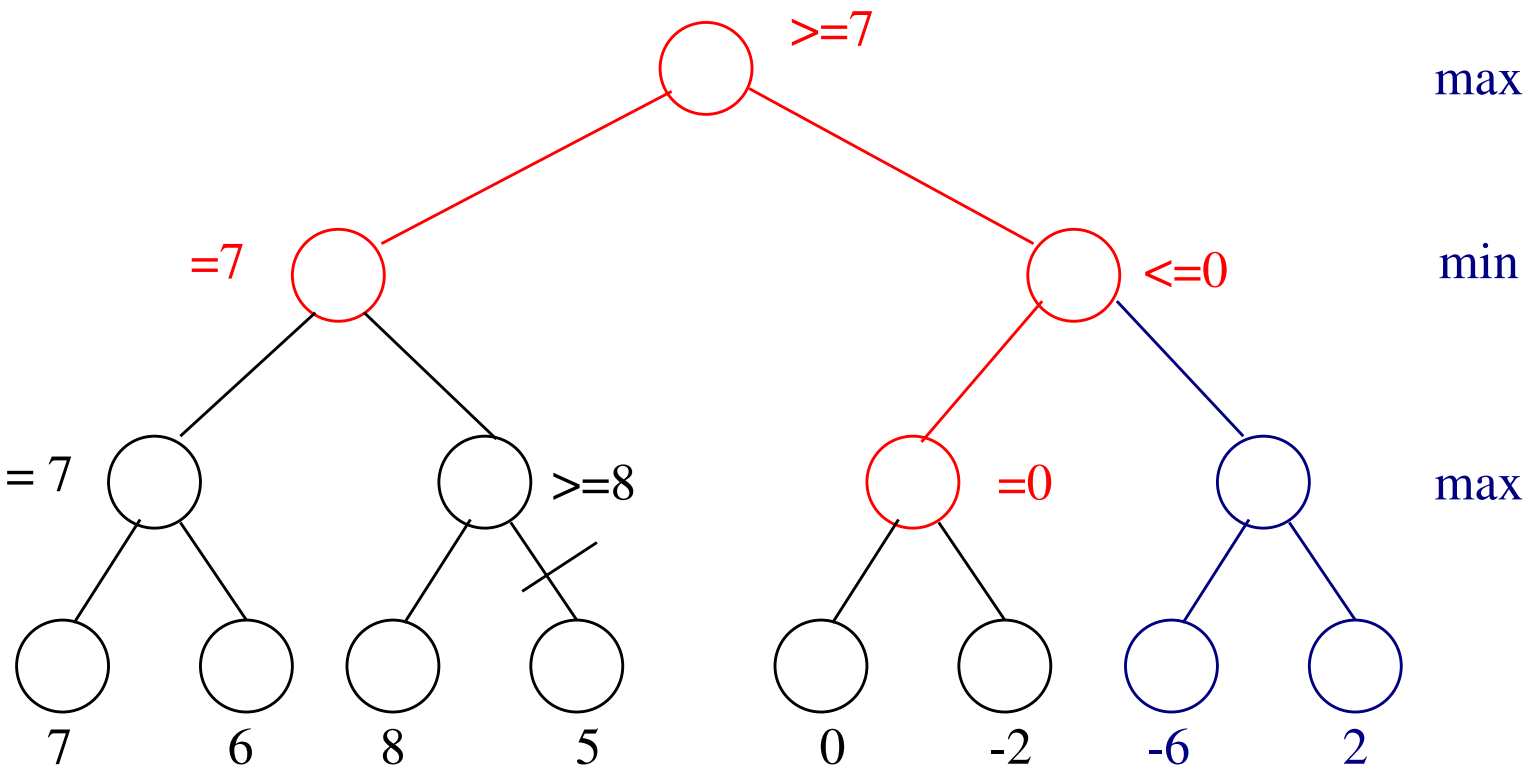
α–β cut-offs



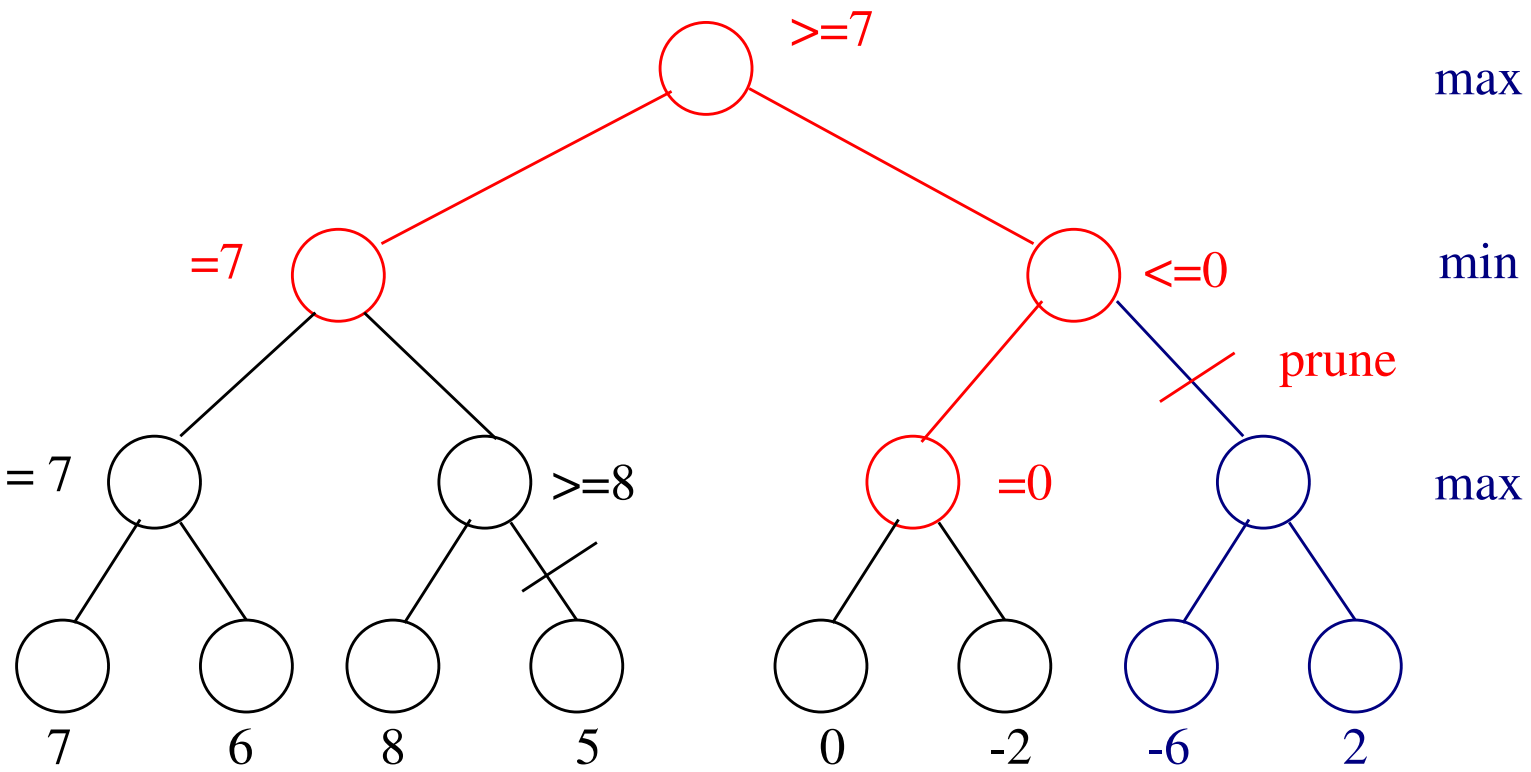
α–β cut-offs



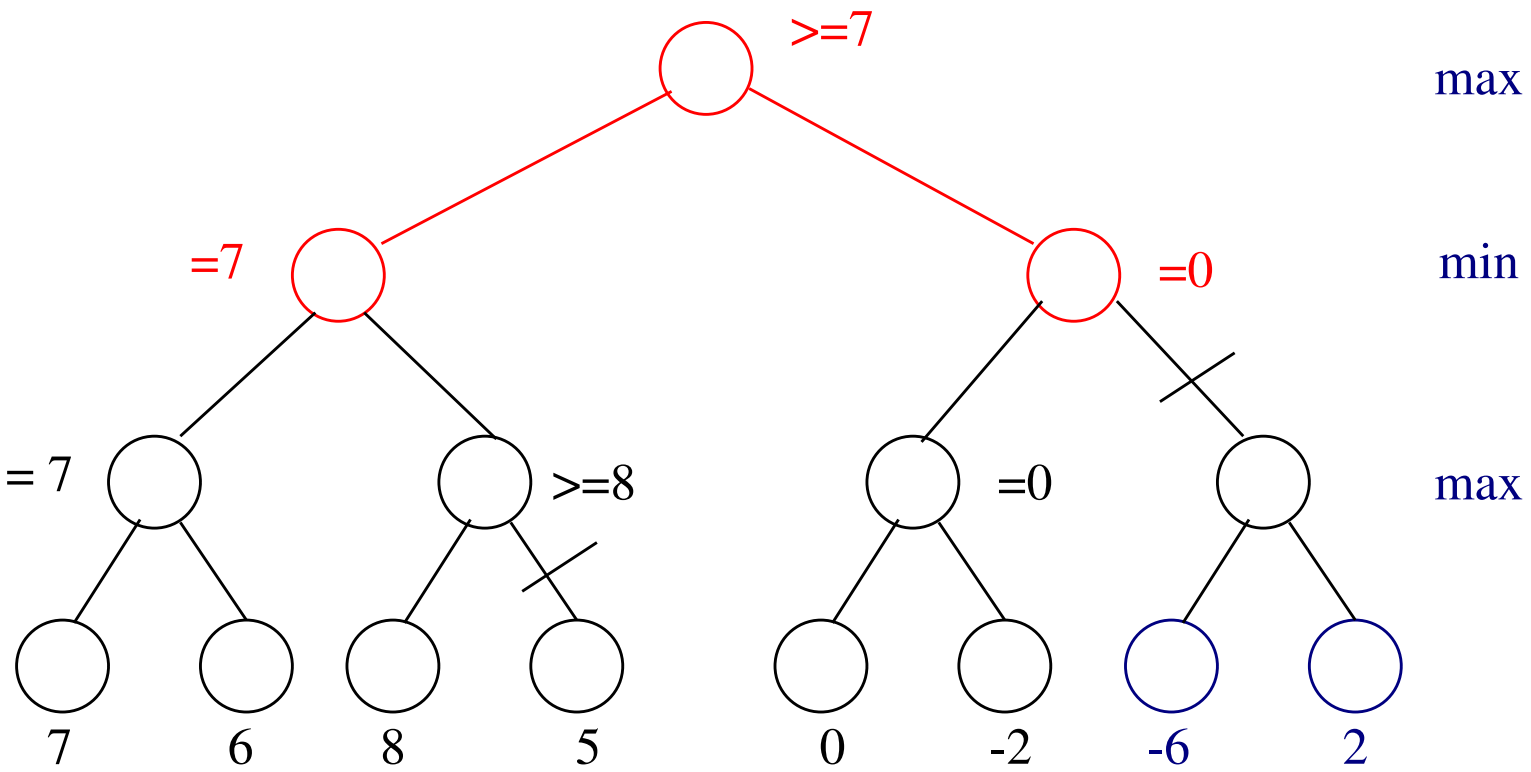
α–β cut-offs



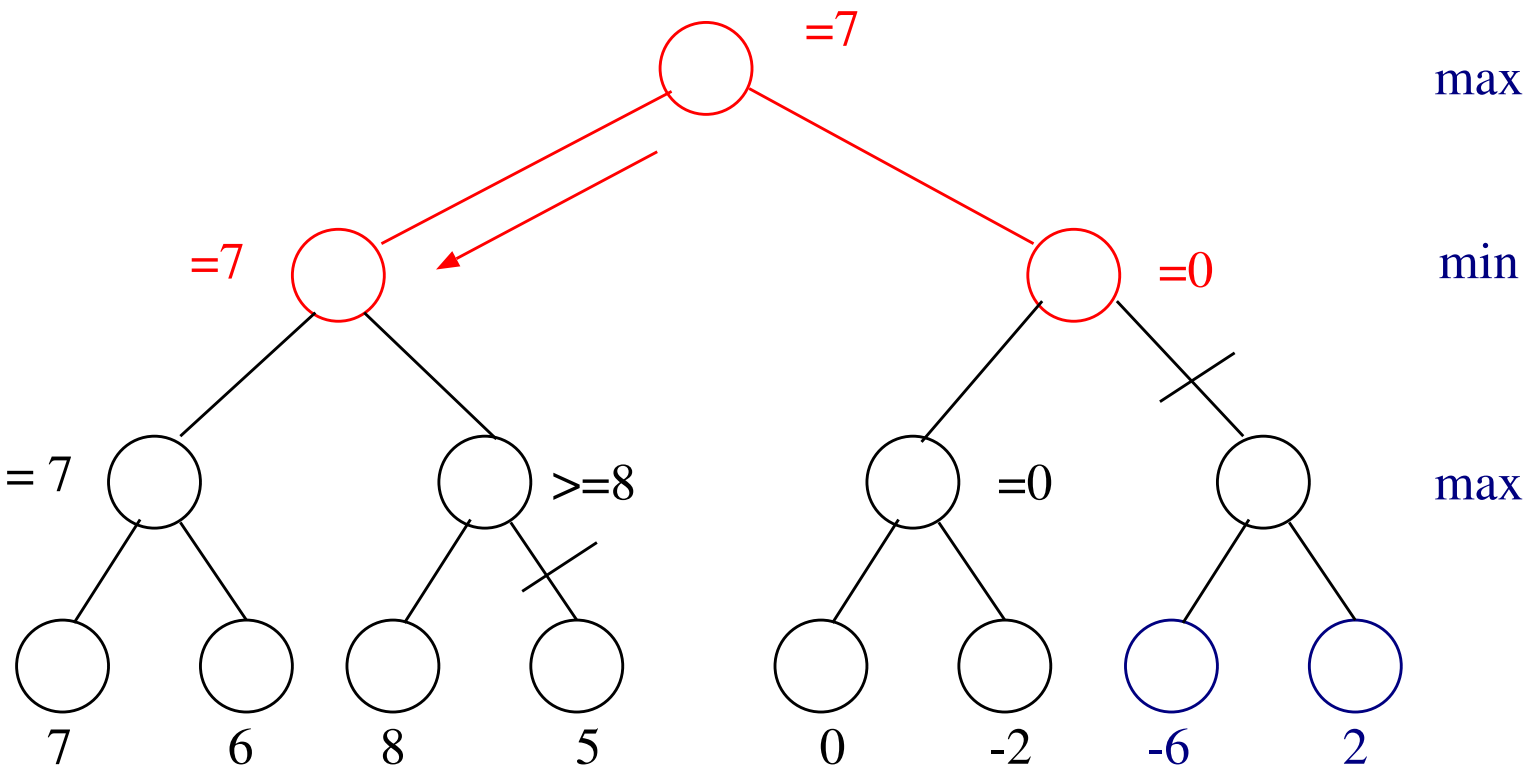
α–β cut-offs



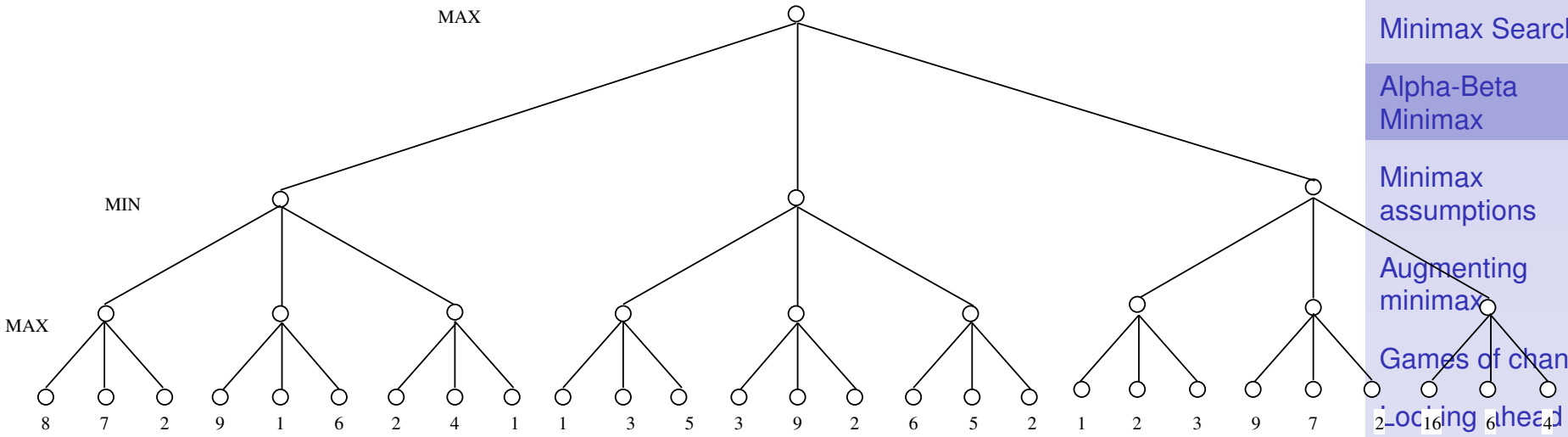
α–β cut-offs



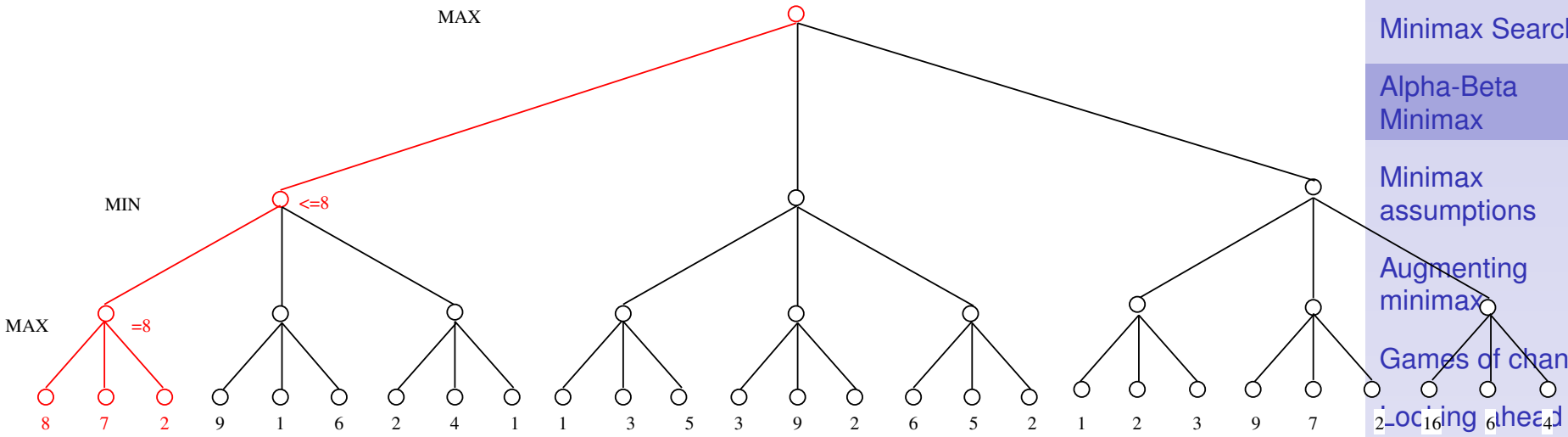
α–β cut-offs



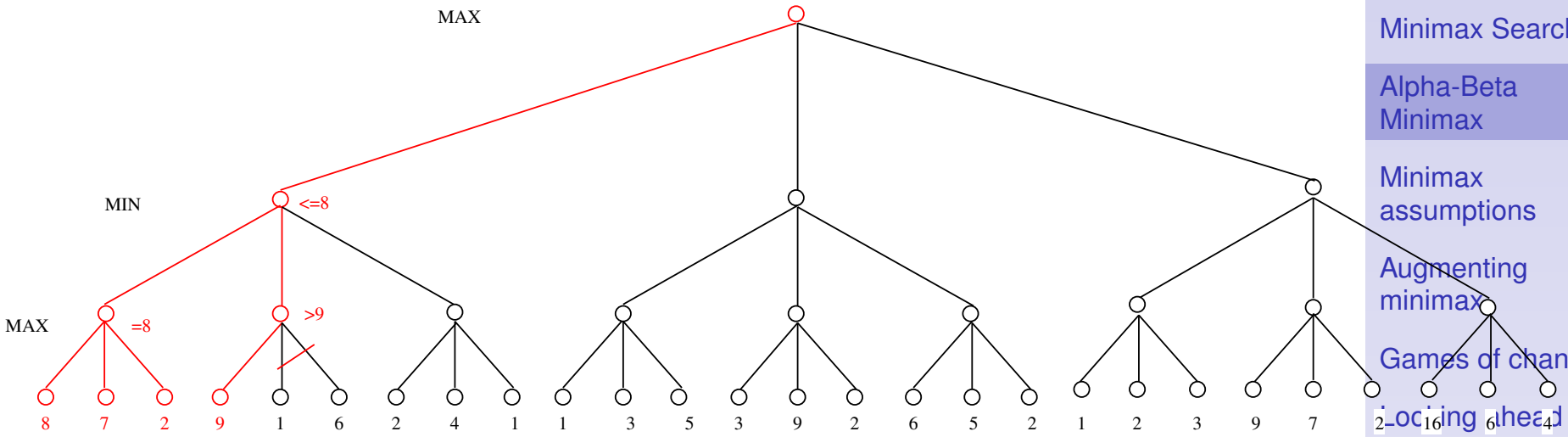
α–β cut-offs



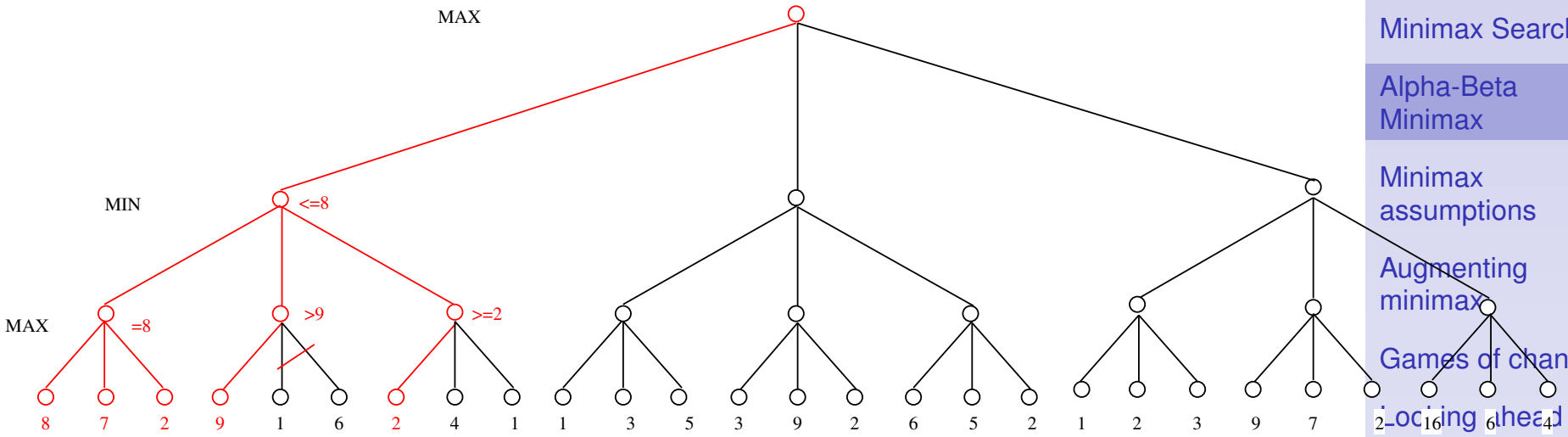
α–β cut-offs



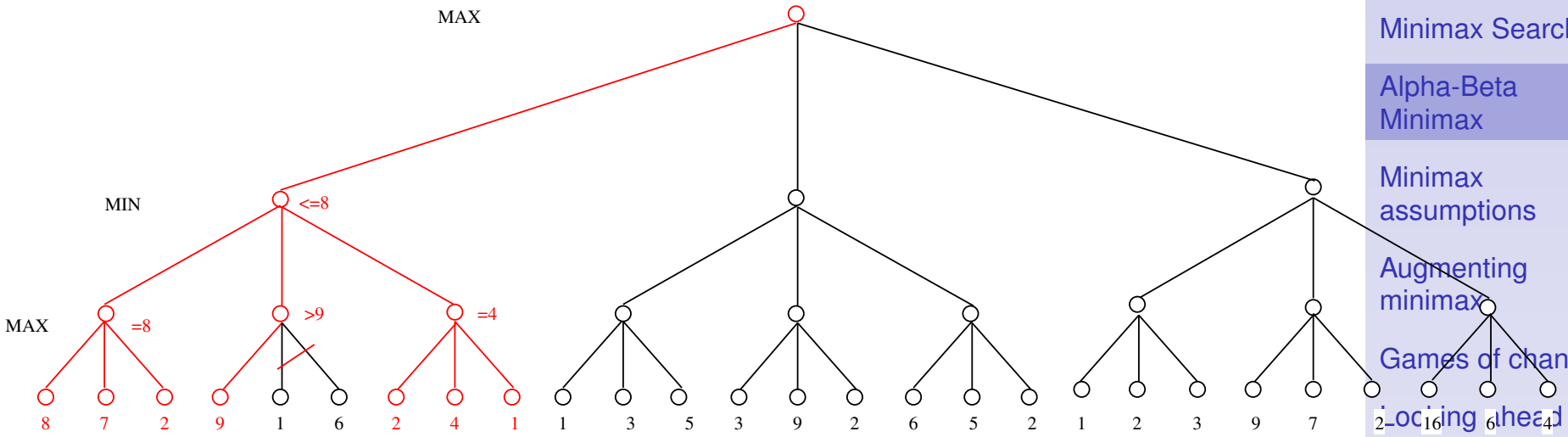
α–β cut-offs



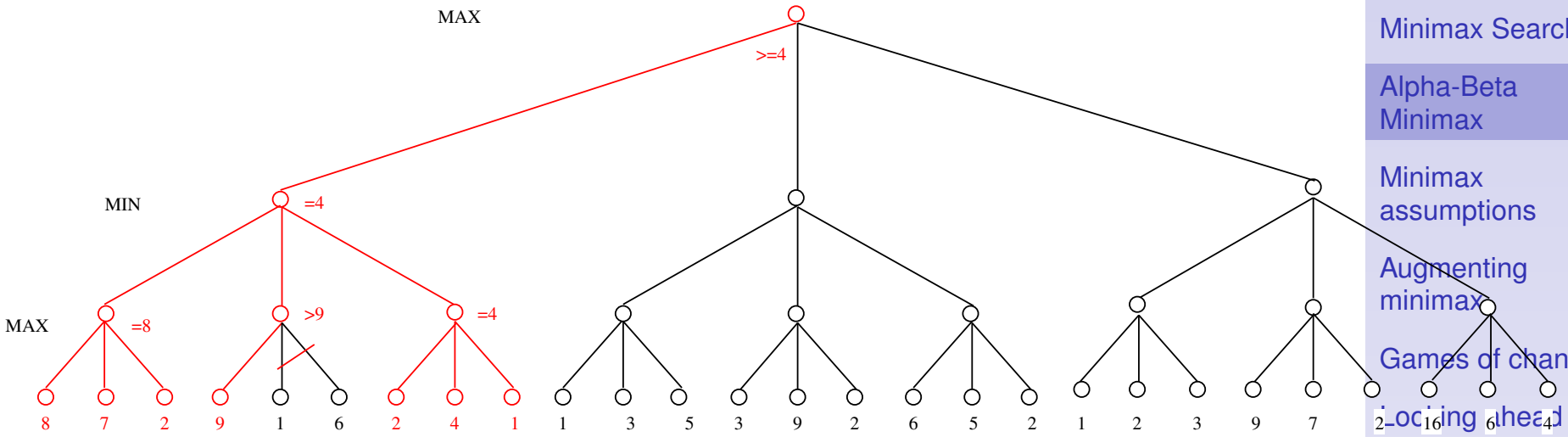
α–β cut-offs



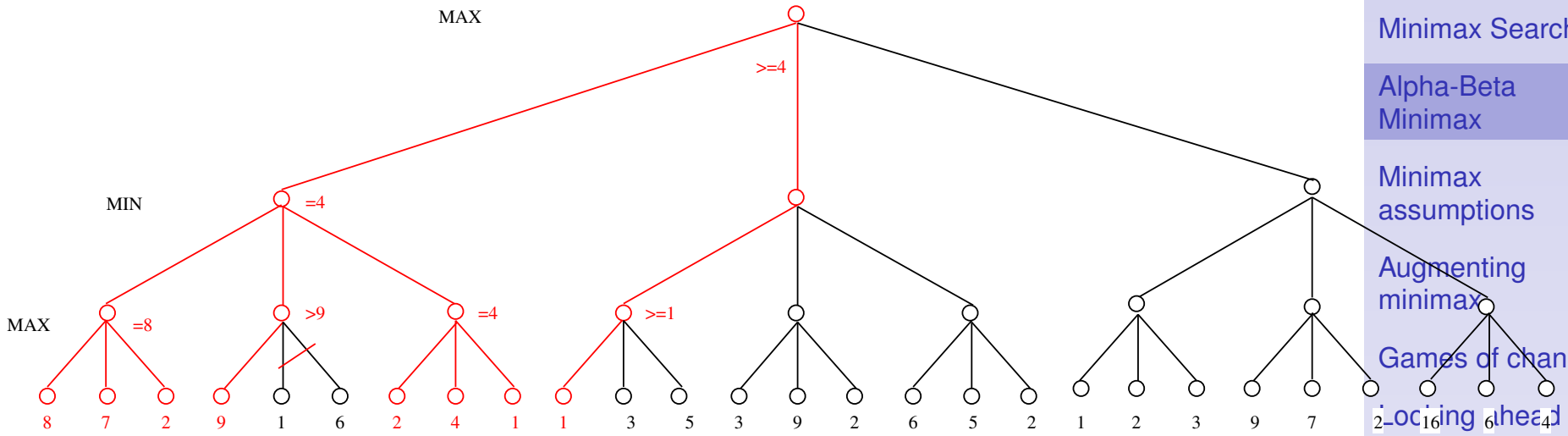
α–β cut-offs



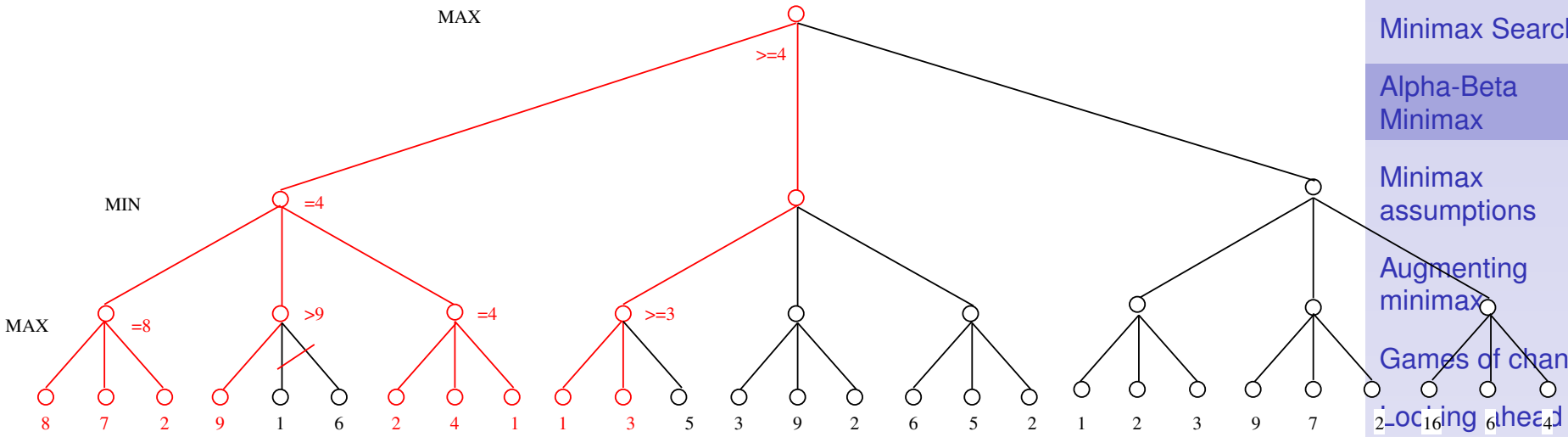
α–β cut-offs



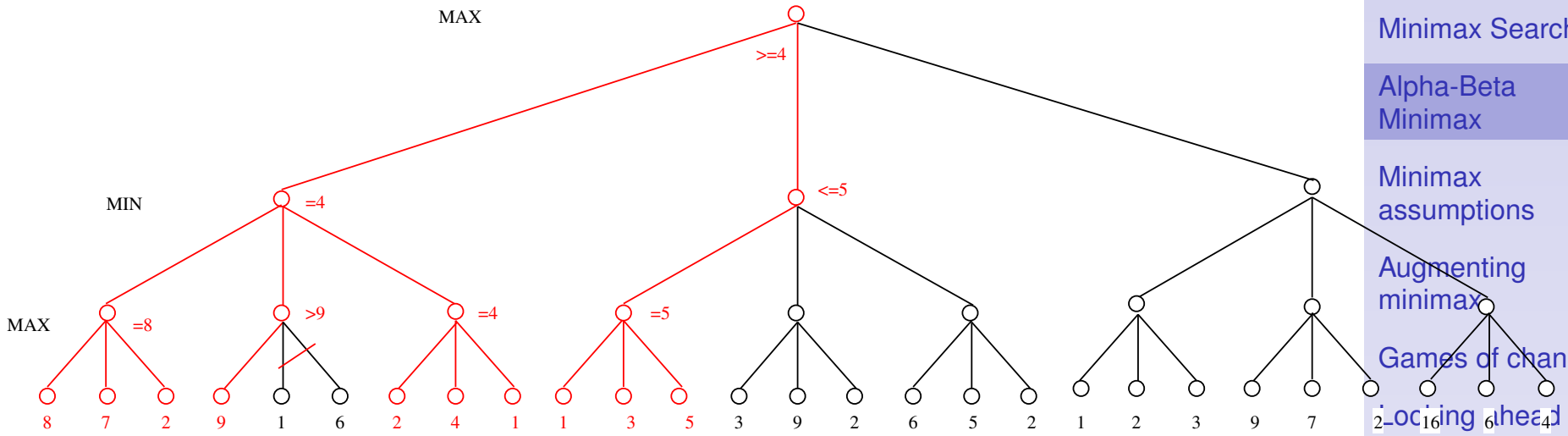
α–β cut-offs



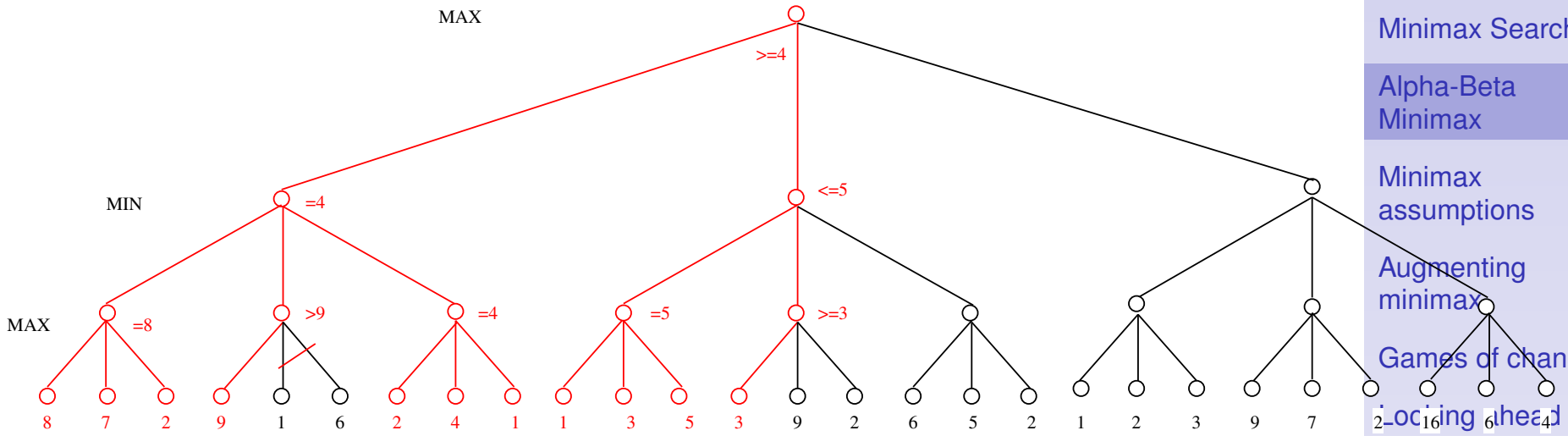
α–β cut-offs



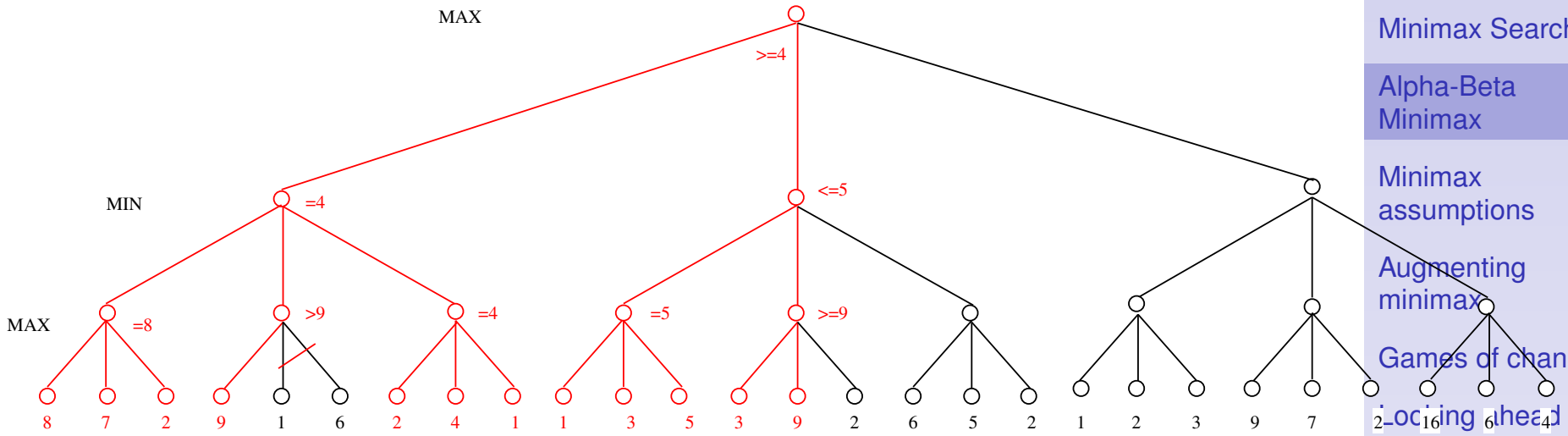
α–β cut-offs



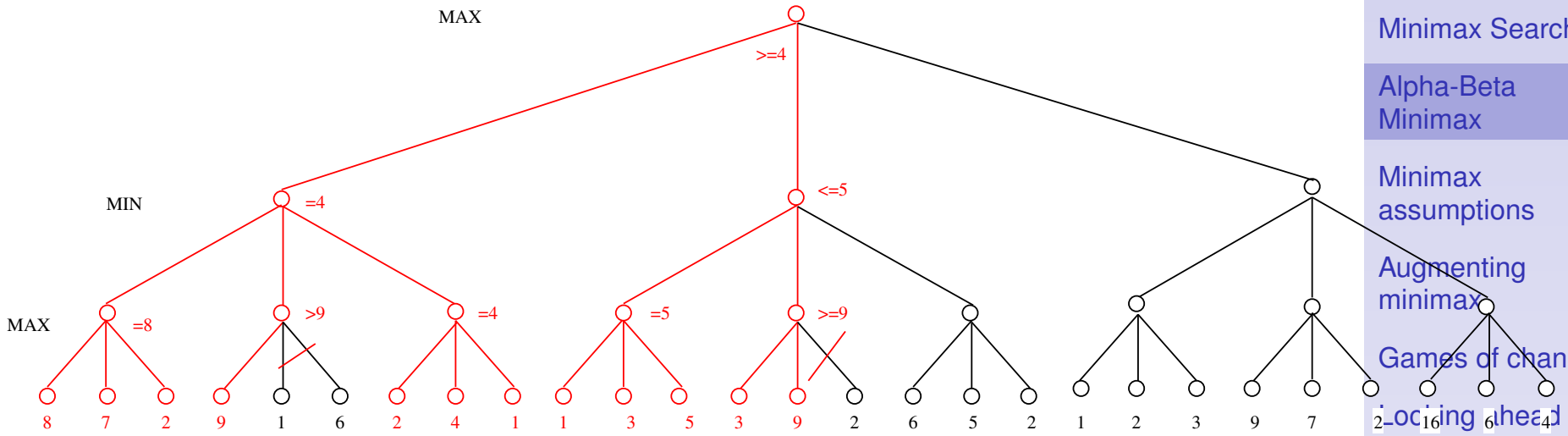
α–β cut-offs



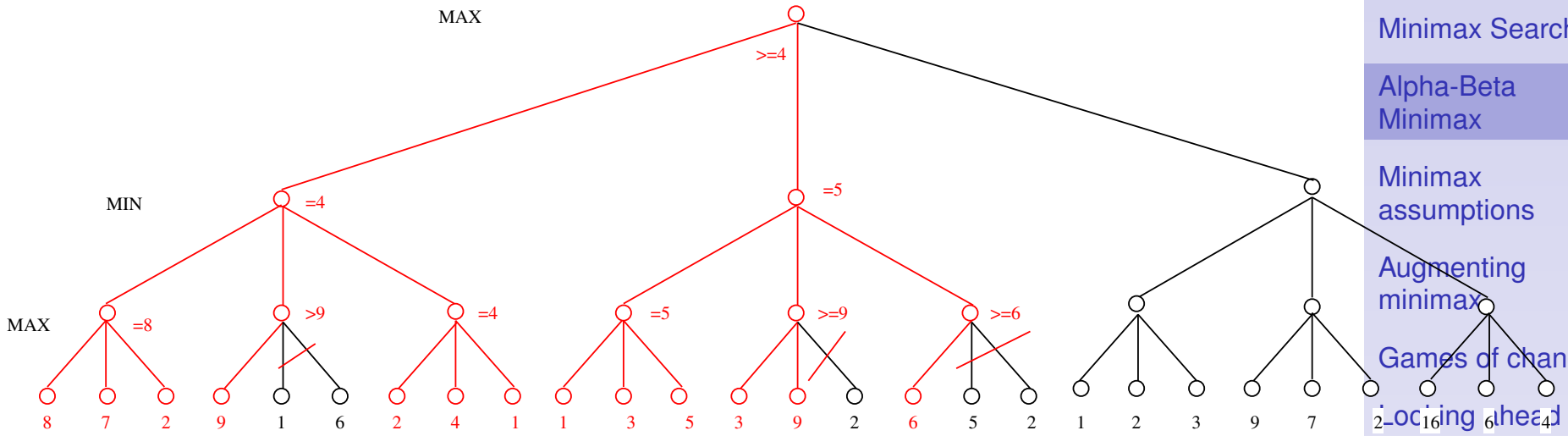
α–β cut-offs



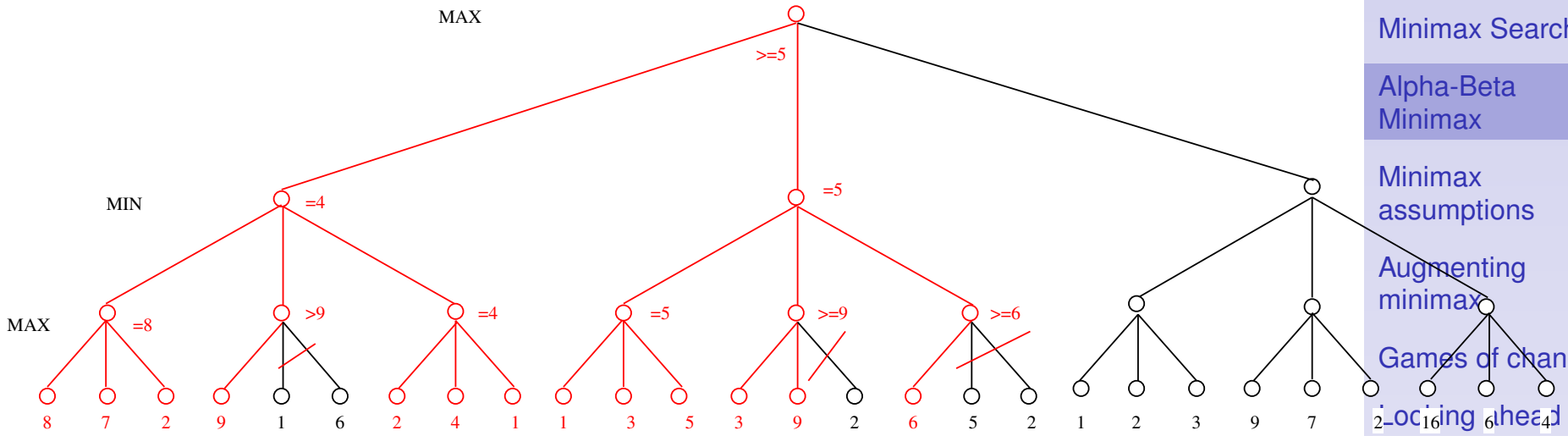
α–β cut-offs



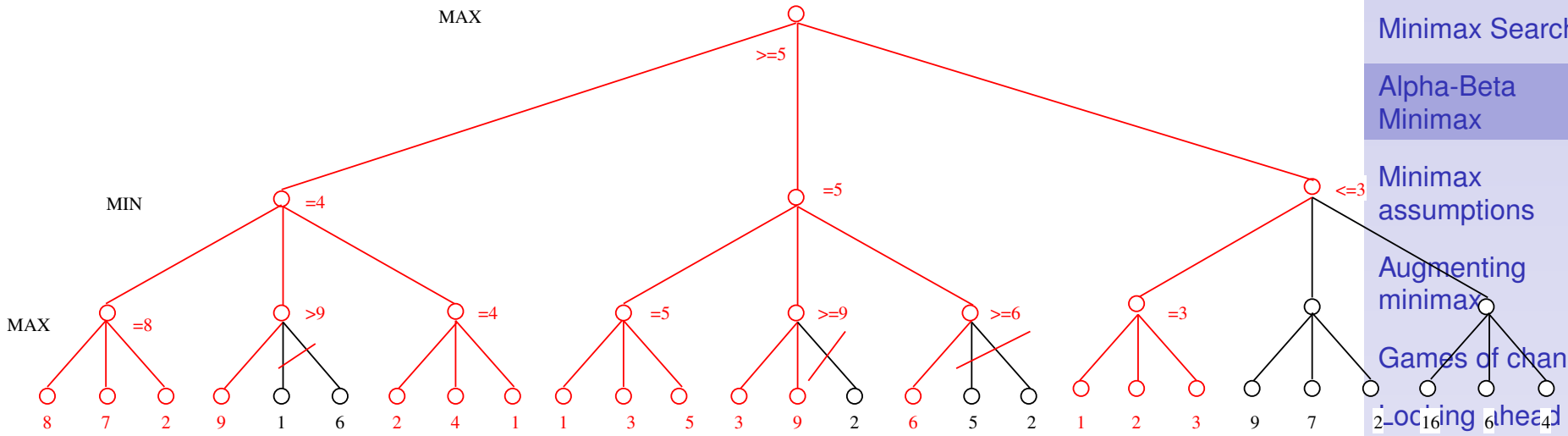
α–β cut-offs



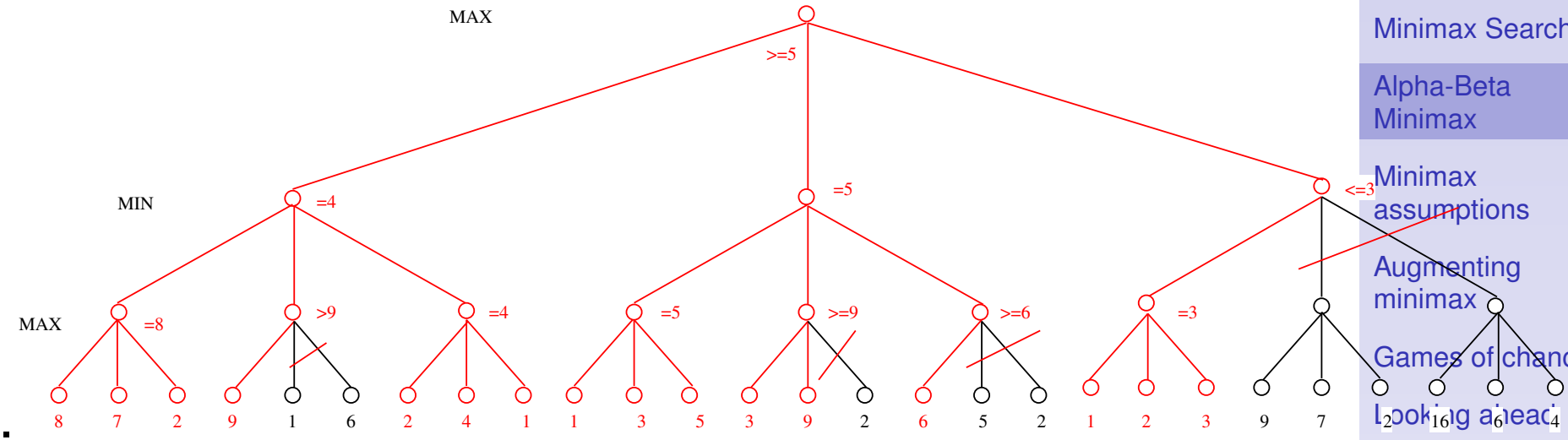
α–β cut-offs



α–β cut-offs



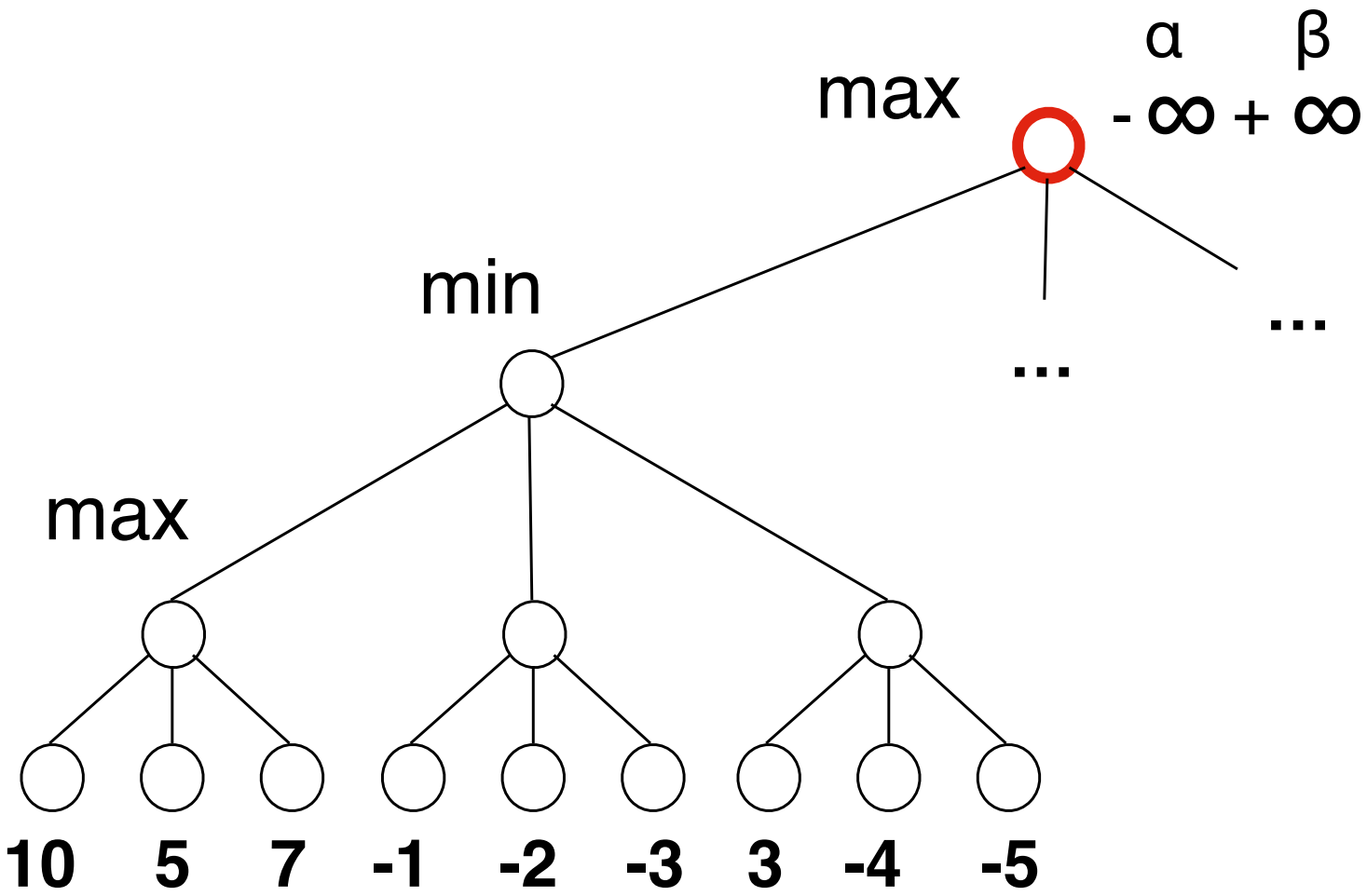
α–β cut-offs



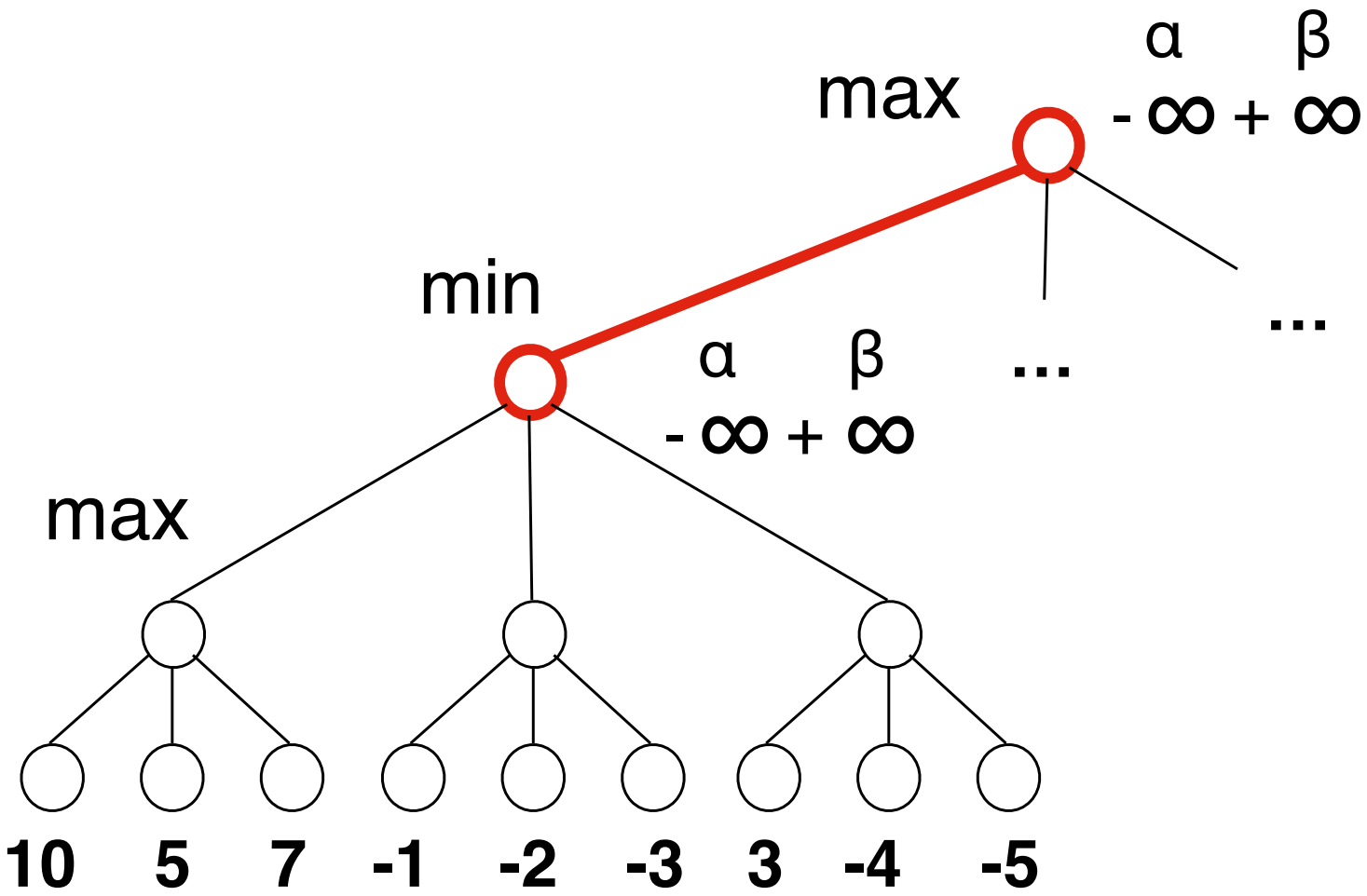
α-β cut-offs

- Keep track of two values:
  - α: Maximum value the maximizer (player) can expect
  - β: Minimum value the minimizer (opponent) can expect
- Initialize to worst case for each: $\alpha = -\infty, \beta = +\infty$
- Update:
  - Maximizer: α = max of α, value of recurring on children
  - Minimizer: β = min of β, value of recurring on children
- If at any point in the update $\beta \leq \alpha$
  - Maximizer: then opponent has a better move than this
  - Minimizer: then player has a better move than this
  - Either way, no point continuing: *prune*

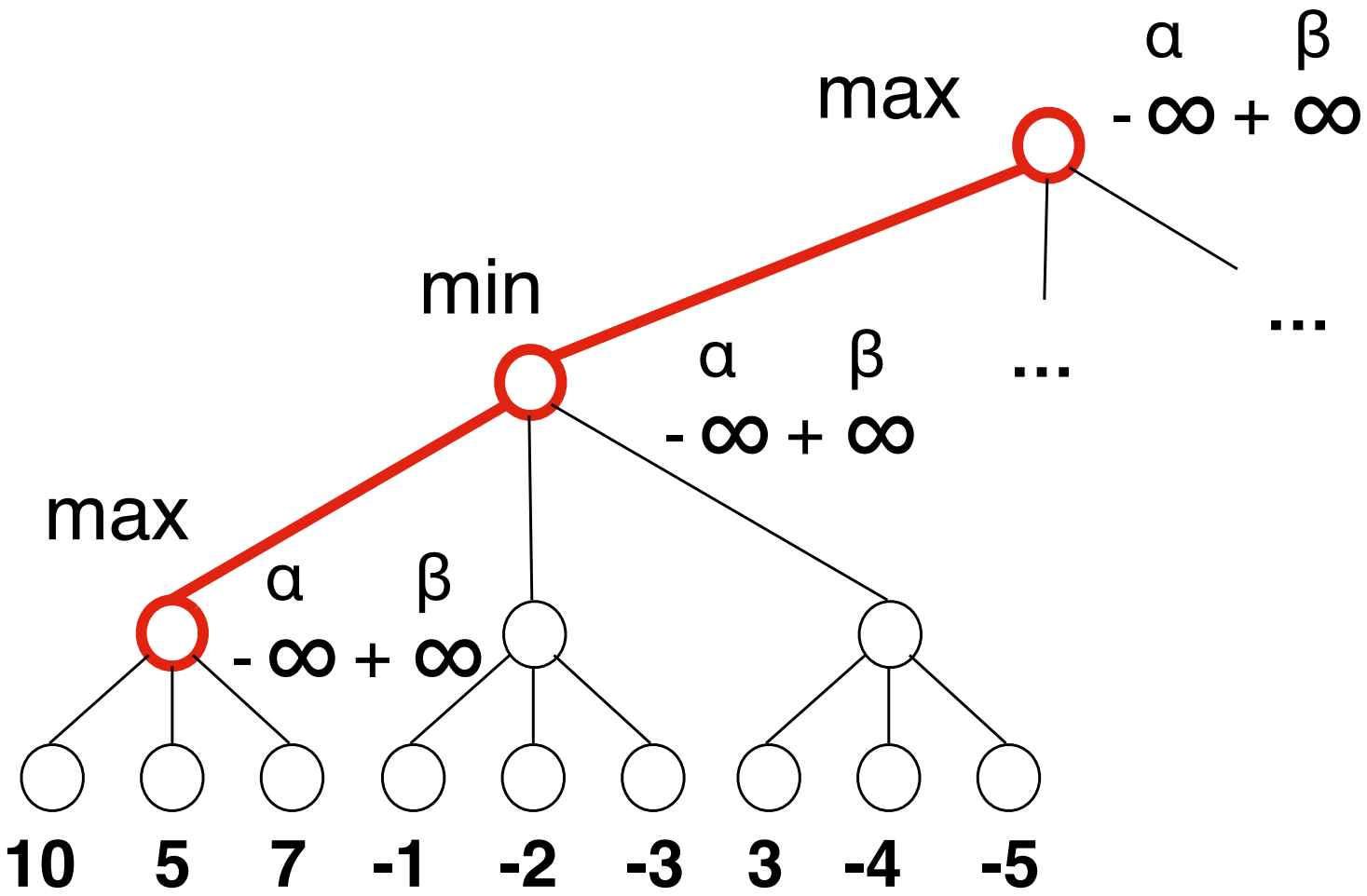
Example



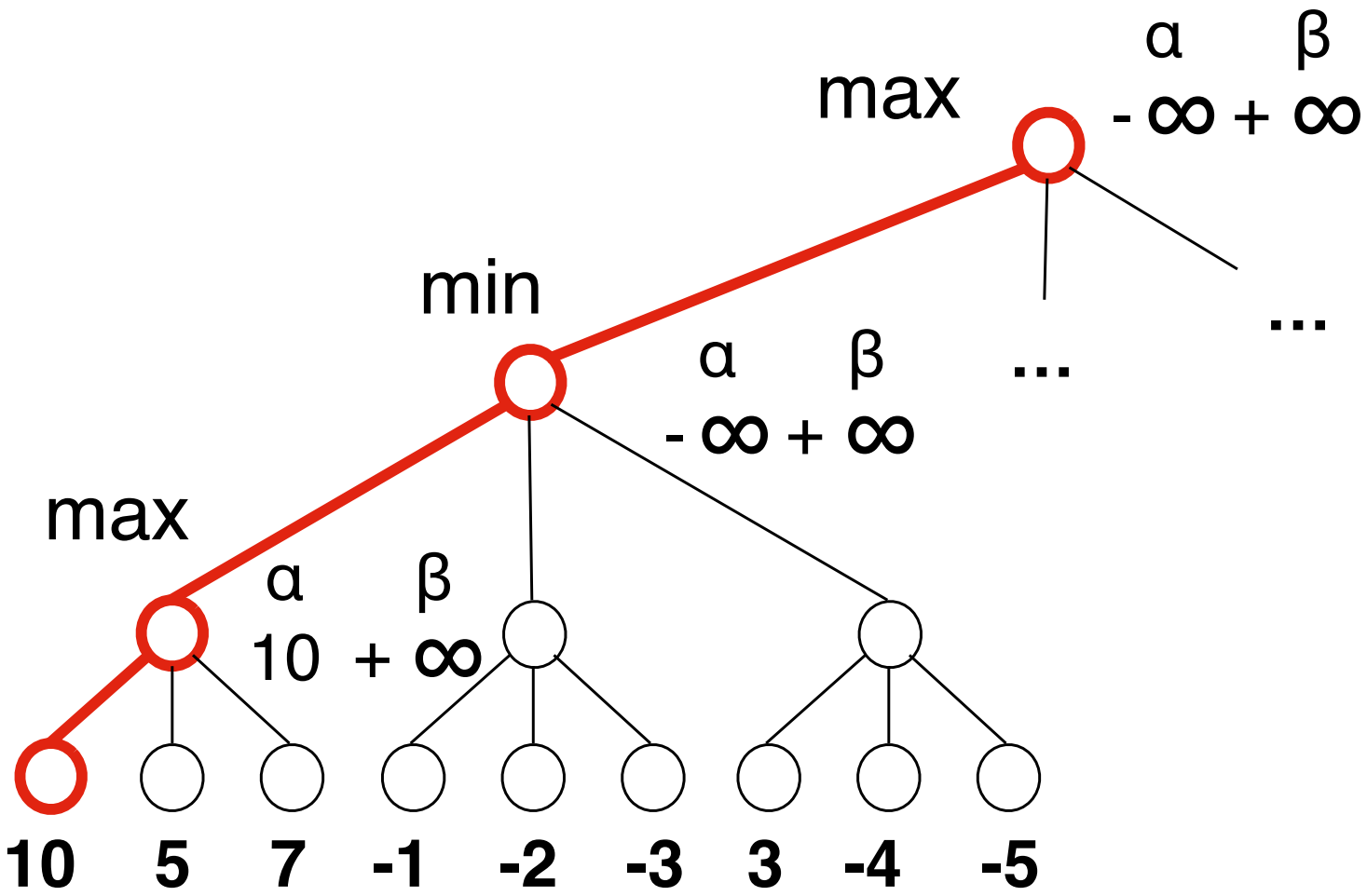
Example



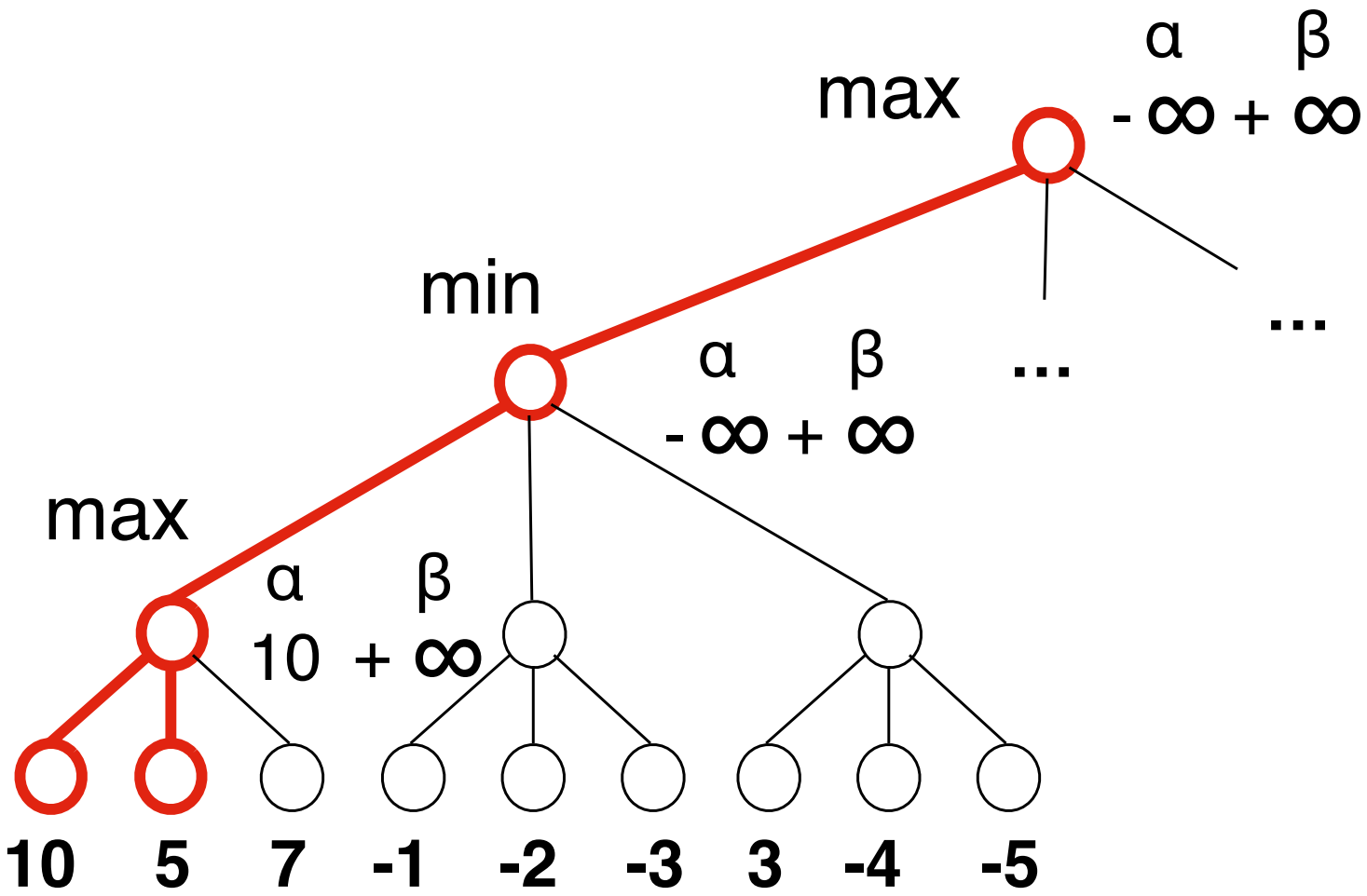
Example



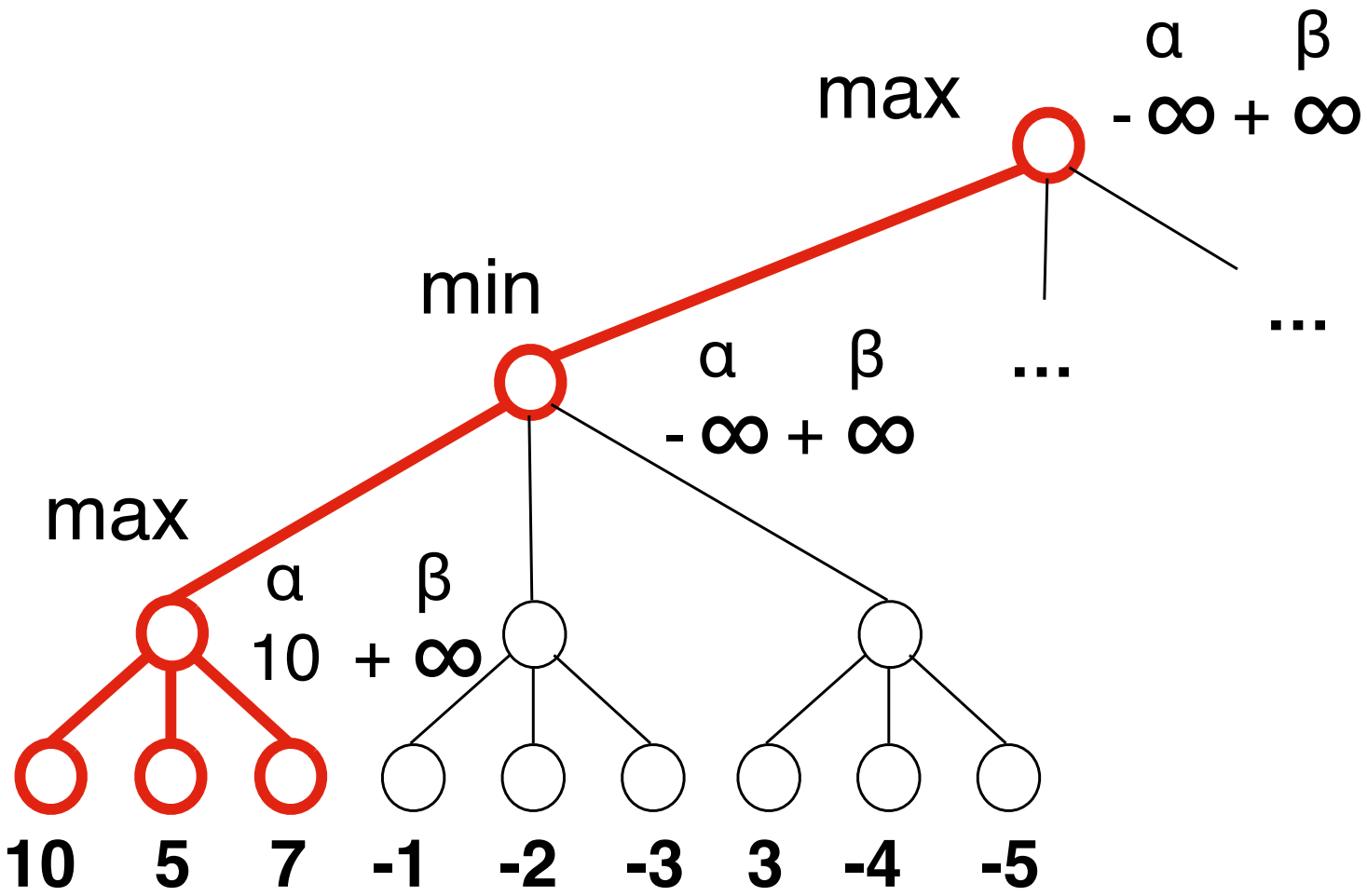
Example



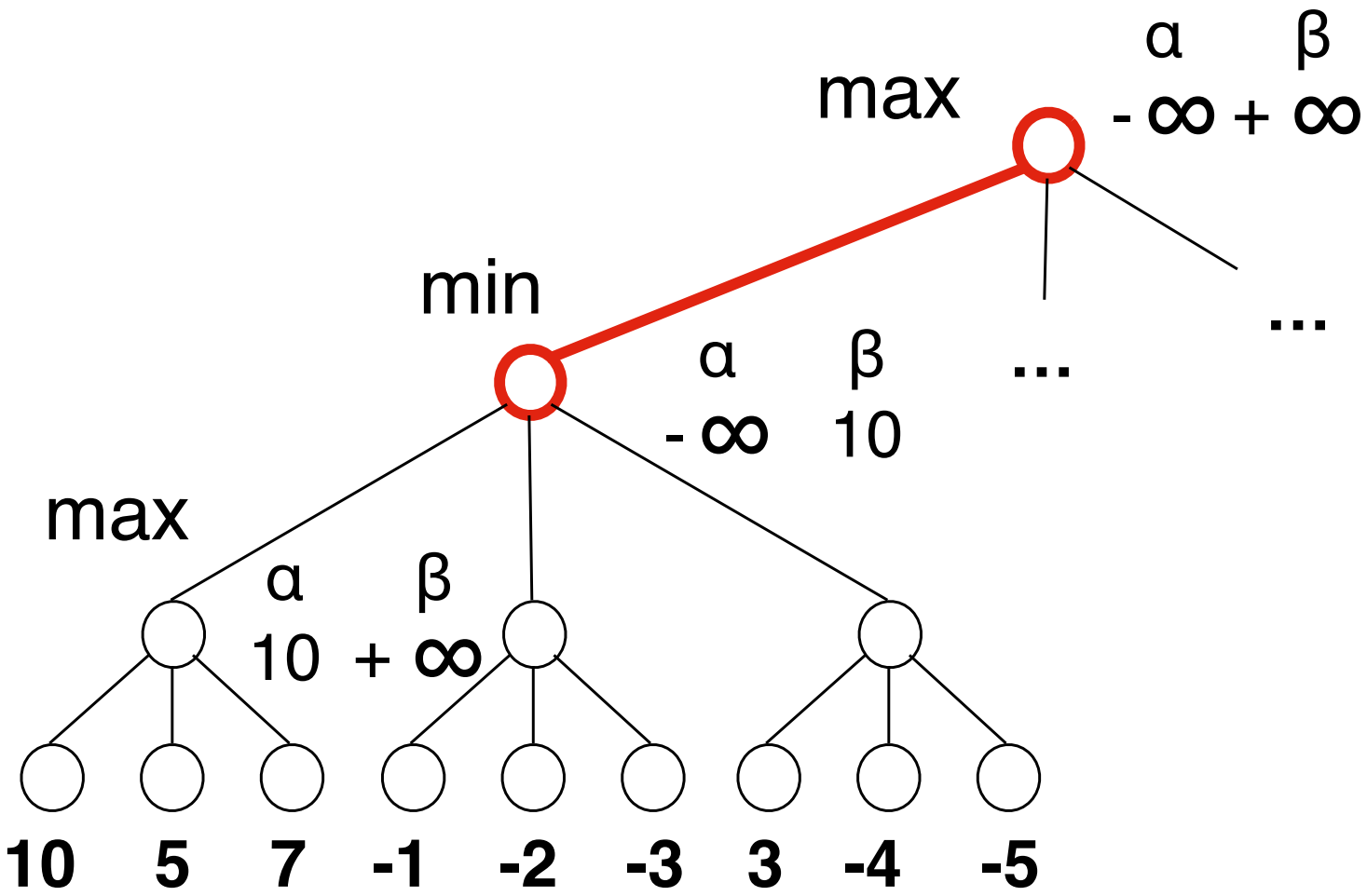
Example



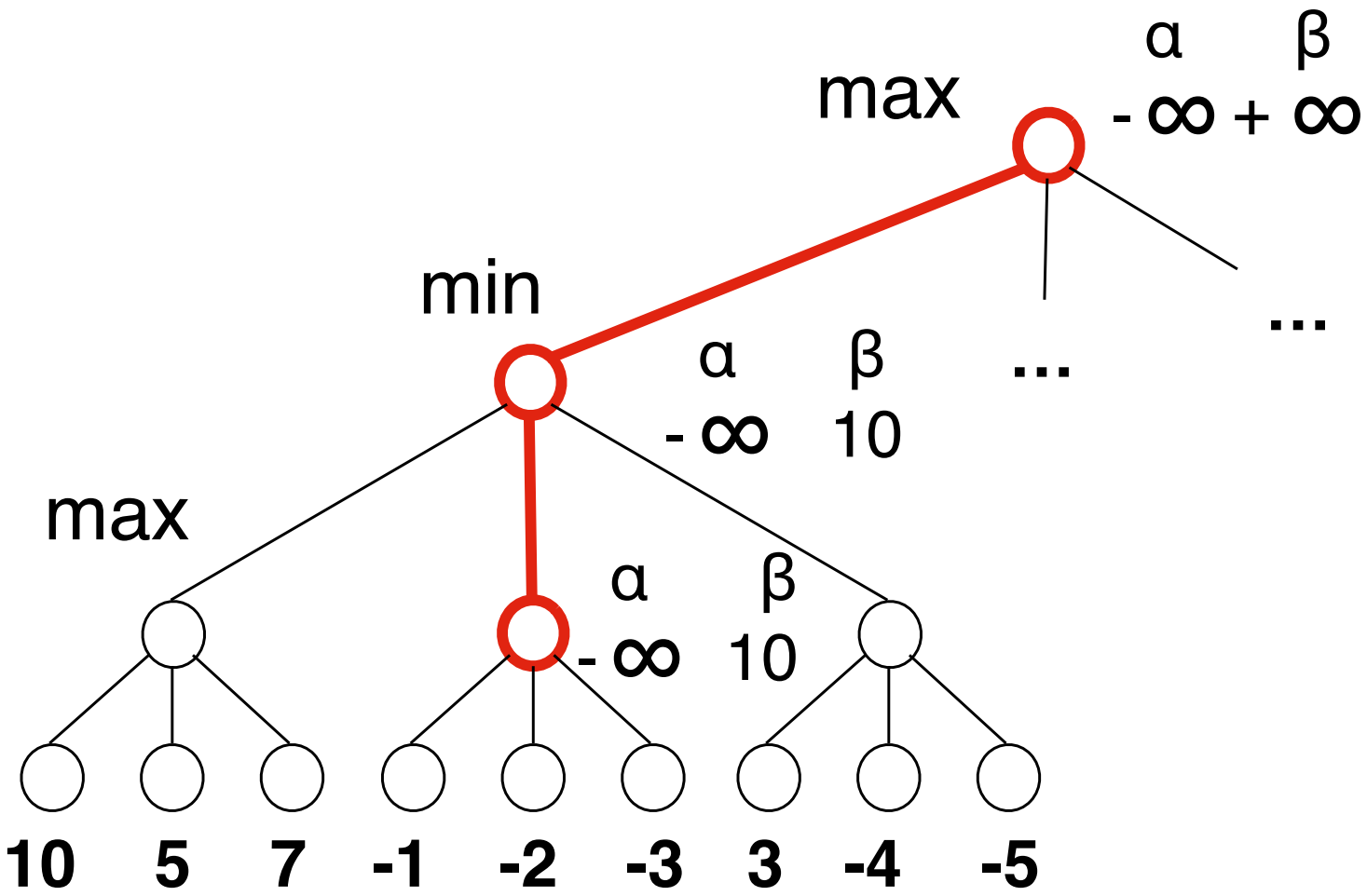
Example



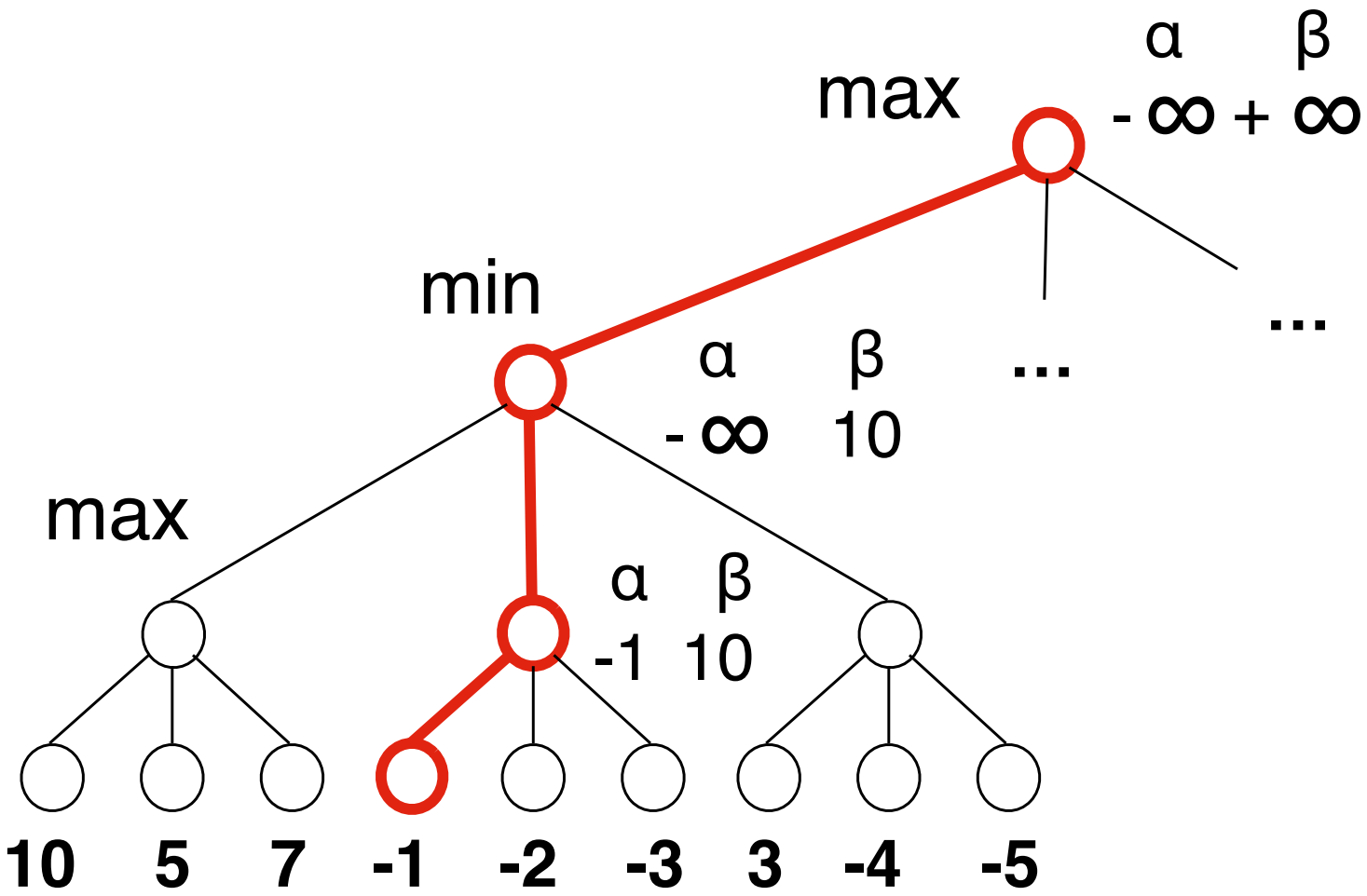
Example



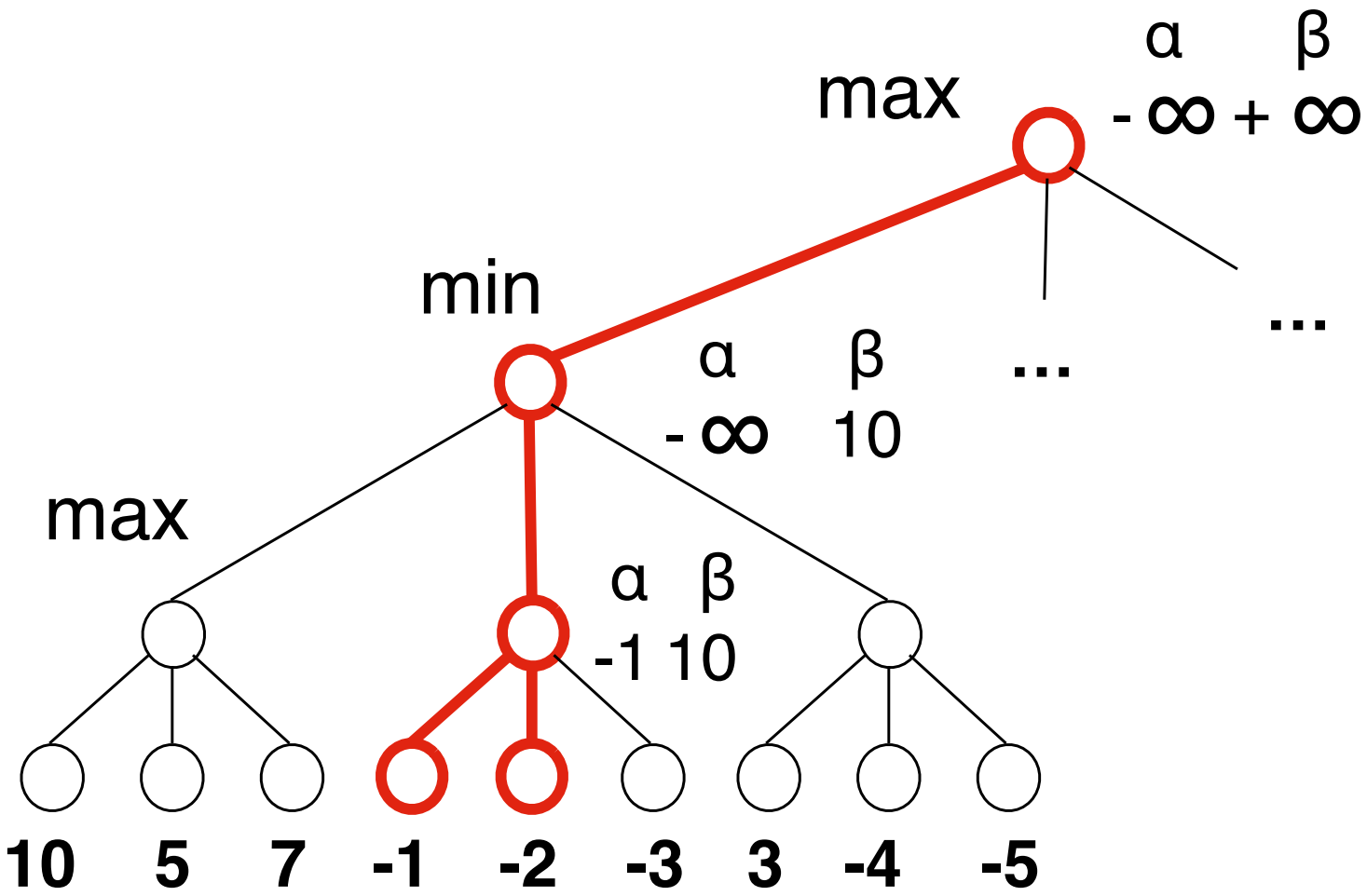
Example



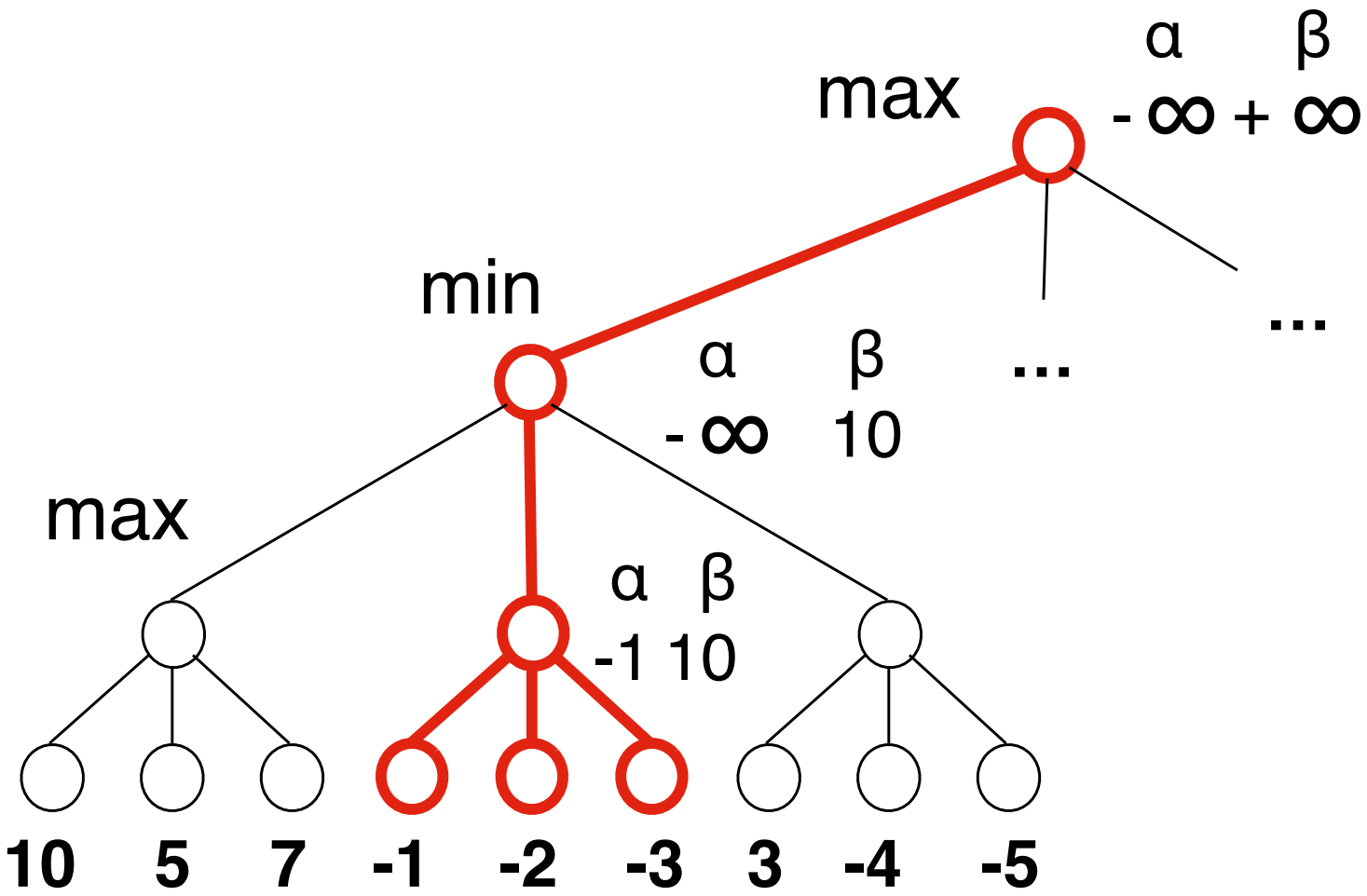
Example



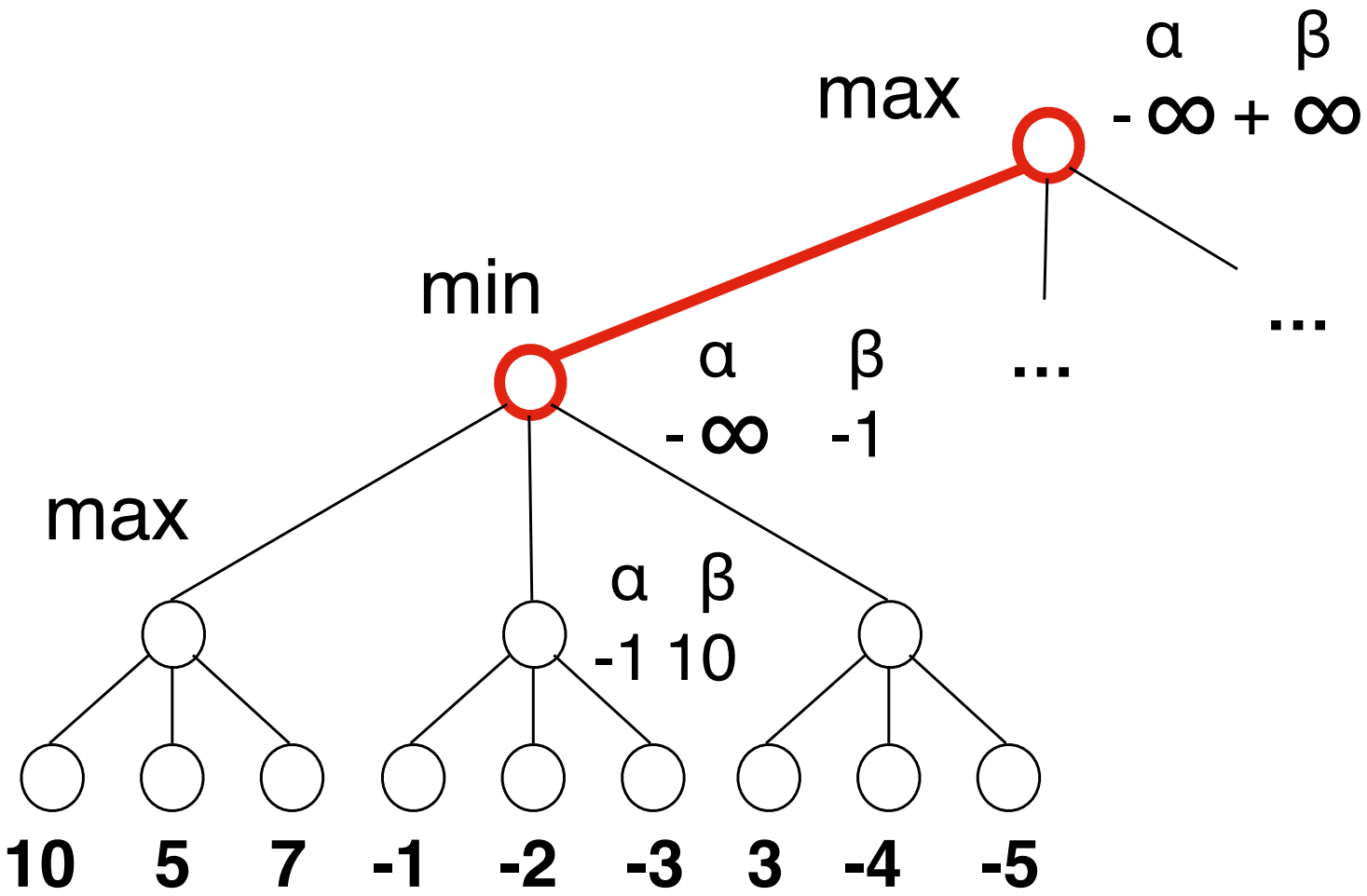
Example



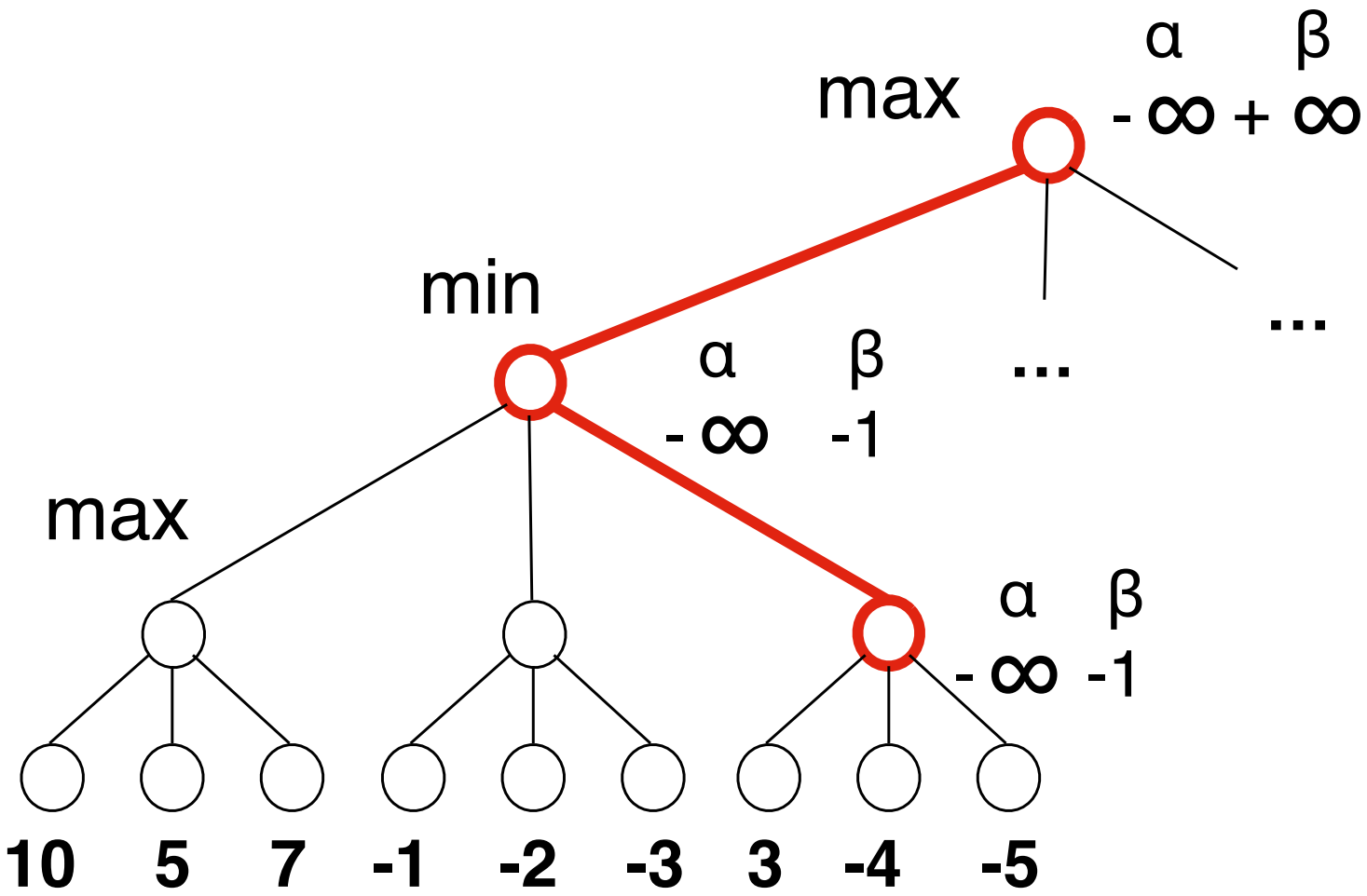
Example



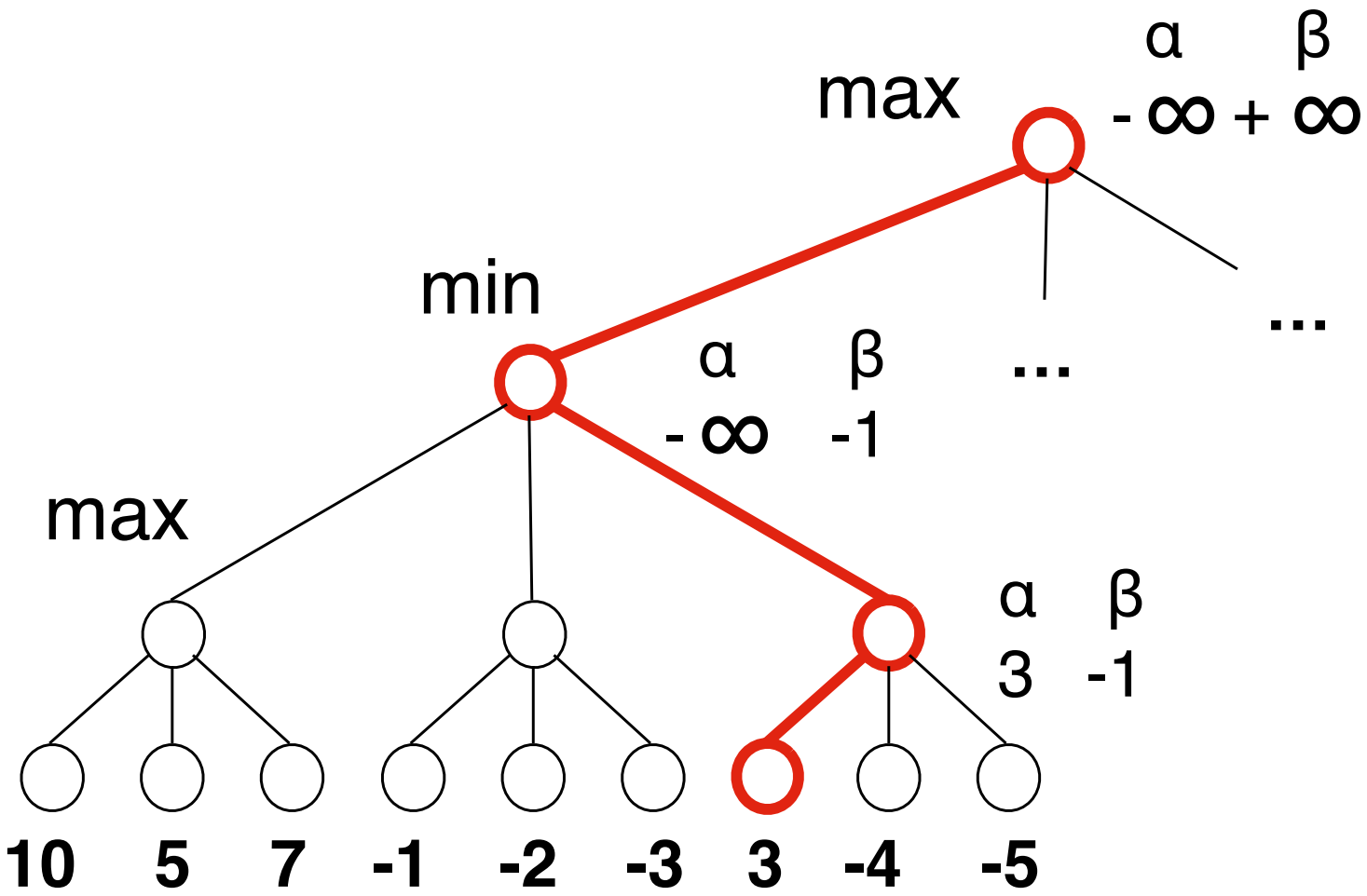
Example



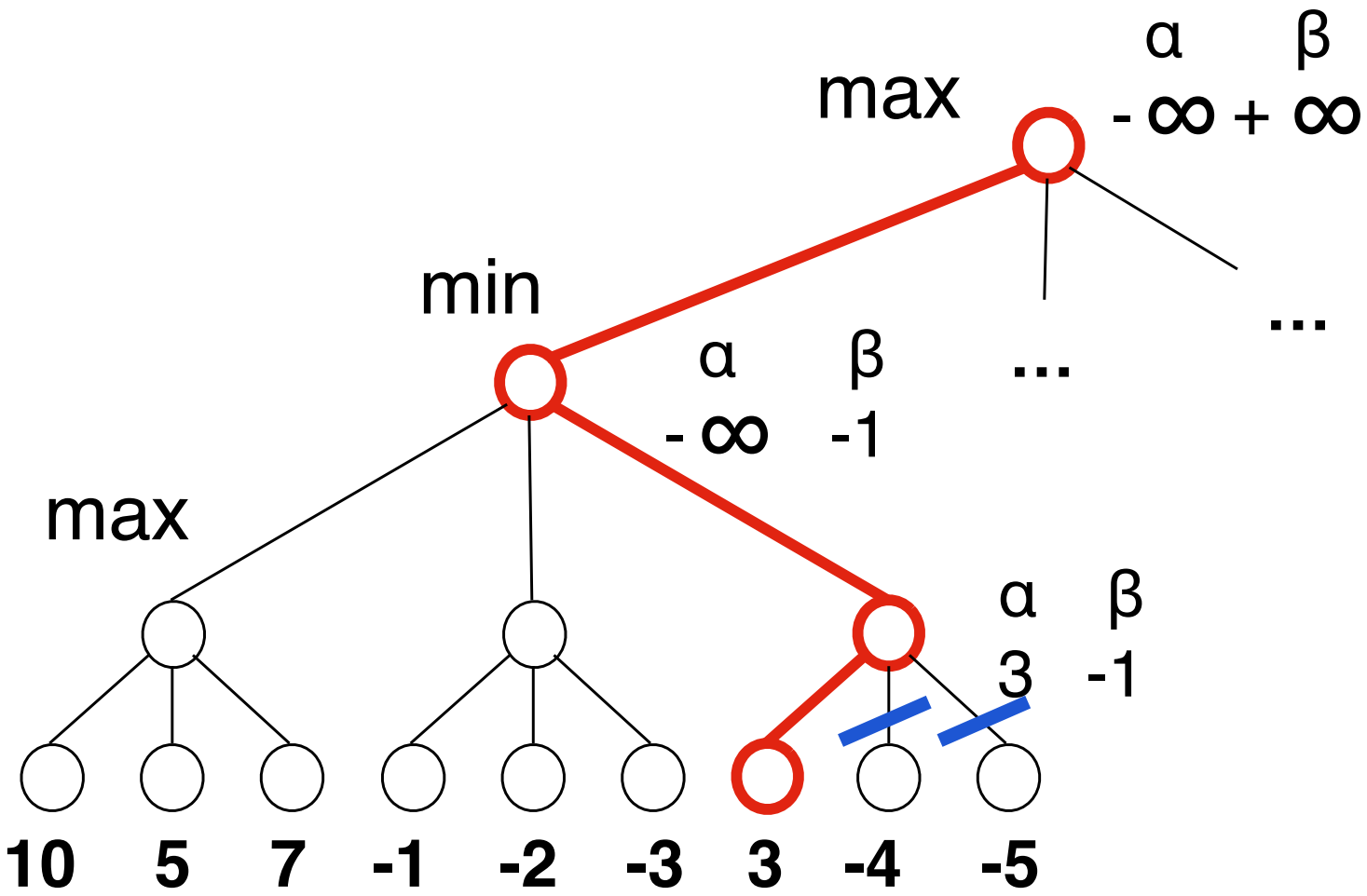
Example



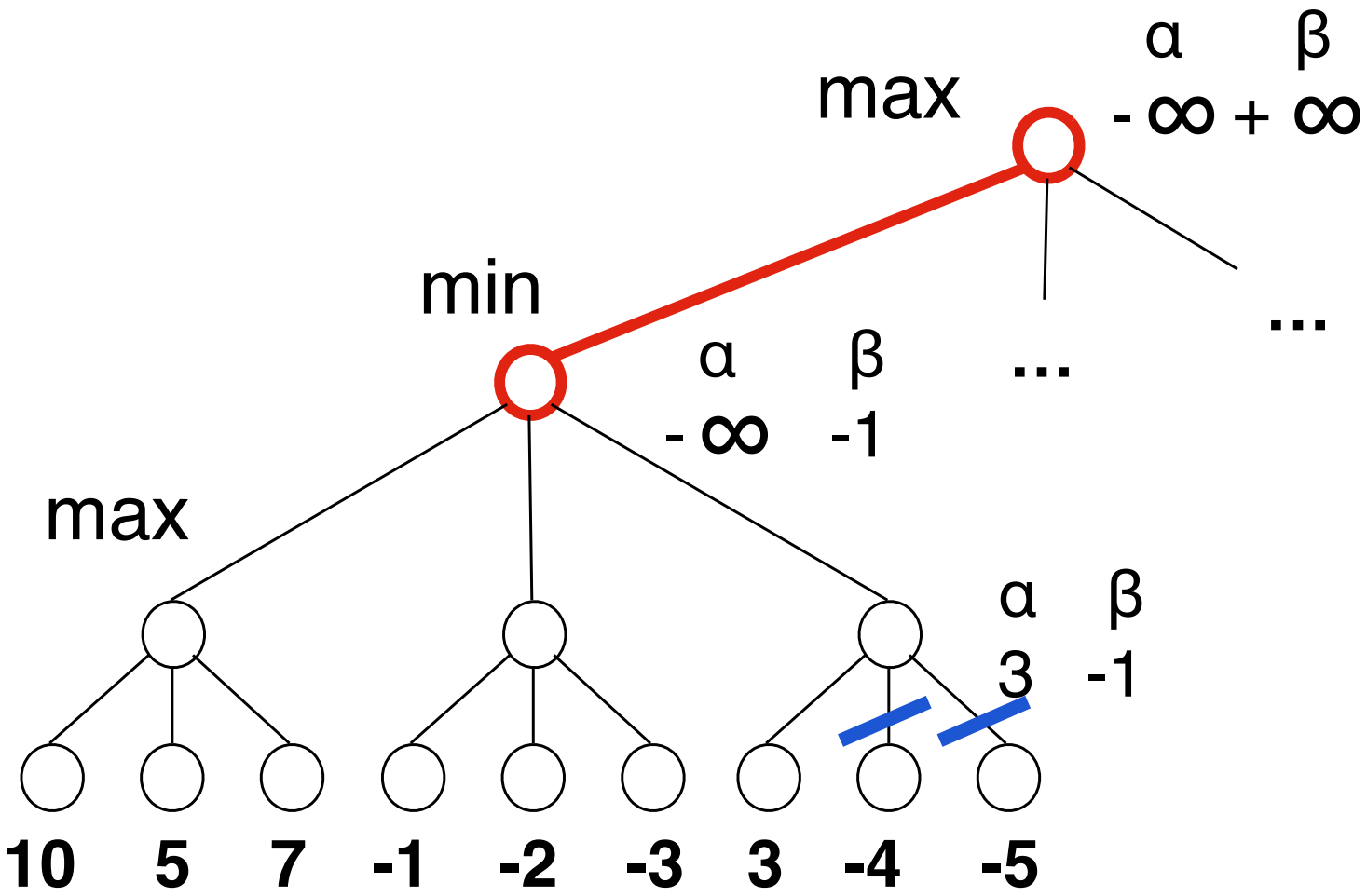
Example



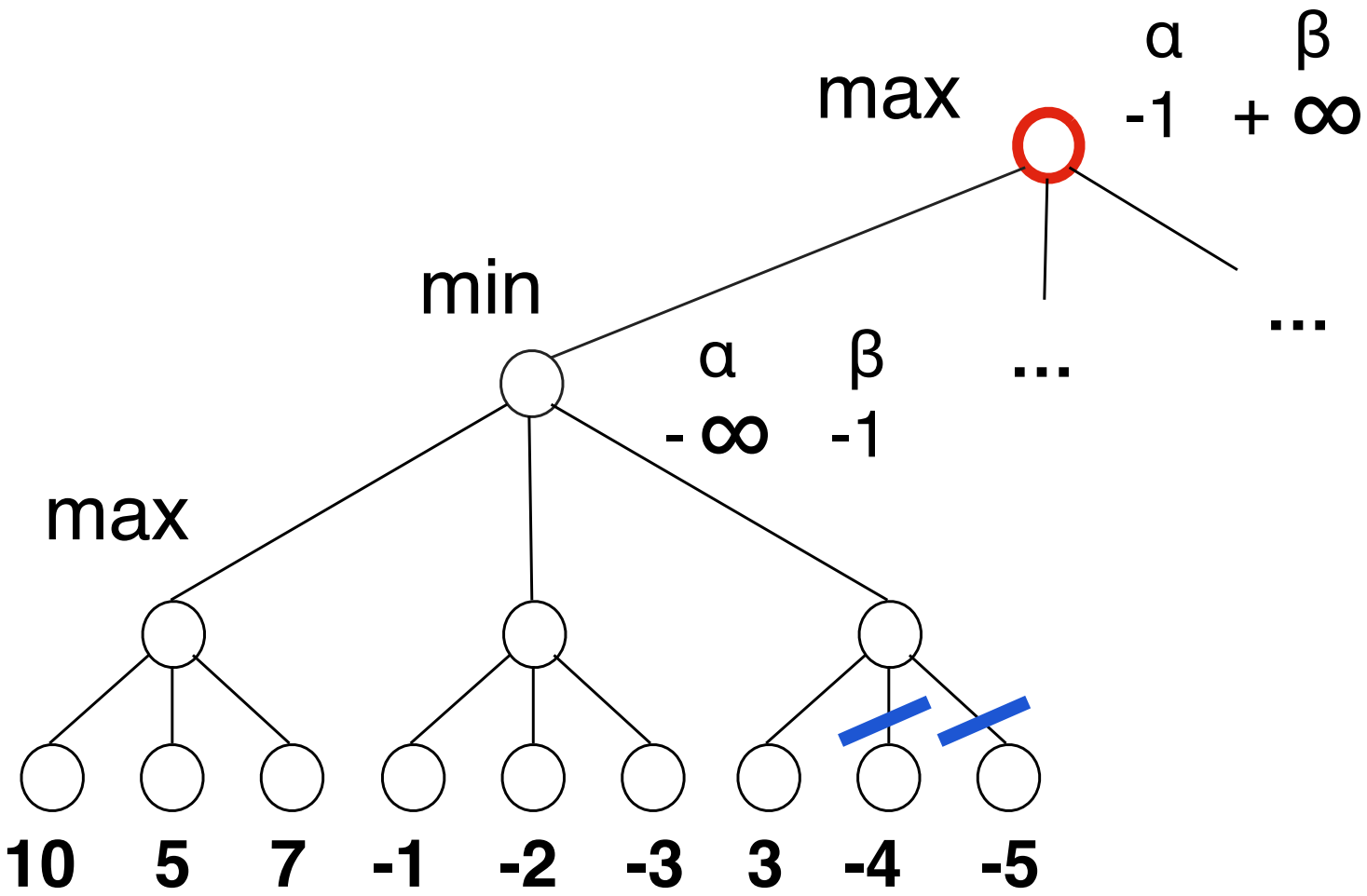
Example



Example



Example



α–β minimax

[From Russell & Norvig]

function ALPHA-BETA-DECISION($state$) **returns** an action
  return the a in ACTIONS($state$) maximizing MIN-VALUE(RESULT($a, state$))

---

function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  inputs: $state$, current state in game
    α, the value of the best alternative for MAX along the path to $state$
    β, the value of the best alternative for MIN along the path to $state$
  if TERMINAL-TEST($state$) **then return** UTILITY($state$)
  $v \leftarrow -\infty$
  for a, s in SUCCESSORS($state$) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(s, α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v

---

function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  same as MAX-VALUE but with roles of α, β reversed

Effectiveness of α–β pruning

- Minimax: $\mathcal{O}(b^d)$ worst case
- α–β minimax: same w.c.
- Best case:
  - If nodes are in best-to-worst order
  - $\Rightarrow \mathcal{O}(b^{d/2})$
  - What does this mean?

Effectiveness of α–β pruning

- Minimax: $\mathcal{O}(b^d)$ worst case
- α–β minimax: same w.c.
- Best case:
  - If nodes are in best-to-worst order
  - $\Rightarrow \mathcal{O}(b^{d/2})$
  - What does this mean?
  - Effective branching factor: $(b^*)^d$, $\therefore b^* = \sqrt{b}$

Effectiveness of α–β pruning

- Minimax: $\mathcal{O}(b^d)$ worst case
- α–β minimax: same w.c.
- Best case:
  - If nodes are in best-to-worst order
  - $\Rightarrow \mathcal{O}(b^{d/2})$
  - What does this mean?
  - Effective branching factor: $(b^*)^d$, $\therefore b^* = \sqrt{b}$
  - Also: search twice as deep, same amount of time

Effectiveness of α–β pruning

- Minimax: $\mathcal{O}(b^d)$ worst case
- α–β minimax: same w.c.
- Best case:
  - If nodes are in best-to-worst order
  - $\Rightarrow \mathcal{O}(b^{d/2})$
  - What does this mean?
  - Effective branching factor: $(b^*)^d$, $\therefore b^* = \sqrt{b}$
  - Also: search twice as deep, same amount of time
- So what is average cost?

Effectiveness of α–β pruning

- Minimax: $\mathcal{O}(b^d)$ worst case
- α–β minimax: same w.c.
- Best case:
  - If nodes are in best-to-worst order
  - $\Rightarrow \mathcal{O}(b^{d/2})$
  - What does this mean?
  - Effective branching factor: $(b^*)^d$, $\therefore b^* = \sqrt{b}$
  - Also: search twice as deep, same amount of time
- So what is average cost?
  - Who knows? But. . .

Effectiveness of α–β pruning

- Minimax: $\mathcal{O}(b^d)$ worst case
- α–β minimax: same w.c.
- Best case:
  - If nodes are in best-to-worst order
  - $\Rightarrow \mathcal{O}(b^{d/2})$
  - What does this mean?
  - Effective branching factor: $(b^*)^d$, $\therefore b^* = \sqrt{b}$
  - Also: search twice as deep, same amount of time
- So what is average cost?
  - Who knows? But. . .
  - Closer to b.c. than w.c.: “nature proving unusually beneficient” (Winston)

Minimax assumptions

Assumptions of (α–β) minimax

- Well-defined operators
- Zero-sum, symmetric: good for me, bad for you by same amount
- Opponent is rational, infallible
- Static evaluation is accurate

Augmenting minimax

Handling “horizon” effect

- *Horizon effect:*
  - Static eval at depth $d = v$
  - But just beyond d, SE $\Rightarrow v' \gg v$ or $v' \ll v$
- How to handle?
  - *Wait for quiescence*
    - If notice SE oscillating as near depth. . .
    - . . . may be in middle of move sequence that takes a loss first (e.g., piece exchange)
    - Continue searching a little more – maybe will stabilize
  - *Secondary search:* choose most promising node, continue search

Book moves

- Maybe have standard “canned” move sequences
- E.g., standard openings in chess
- Avoids/short-circuits search entirely early/late in process

Risk in face of loss

- What if all moves $\Rightarrow$ loss (or very poor SE)?
- Pick a move that *could* result in win. . .
- . . . and hope opponent makes a mistake

Iterative deepening

- Problem:
  - Suppose we have timed moves (e.g., chess)
  - Want to use up all time
  - Don't want to be caught without move, though
- Solution:
  - Iterative deepening
  - Search to depth 1, then 2, then...
  - At any time, always have last best move "on tap"
- Example of an *anytime algorithm*

Games of chance

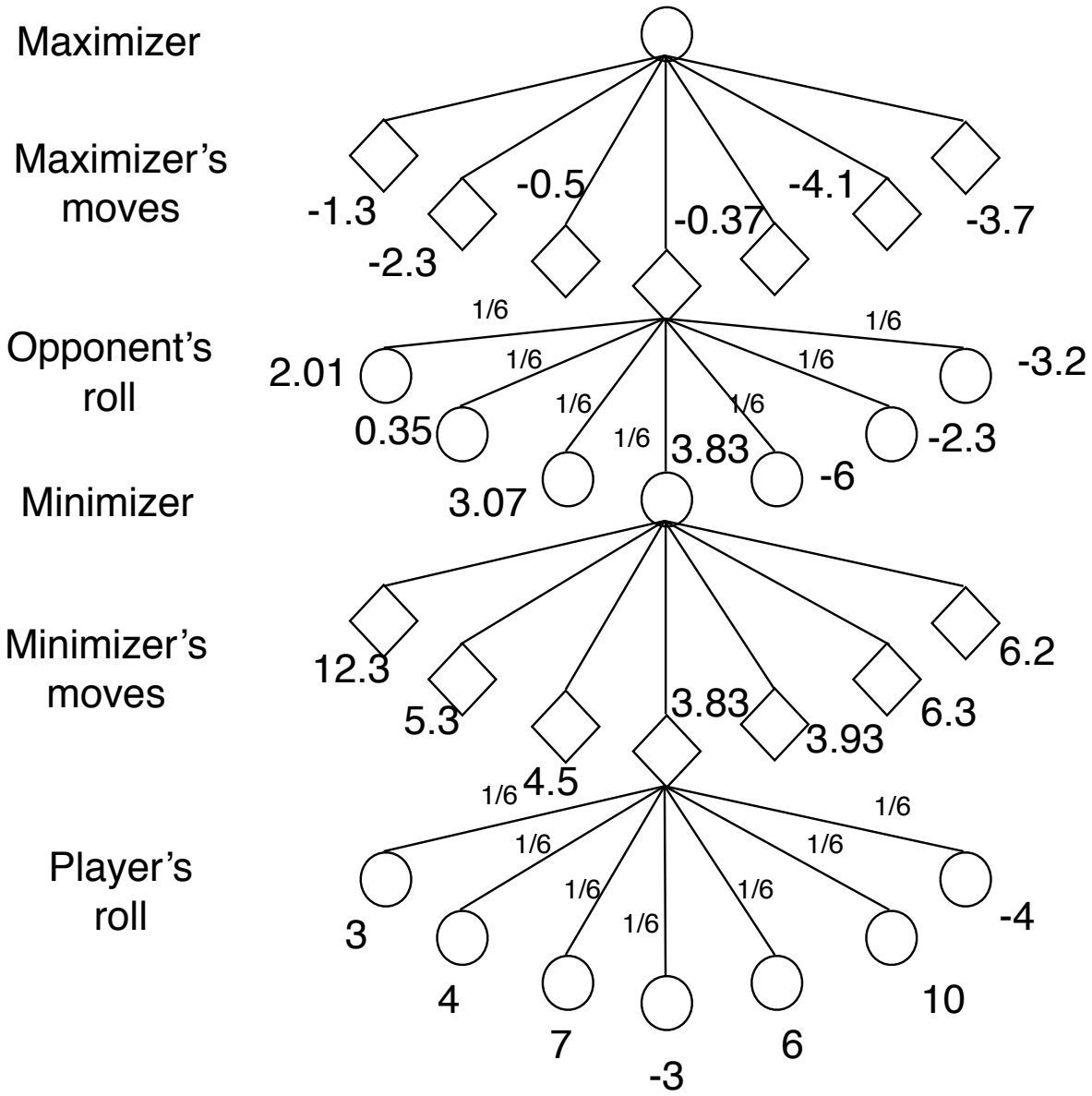
Games of Chance

- Many games: *chance* (or stochastic) element
  - Cards, dice, spinners, etc.
  - Uncertainty about opponent's rationality/utility function
- How to modify minimax?
  - First iteration level is easy: roll the dice (e.g.), then minimax
  - What about opponent's move?
  - What about player's future moves?

Chance nodes

- Add *chance nodes* to game tree at points of uncertainty
- Branches:
  - Different outcomes
  - Tag with probability
  - E.g., roll two dice:
    - Chance node has 11 branches
    - Tagged: $\frac{1}{36}, \frac{2}{36}, \frac{3}{36}, \frac{4}{36}, \frac{5}{36}, \frac{6}{36}, \frac{5}{36}, \frac{4}{36}, \frac{3}{36}, \frac{2}{36}, \frac{1}{36}$
- Explore tree as in minimax, but:
  - Chance node $\Rightarrow$ weighted average of children
  - Weights: probabilities

Example



Algorithm (w/o α-β cutoffs)

```
1: function EXPECTIMINIMAX(state,depth)
2:     if depth = 0 then
3:         return state & STATICEVAL(state)
4:     else if maximizer then
5:         for child in CHILDREN(state) do
6:             best = argmax(best, EXPECTIMINIMAX(child,depth -1)
7:         return best
8:     else if minimizer then
9:         for child in CHILDREN(state) do
10:            best = argmin(best, EXPECTIMINIMAX(child,depth -1)
11:        return best & value(best)
12:    else
13:        for each chance value do
14:            sum = sum + value of state returned by
15:              EXPECTIMINIMAX(UPDATE(state,chance value))
16:        return state & sum ÷ # chance values
```

Properties

- Not solely contingent on utilities
- May predict, but not get, a win (etc.)
- Time: $\mathcal{O}(b^d c^d) \leftarrow c$ = # chance values

Games and the Real World

- Some people consider game-playing a metaphor for real-world problem solving
- With opponents, it's clear...
- ...but can also treat the *world* as an opponent

Games and the Real World

- Many researchers use *game theory* to make decisions
  - Think of it as a generalization of minimax
  - *Payoff matrices*: play the role of static evaluation function

		Player 2	
		c	d
player 1	a	3 4	1 2
	b	4 1	2 3

Looking ahead

Looking ahead: Deep learning

- Deep reinforcement learning: AlphaGo, AlphaZero
- So why minimax?
  - Simple
  - Doesn't require enormous computational bandwidth to train
  - Some insights about dealing with more complex scenarios than games