Lecture 22
Principles of Functional Programming
Summer 2020

Games II

Minimax and Alphabeta

Last Time

- Implemented playable games in SML
- Our game implementation consisted of:
  - A `GAME` (specifying rules, how to make moves, etc.)
  - `PLAYER`s (plays a particular `GAME`, provides function `next_move` assigning a "choice" of move to each state)
  - Games are refereed by a `CONTROLLER`, who facilitates play between two `PLAYER`s playing the same game.
- Implemented `Nim`, where states were of the form `(s,Minnie)` or `(s,Maxie)` for `s:int` nonnegative. A move is a positive `int` `i` which is less than or equal to `Int.min(3,s)`.

Making plays

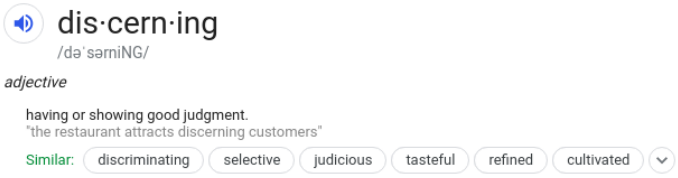
We'll deal with 4 different kinds of players:

- Human players (our game library includes utilities to accept user input to determine `next_move`)
- Directly-implemented players (`NimPlayer` from tomorrow's lab)
- MiniMax players (this lecture)
- Alphabeta players (Lecture 22.5, `games` homework)

Section 1

How to Build Smart PLAYERs

What do we mean by smart?



We want to design our `PLAYER`s such that their `next_move` function makes decisions which generally lead to it winning the game more often. Contrast:

```
RunNim.play RunNim.HvM;
RunNim.play RunNim.HvP;
```

So what we want to do is build a player who "knows what's good for her": who is able to assess the moves available to her, decide which one has the most favorable outcome, and make the corresponding move her `next_move`.

Game Trees

Formally, we make sense of games mathematically by examining the corresponding *game tree*. A game tree is a finitely-branching tree where

- The nodes represent *game states*
- The edges represent *moves*
- The root node is the current state of the game, and the rest of the tree represents different outcomes achieveable by a certain series of moves from the two players
- The children of a given node are the states reachable from that game state by the current player making a valid move.

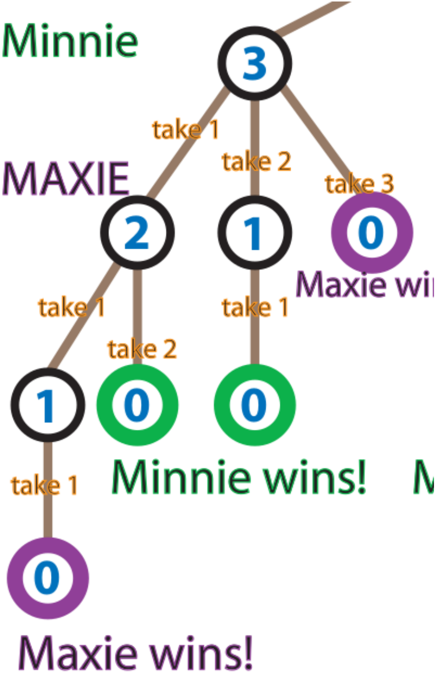
We'll call our players '**Maxie**' and '**Minnie**'.

(Game Tree Example)

Developing a game strategy

Game trees allow us to more easily make sense of the following observation:

A good human player is one who is thinking a few moves into the future. To decide which next move is best for her to take, she thinks through some scenarios of how the game *could* go *if* she were to make that move (and what their opponent might do in response, and how she could respond to that, and so on), and then pick the move with the most attractive range of possible outcomes.


Minnie
3
take 1
take 2
take 3
MAXIE
2
1
0
Maxie wi
take 1
take 2
take 1
1
0
0
take 1
Minnie wins!
0
Maxie wins!

Developing a game strategy

Game trees allow us to depict the following observation:
A good human player is one who is thinking a few moves into the future. To decide which next move is best for them to take, they think through some scenarios of how the game *could* go *if* they were to make that move (and what their opponent might do in response, and how they could respond to that, and so on), and then pick the move with the most attractive range of possible outcomes.
To design smart computer `PLAYER`s which mimic this, we'll have our computer `PLAYER`s recursively explore the game tree, and determine the outcomes of play (assuming the players are playing optimally), ultimately to determine what move would be best from the current situation.

(Game Tree Example)

Estimation Nation

- Problem: it's impractical (and often impossible) to visit every node of the tree
- Solution: explore some of the tree, and guess
  - Have a fixed 'search depth' d
  - Explore the top d levels of the tree (i.e. the game states than can be reached from the current one in d moves or fewer)
  - When you hit your search depth, use your knowledge of the game to assign an appropriate value to that state, and treat that value as the value of the node.
- More precisely: we'll have a function `estimate` which takes a game state (for instance, a value of type `Nim.State.t`) and returns a "guess" of the goodness or badness of that state.

Estimators : Some design principles

An *estimator* for a game `G` is a function assigning "guesses" to each state to (perhaps roughly) indicate who's winning.

- The "guesses" will usually be numerical (e.g. `int`s): lower numbers better for **Minnie**, larger numbers better for **Maxie**. The scale is arbitrary: all that matters is the relative ordering of states.
- The goal here is to induce an ordering on states, i.e. articulate a sense in which states are "better" or "worse" than each other (from one player's perspective).
- We want "better" to mean "more likely to win" (as best as possible)
- A given `GAME` will have many possible estimators, with varying degrees of sophistication, and which may weight different factors differently. When we make `PLAYER`s who use these estimators to calculate their `next_move`, these differences will correspond to different *playing styles* or *strategies*.

ESTIMATOR.sig

22.9

```
signature ESTIMATOR =
sig
  structure Game : GAME

  type guess
  datatype est = Definitely of Game.Outcome.t
               | Guess of guess

  val compare : est * est -> order
  val toString : guess -> string

  val estimate : Game.State.t -> guess
end
```

Notes about estimator

- Note that the only operation on values of type `est` is comparison (the function `compare`). We don't – in general – require `guesses` to be numbers at all, we just require that they be ordered.
- We **transparently** ascribe to this signature. While we don't require in general that `guess` is implemented as `int` or `real`, if we do happen to implement it that way we want to have access to the associated methods (e.g. from the basis structures `Int` and `Real`).

Nim has a perfect estimator

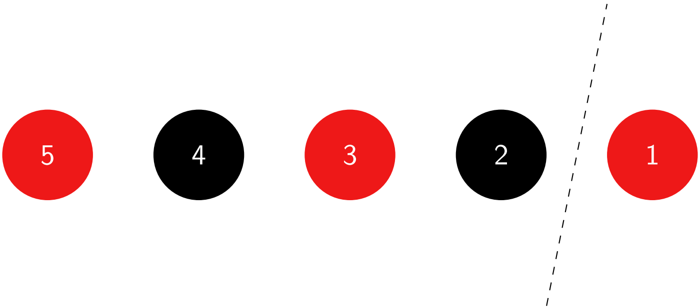
Player `p` can guarantee a win from `(s,flip p)`

iff

$\texttt{s mod 4} \cong 1$

(remember

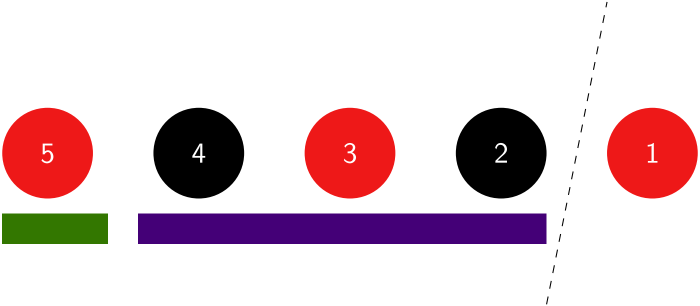
```
fun flip Maxie = Minnie | flip Minnie = Maxie)
```



Nim has a perfect estimator

Player `p` can guarantee a win from `(s,flip p)`
iff
`s mod 4` $\cong$ 1

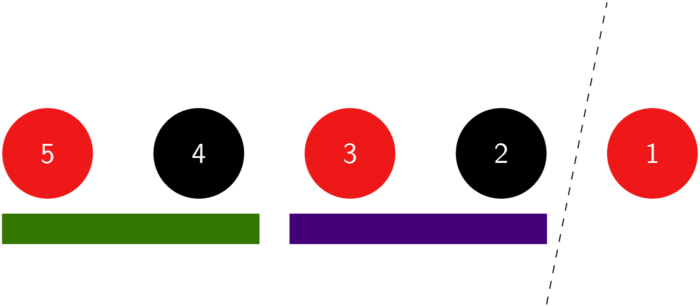
(remember
```
fun flip Maxie = Minnie | flip Minnie = Maxie
```
)



Nim has a perfect estimator

Player `p` can guarantee a win from `(s, flip p)`
iff
`s mod 4` ≅ 1

(remember
```
fun flip Maxie = Minnie | flip Minnie = Maxie)
```

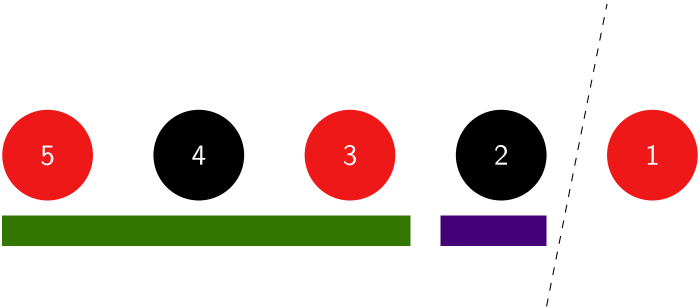


Nim has a perfect estimator

Player `p` can guarantee a win from `(s,flip p)`
iff
`s mod 4` $\cong$ 1

(remember

```
fun flip Maxie = Minnie | flip Minnie = Maxie)
```



Nim has a perfect estimator

So, assuming the other player plays optimally, whoever's turn it is when `s` is of the form `(4*k)+1` for some `k:int` will *lose*.

```
(* recall a value of Nim.State.t is (s,p)
   for some nonnegative int s and either
   p=Minnie or p=Maxie *)
(* estimate : Nim.State.t -> int *)
fun estimate (s,p) =
      case (s mod 4, p) of
         (1,Minnie) => 1
       | (1,Maxie) => ~1
       | (_,Minnie) => ~1
       | (_,Maxie) => 1
```

This is somewhat *too* clean of an example: most games don't have perfect estimators. Rather, the best we can do is make pretty good guesses! To design an estimator, we'll usually use some combination of simple heuristics and more sophisticated theory.

For instance, here's a common heuristic for chess: for a chess piece `p`, let `v(p)` be given by the following chart

Symbol	♙	♘	♗	♖	♕
Piece	pawn	knight	bishop	rook	queen
Value	1	3	3	5	9

Then put

$$\texttt{estimate(S)} = \left(\sum_{\substack{\text{Pieces } \texttt{p} \text{ \textbf{Maxie}} \\ \text{has in play (in } \texttt{S}\text{)}}} \texttt{v(p)} \right) - \left(\sum_{\substack{\text{Pieces } \texttt{p} \text{ \textbf{Minnie}} \\ \text{has in play (in } \texttt{S}\text{)}}} \texttt{v(p)} \right)$$

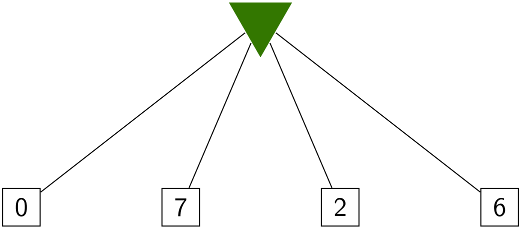
Section 2

The MiniMax Algorithm

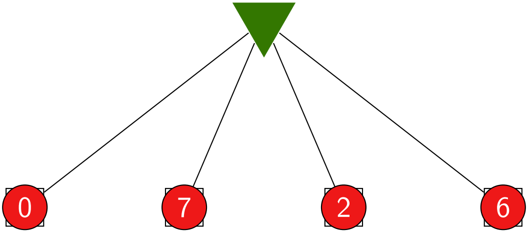
Takeaways

- We should assign each node an estimator guess, its “value”.
- The value of a node should reflect who’s winning from that node, which depends on the moves available from that state.
- Should assume players play optimally.

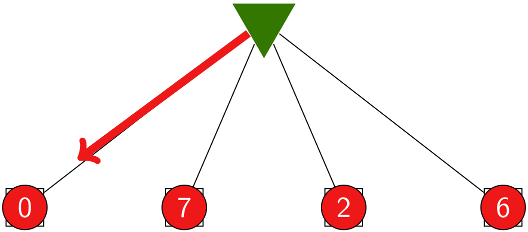
Minnie Search



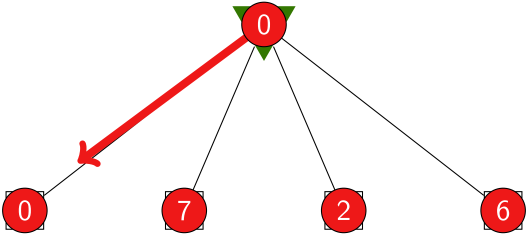
Minnie Search



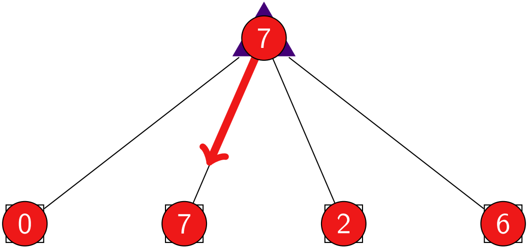
Minnie Search



Minnie Search



Maxie Search



The MiniMax Algorithm

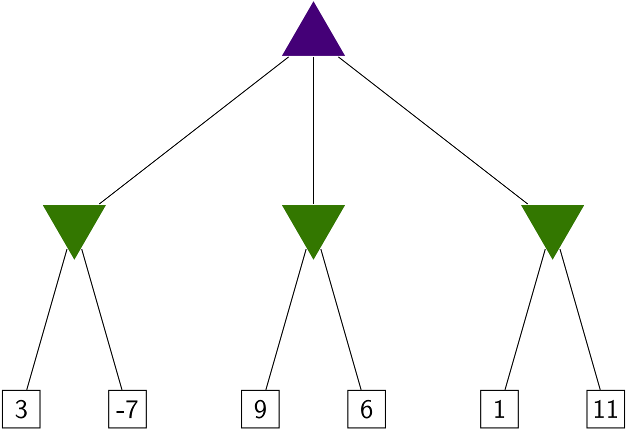
Fix a search depth d.

1. Traverse the game tree down to the d-th level.[1]
2. Call the estimator to assign values to the d-th level.
3. Work upwards, assigning values to nodes according to the Minnie and Maxie principles described above
   - For **Minnie** nodes: the value should be the *minimum* of the values of the child nodes
   - For **Maxie** nodes: the value should be *maximum* of the values of the child nodes.

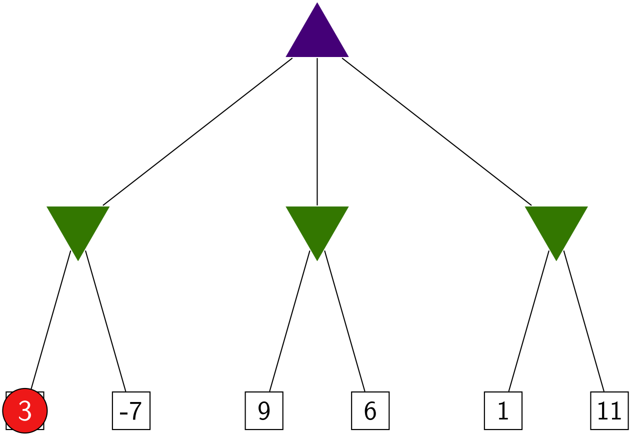
Once we've filled all the way to the top of the tree (our current state), then we can decide which move to make based on the estimated values.

[1] For every node encountered where the game is over, assign such nodes the value `Definitely` of whoever the winner is.

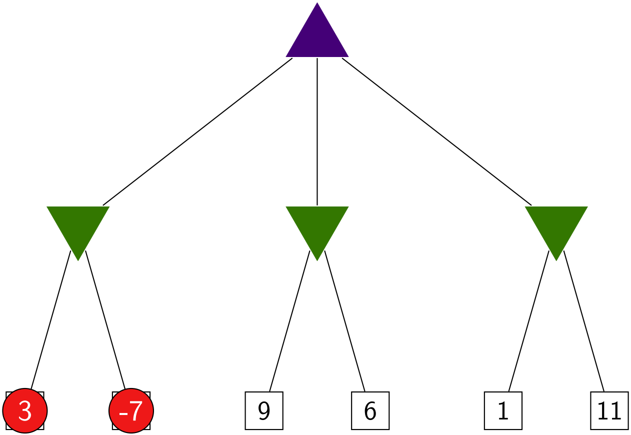
MiniMax



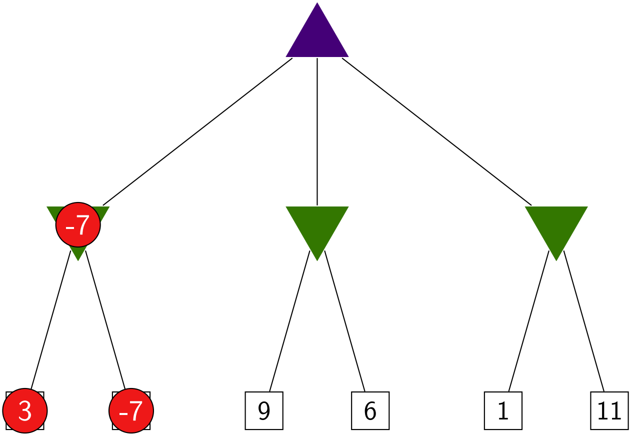
MiniMax



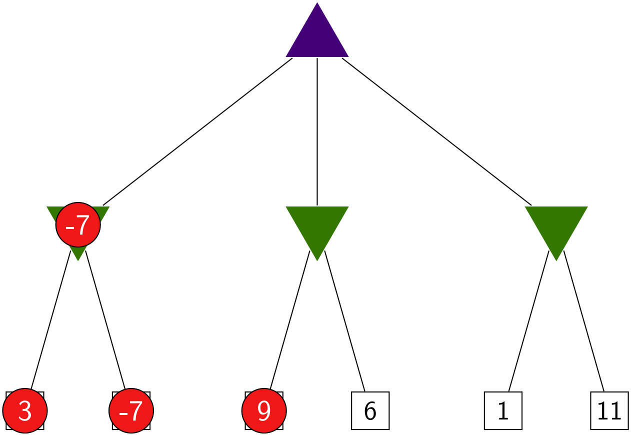
MiniMax



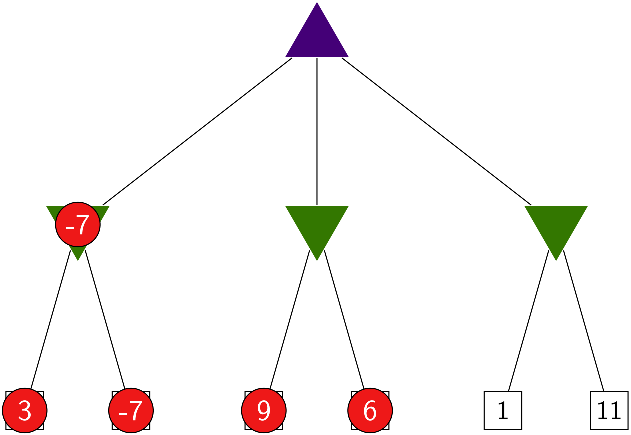
MiniMax



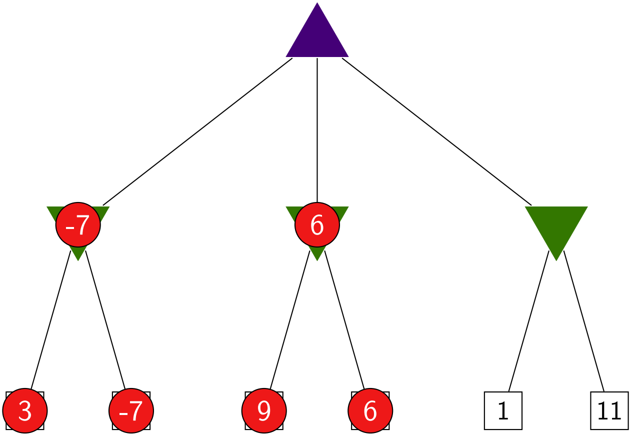
MiniMax



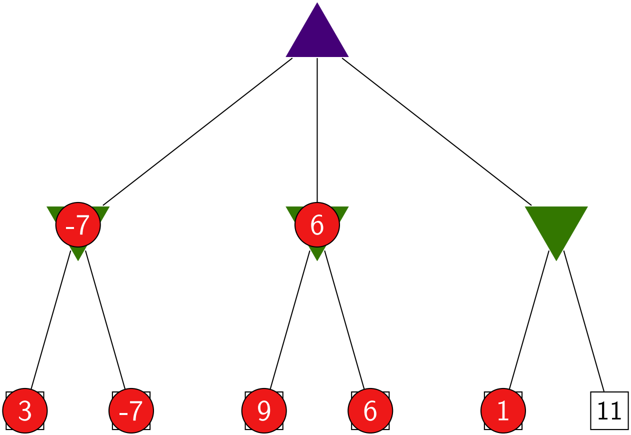
MiniMax



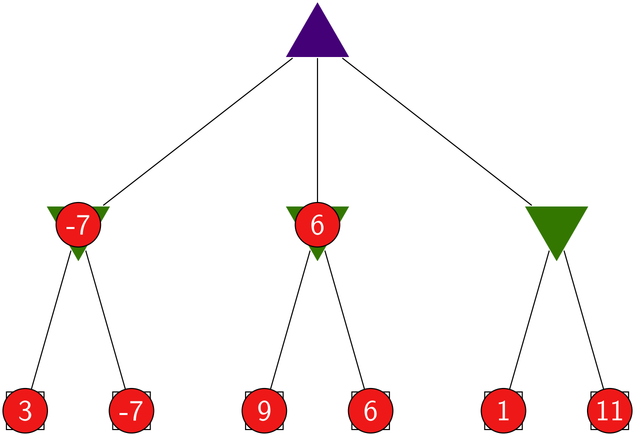
MiniMax



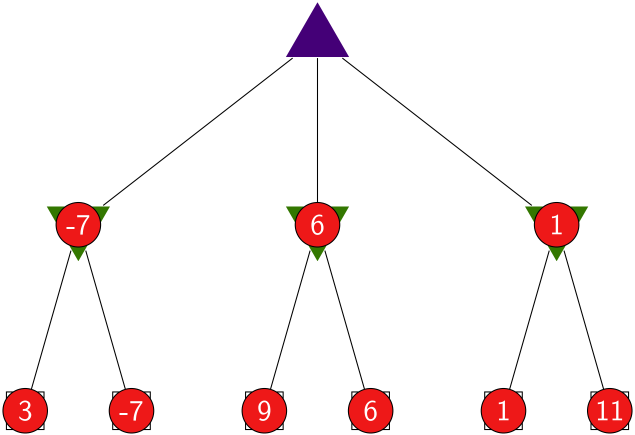
MiniMax



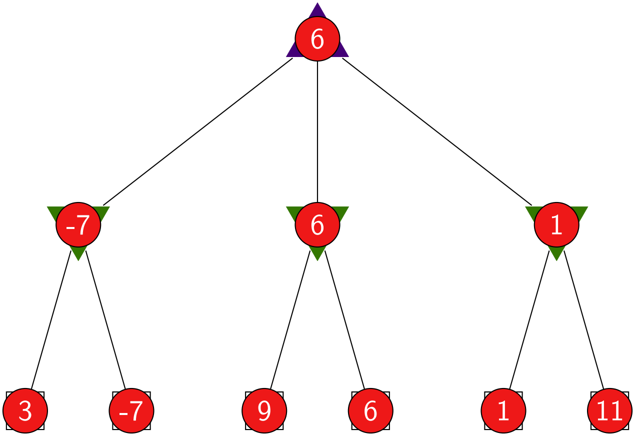
MiniMax



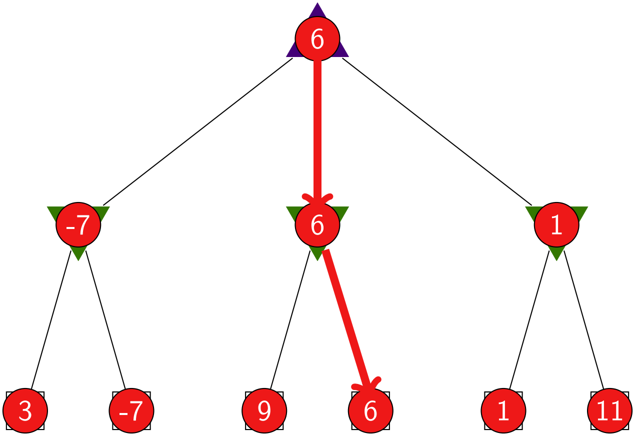
MiniMax



MiniMax



MiniMax



22.10

```
signature SETTINGS =
sig
  structure Est : ESTIMATOR

  val search_depth : int
end
```

22.11

```
functor MiniMax (Settings : SETTINGS):>PLAYER
  where Game = Settings.Est.Game =
struct
  structure Est = Settings.Est
  structure Game = Est.Game
```

22.12

```
type edge = Game.Move.t * Est.est
fun valueOf ((_,value) : edge) = value
fun moveOf ((move,_) : edge) = move

fun max ((m1,v1) : edge, (m2,v2) : edge) : edge =
  case Est.compare (v1, v2) of
    LESS => (m2, v2)
  | _ => (m1, v1)

fun min ((m1,v1) : edge, (m2,v2) : edge) : edge =
  case Est.compare (v1, v2) of
    GREATER => (m2, v2)
  | _ => (m1, v1)
```

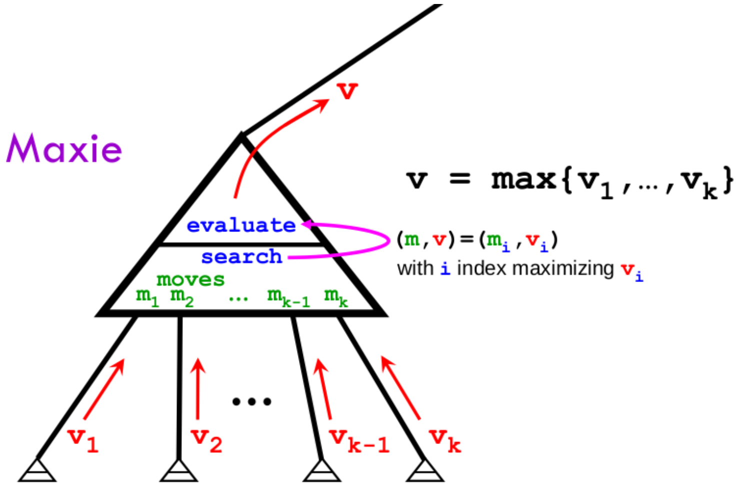
```
reduce1 : ('a * 'a -> 'a) -> 'a Seq.seq -> 'a
```
REQUIRES: g is total and associative, S is nonempty

ENSURES:

`reduce1 g` $\langle$`x1,...,xn`$\rangle \cong$ `g(x1,g(x2,g(...,, xn)))`

22.13

```
(* choose:Player.t -> edge Seq.seq -> edge *)
  fun choose Player.Maxie  = Seq.reduce1 max
    | choose Player.Minnie = Seq.reduce1 min
```



Maxie
v
$v = max\{v_1, \ldots, v_k\}$
evaluate
search
$(m, v) = (m_i, v_i)$
with i index maximizing v_i
moves
m_1 m_2 … m_{k-1} m_k
…
v_1
v_2
v_{k-1}
v_k

22.14

```
(* search : int -> G.State.t -> edge       *)
(* REQUIRES: d > 0                          *)
fun search (d : int) (s : Game.State.t):edge =
  choose
    (Game.player s)
    (Seq.map
       (fn m => (m, evaluate
                      (d - 1)
                      (Game.play (s,m))))
       (Game.moves s)
    )
```

22.15

```
(* evaluate : int -> Game.status -> Est.est *)
(* REQUIRES: d >= 0                          *)
and evaluate (d : int) (st : Game.status) :
    Est.est =
  case st of
    Game.Playing s => (
      case d of
        0 => Est.Guess (Est.estimate s)
      | _ => valueOf (search d s)
      )
   | Game.Done oc => Est.Definitely oc
```

22.16

```
val next_move =
  moveOf o search Settings.search_depth
```

Lecture 22.5 (to be released tonight)

- Advantages and disadvantages of MiniMax
- Saving some work: Alpha-Beta Pruning

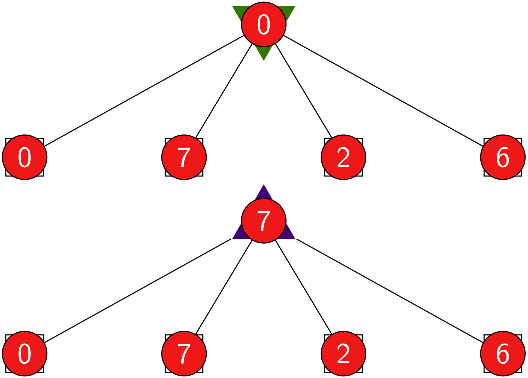
Thank you!

Lecture 22.5
Principles of Functional Programming
Summer 2020

Games 2½

Alpha-Beta Pruning

Minimax



Pros and Cons

Advantages of Minimax:

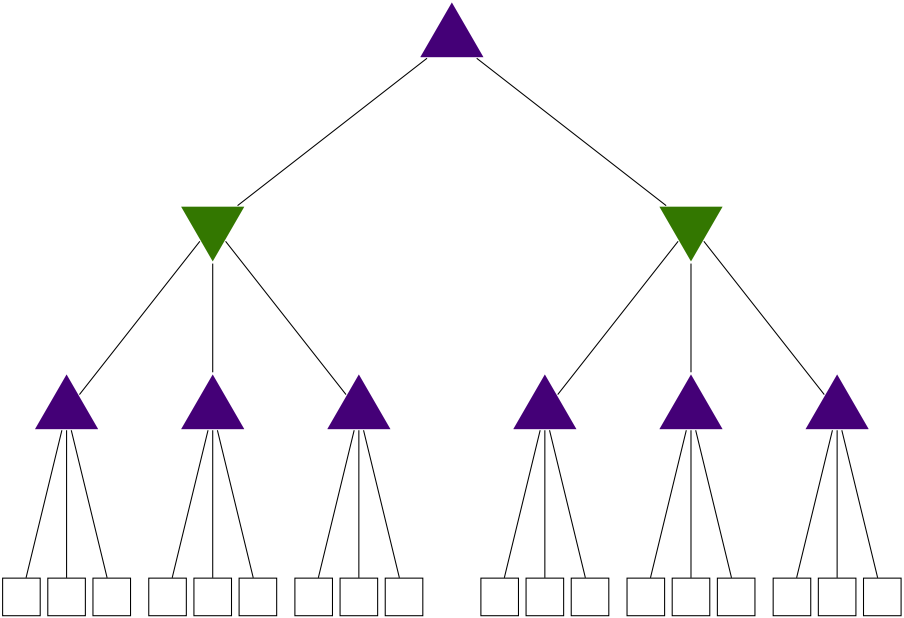
- Correctly determines optimal play
- Massively parallelizable

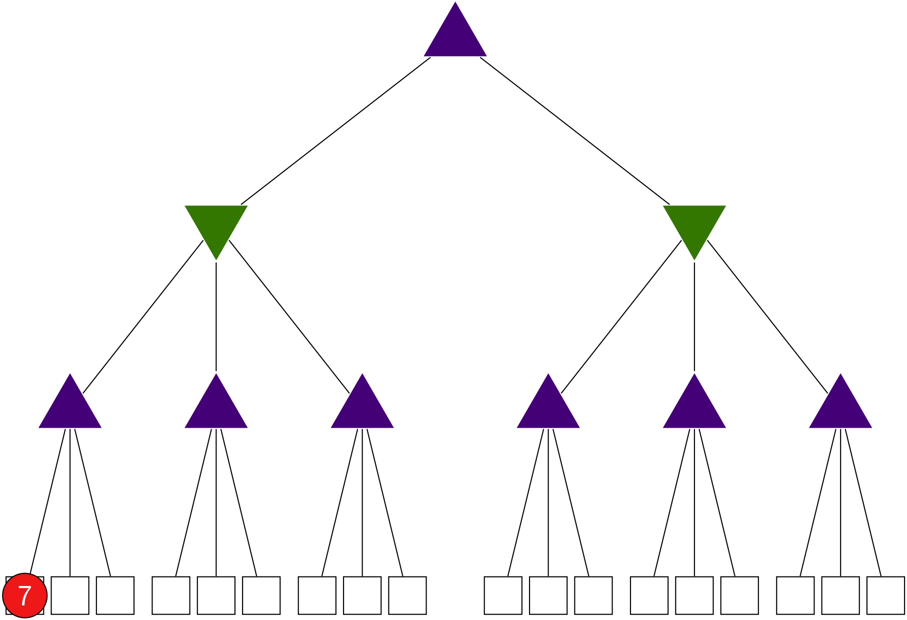
Disadvantages of Minimax:

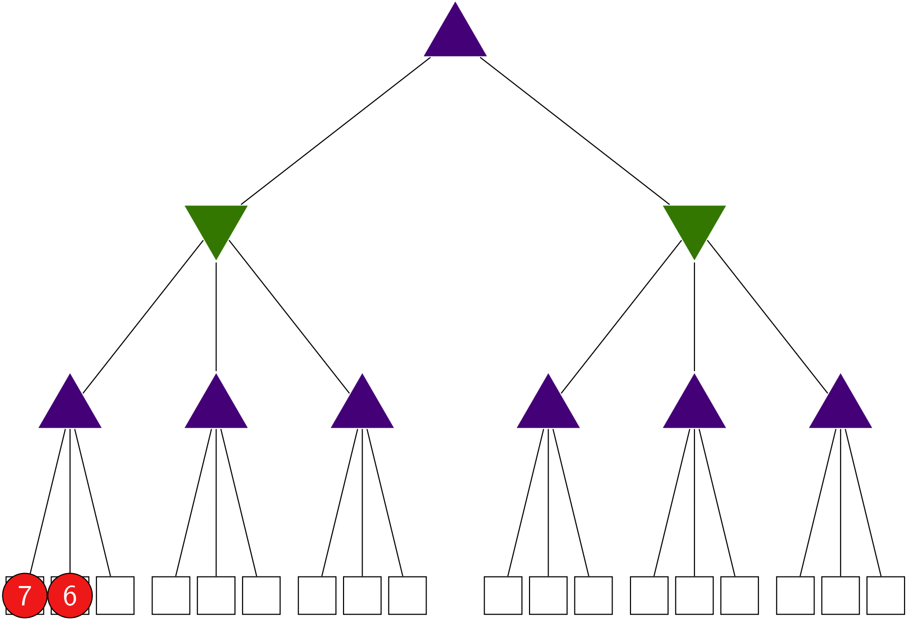
- Huge amount of work
- Indeed, often performs unnecessary computation

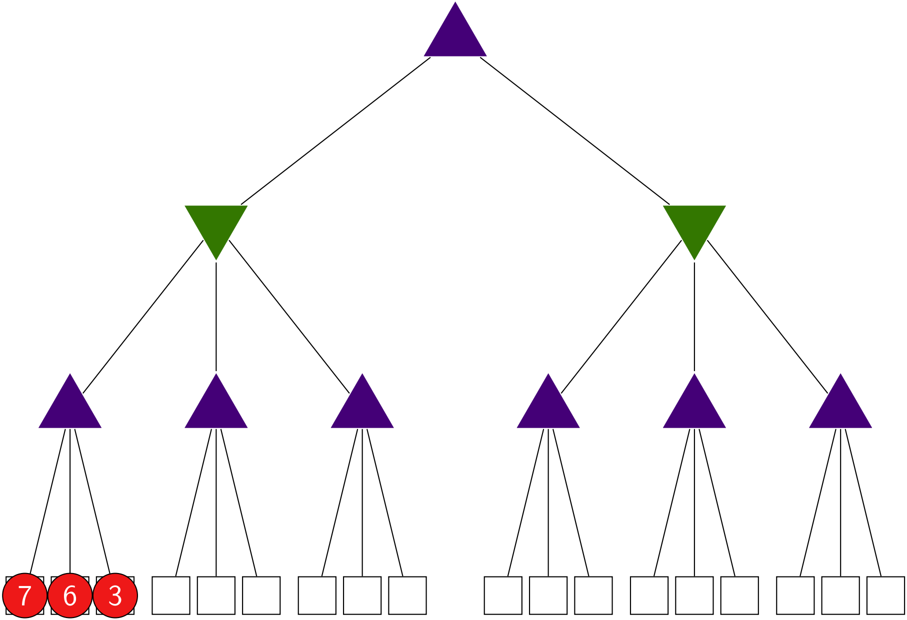
Section 3

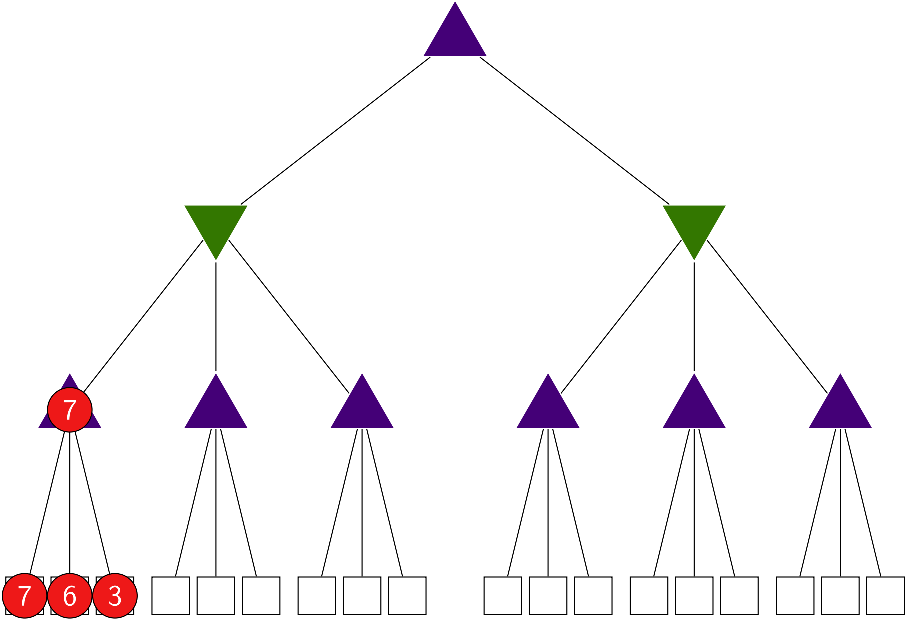
Alpha-Beta Pruning

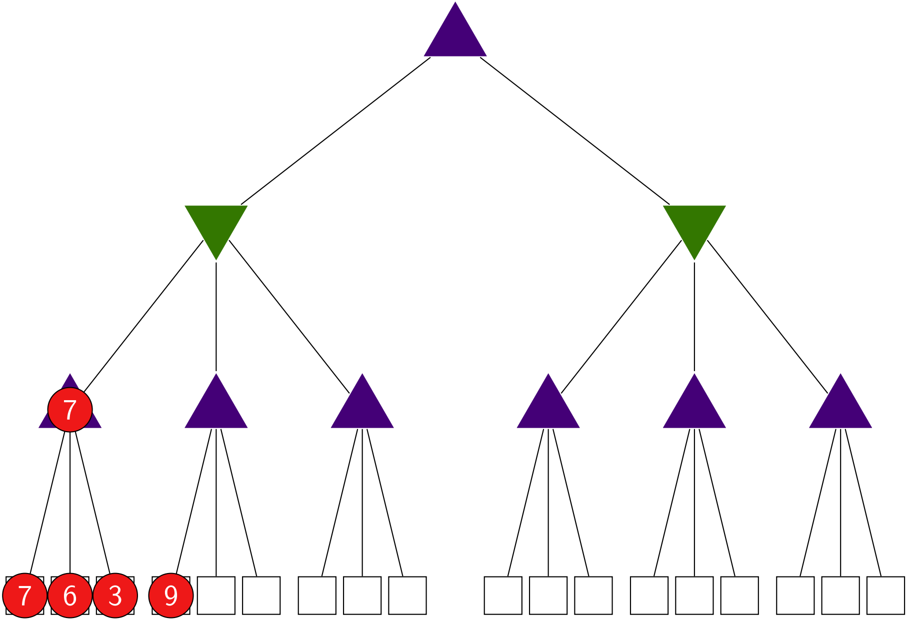


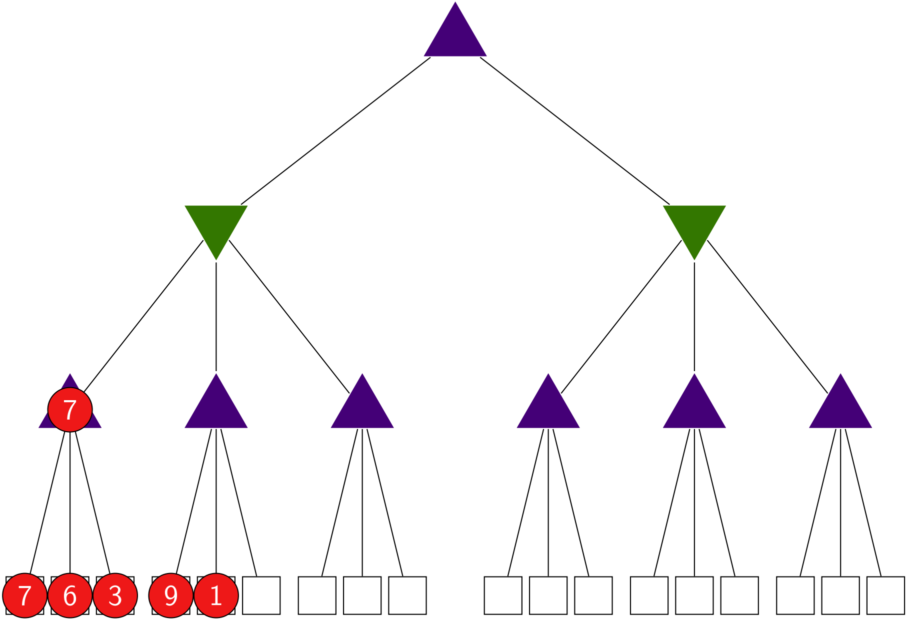

7

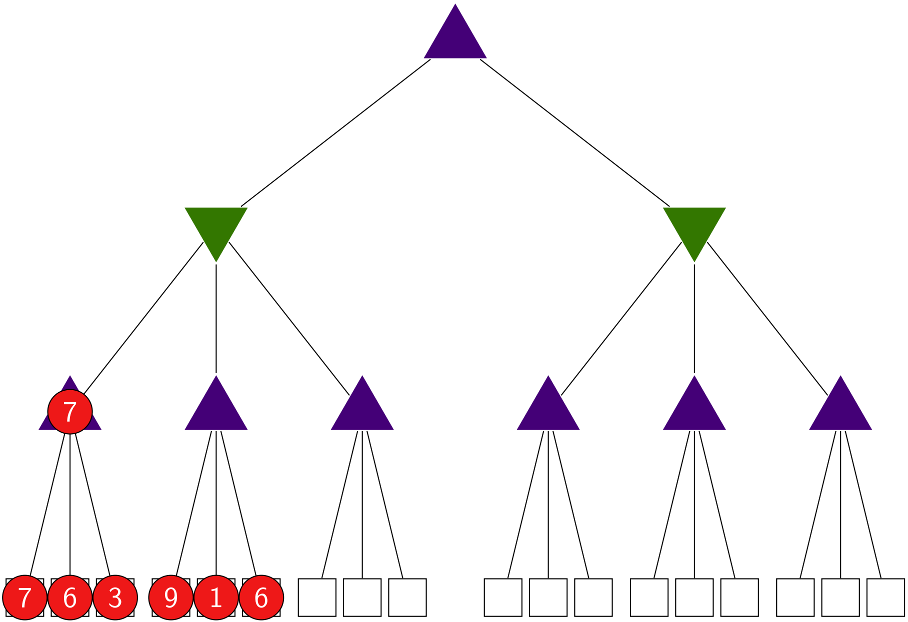

7
6

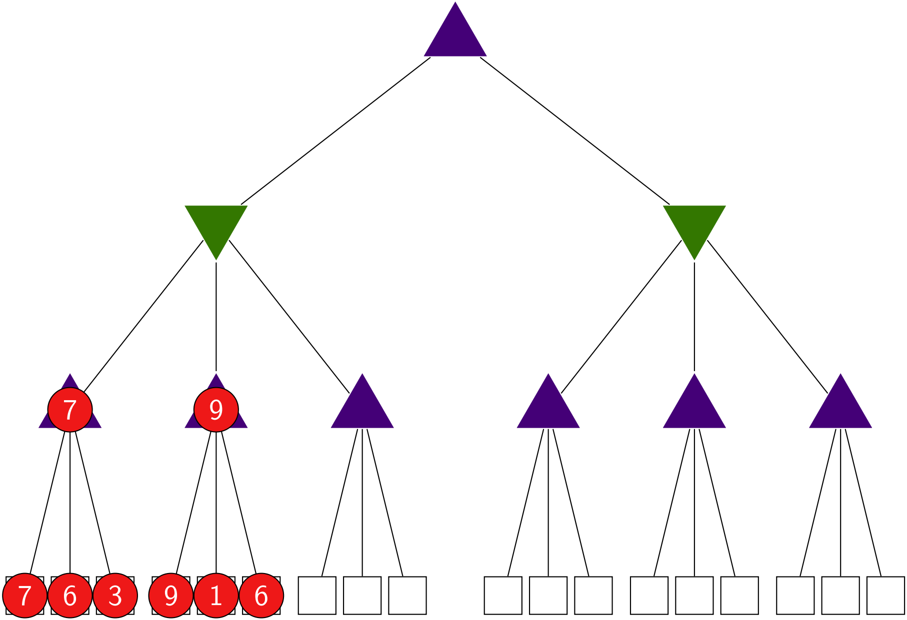

7
6
3

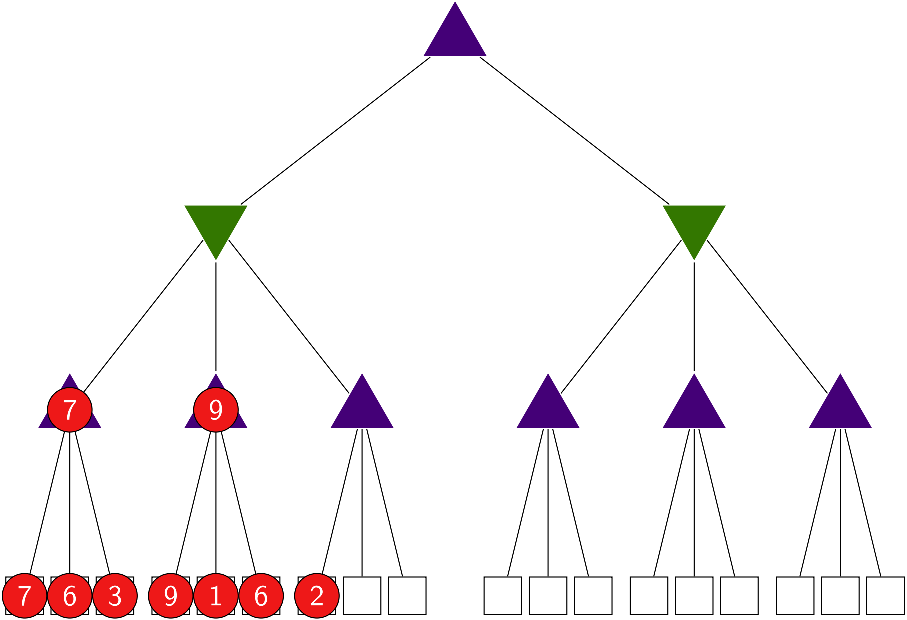

7
7
6
3

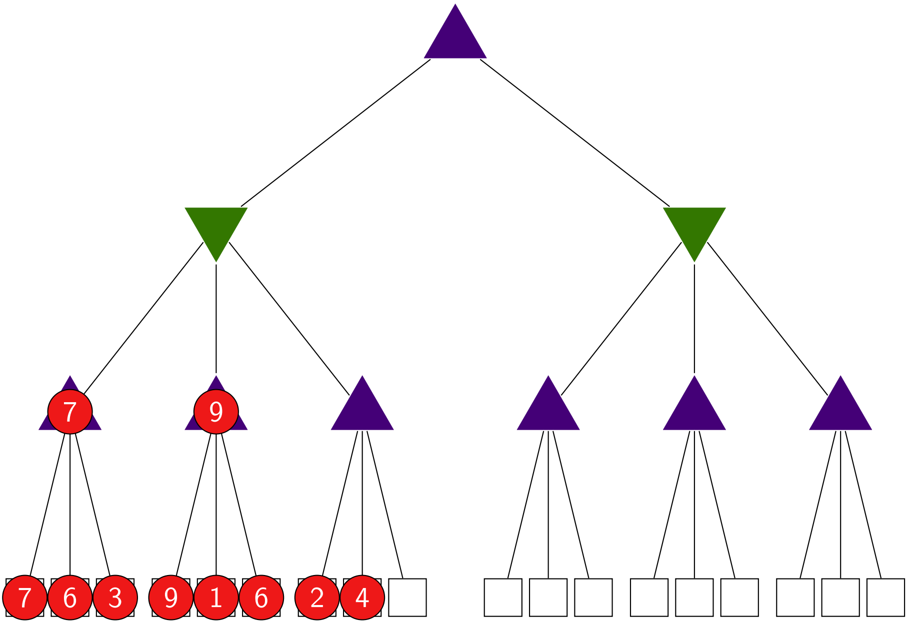

7
7
6
3
9

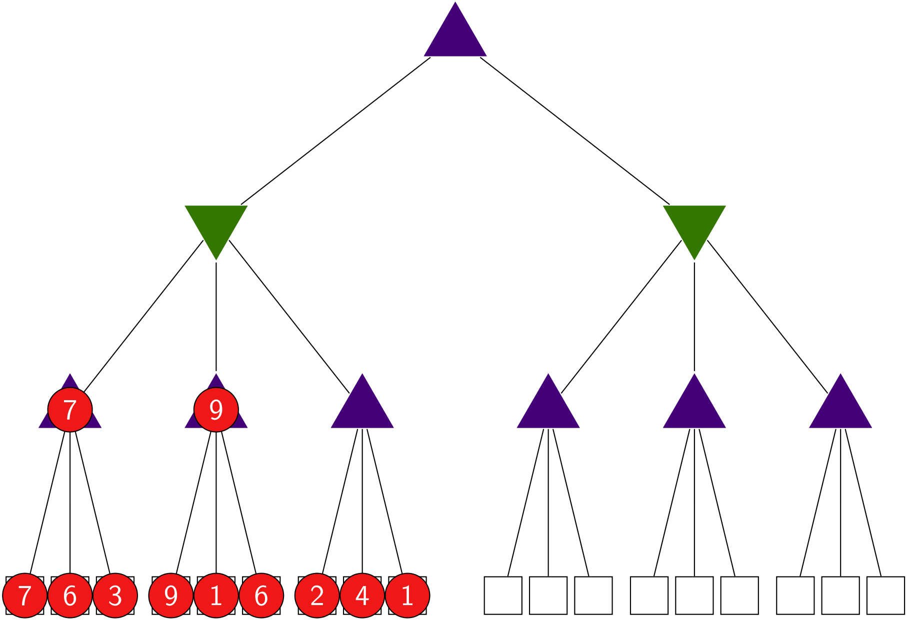

7
7
6
3
9
1

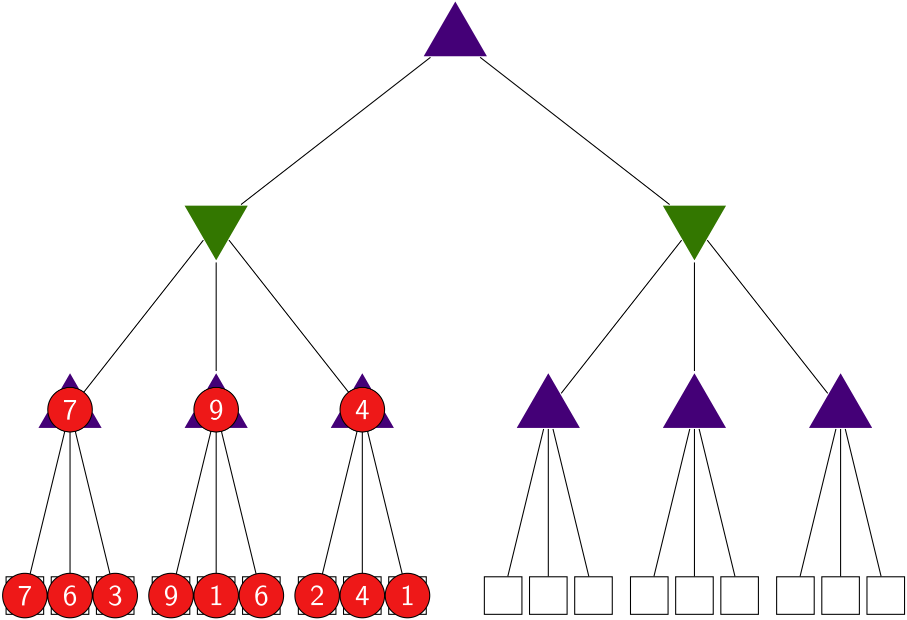

7
7
6
3
9
1
6

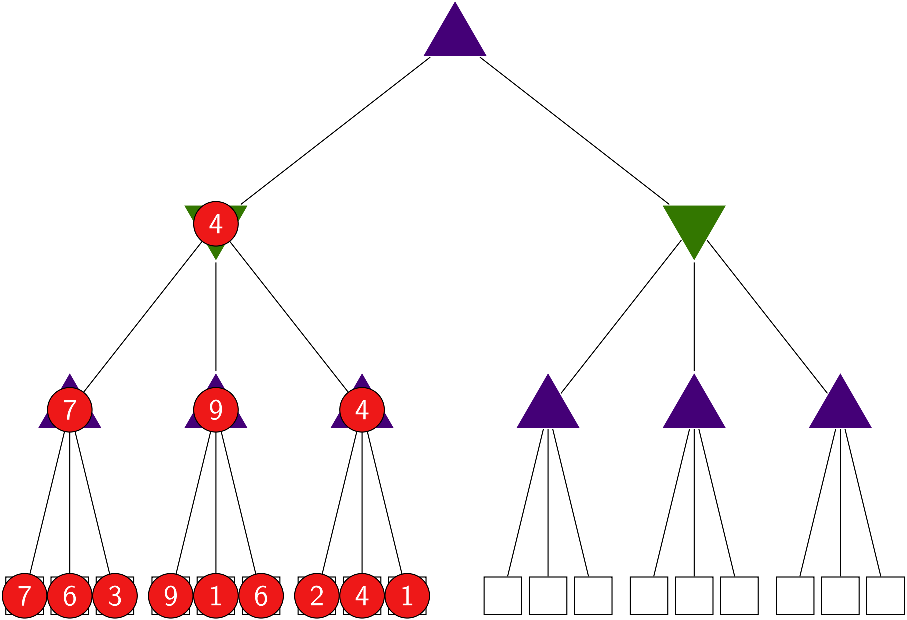

7
9
7
6
3
9
1
6

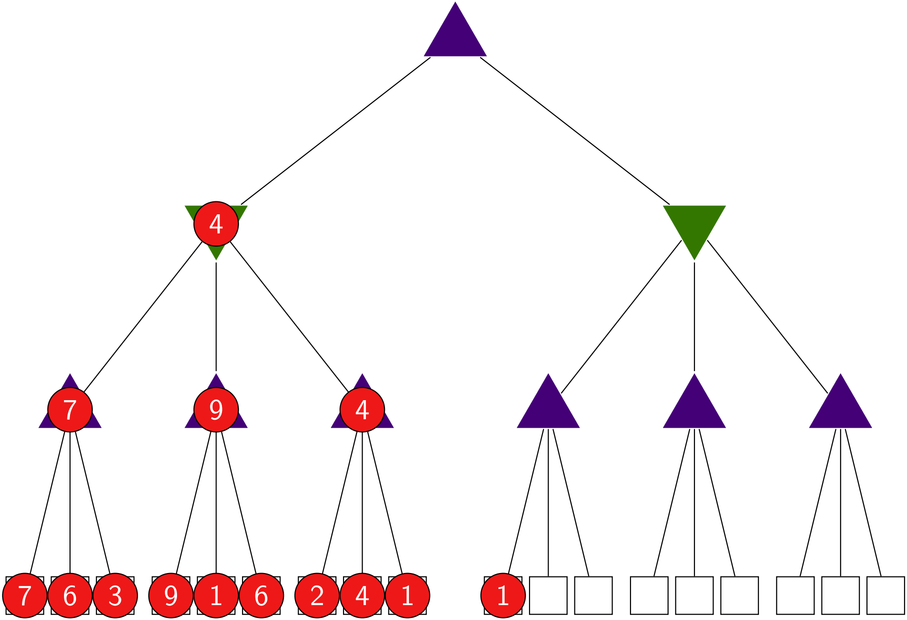

7
9
7
6
3
9
1
6
2

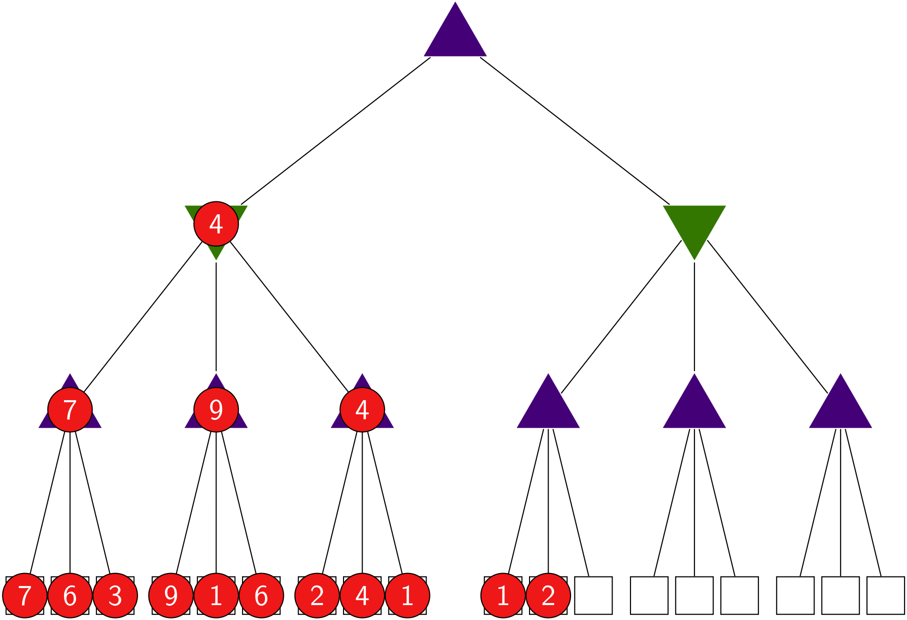

7
9
7
6
3
9
1
6
2
4

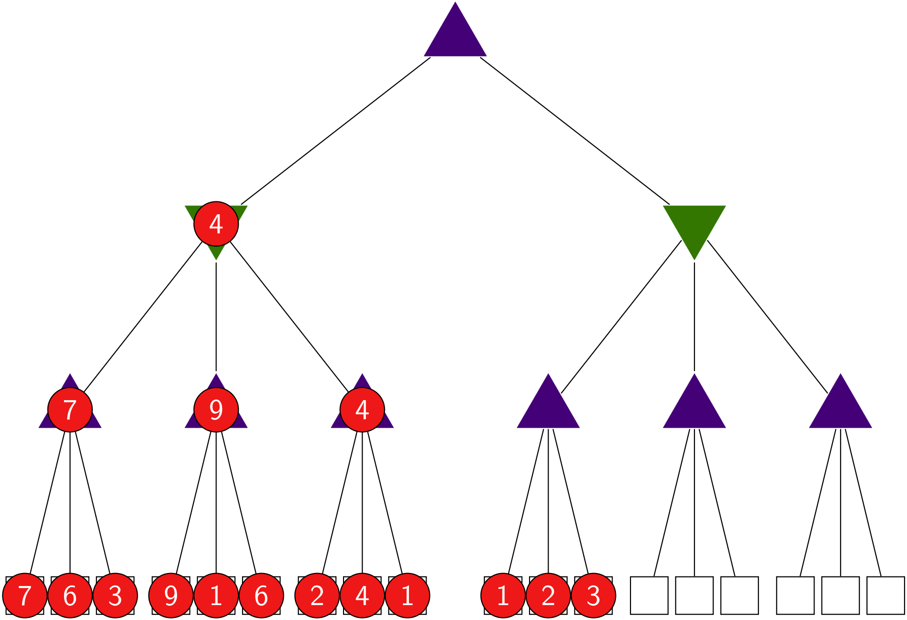

7
9
7
6
3
9
1
6
2
4
1

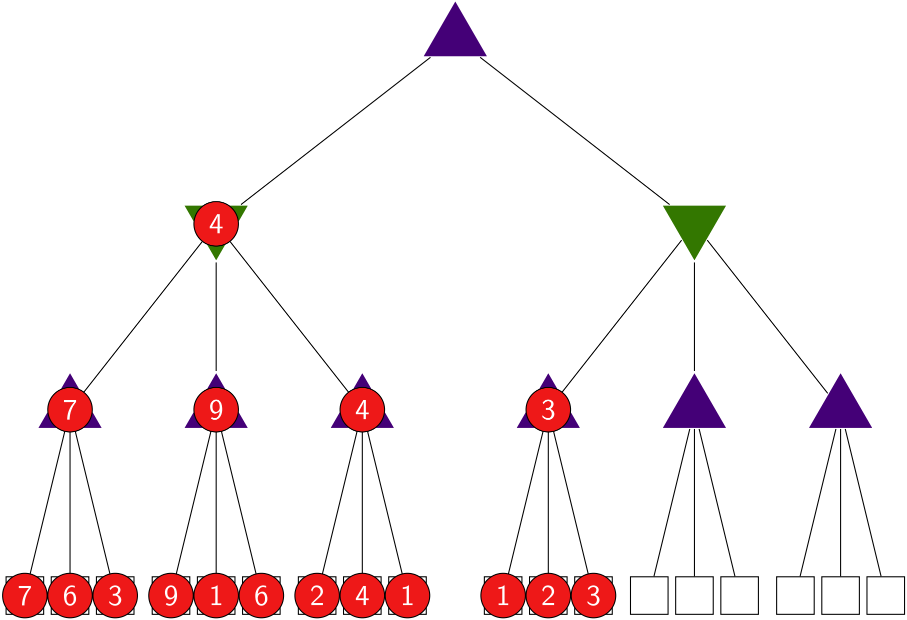

7
9
4
7
6
3
9
1
6
2
4
1

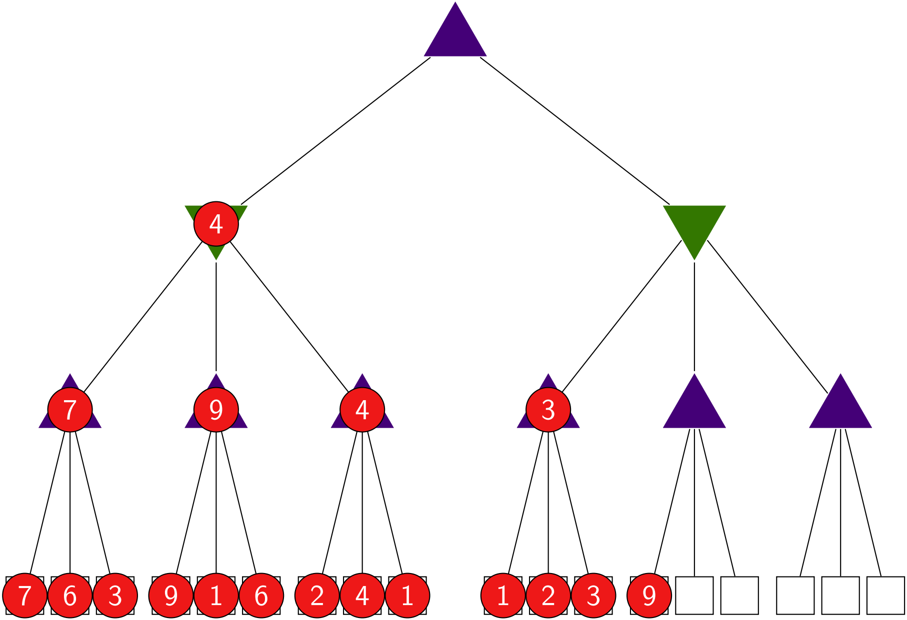

4
7
9
4
7 6 3 9 1 6 2 4 1

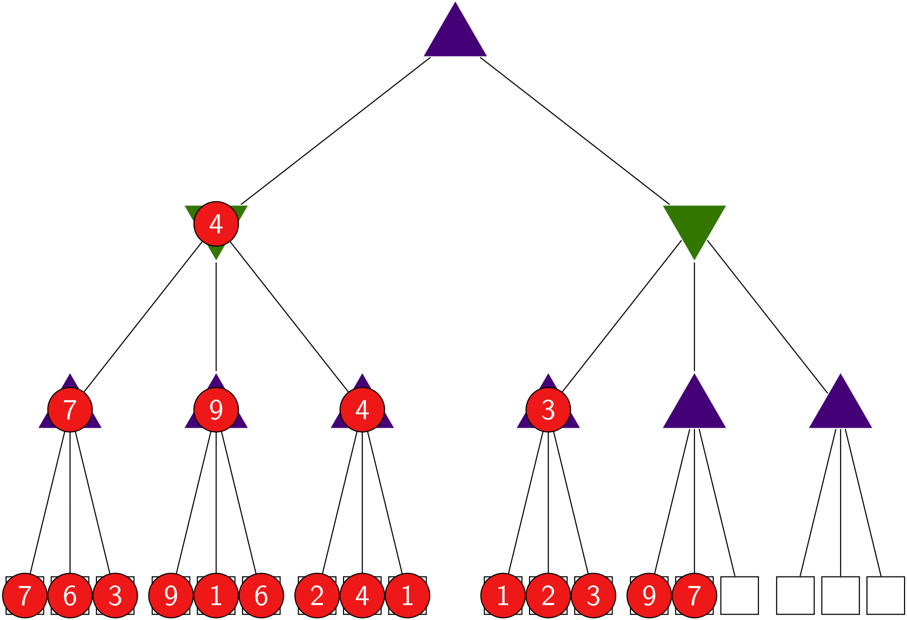

4
7
9
4
7
6
3
9
1
6
2
4
1
1

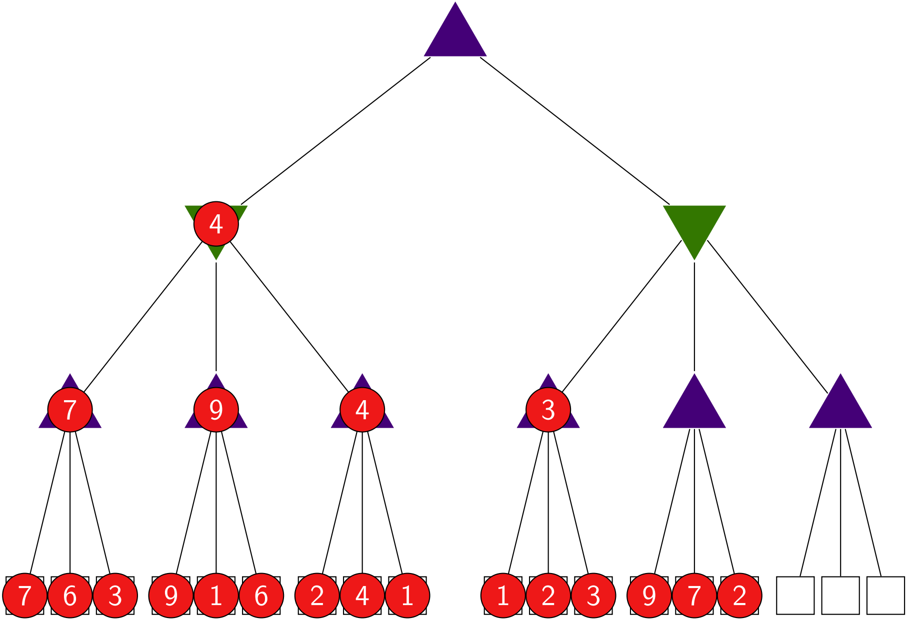

4
7
9
4
7
6
3
9
1
6
2
4
1
1
2

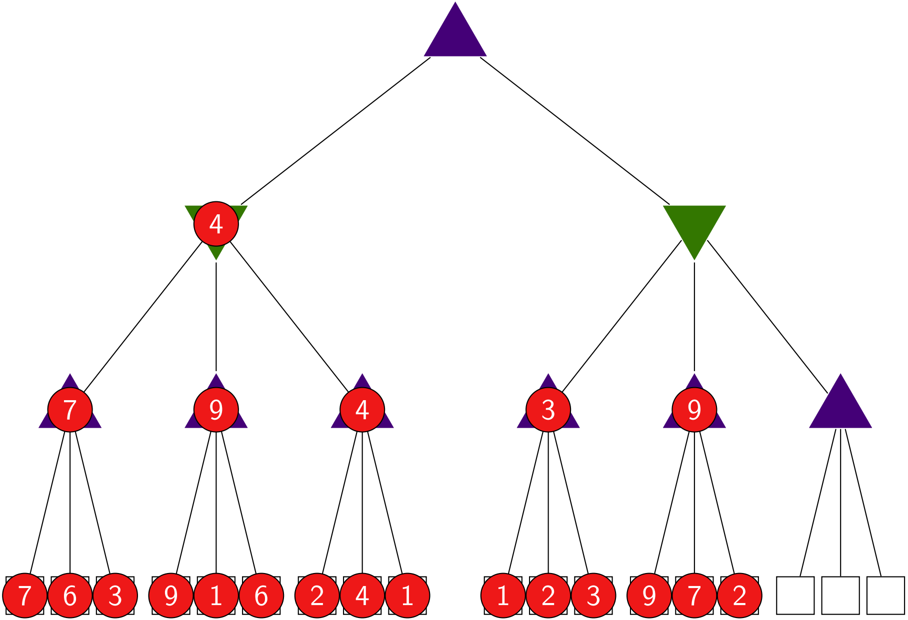

4
7
9
4
7
6
3
9
1
6
2
4
1
1
2
3

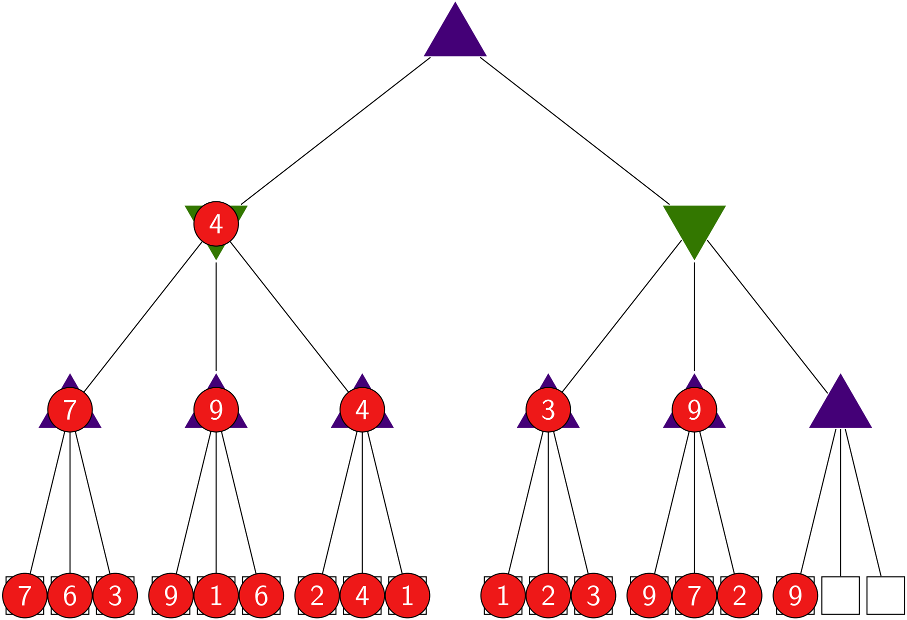

4
7
9
4
3
7
6
3
9
1
6
2
4
1
1
2
3

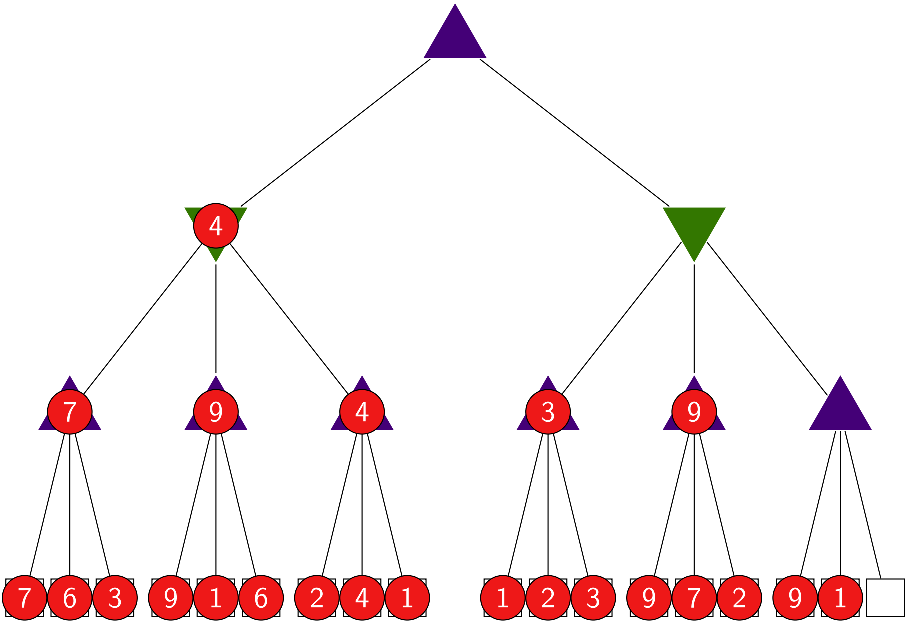

4
7
9
4
3
7
6
3
9
1
6
2
4
1
1
2
3
9

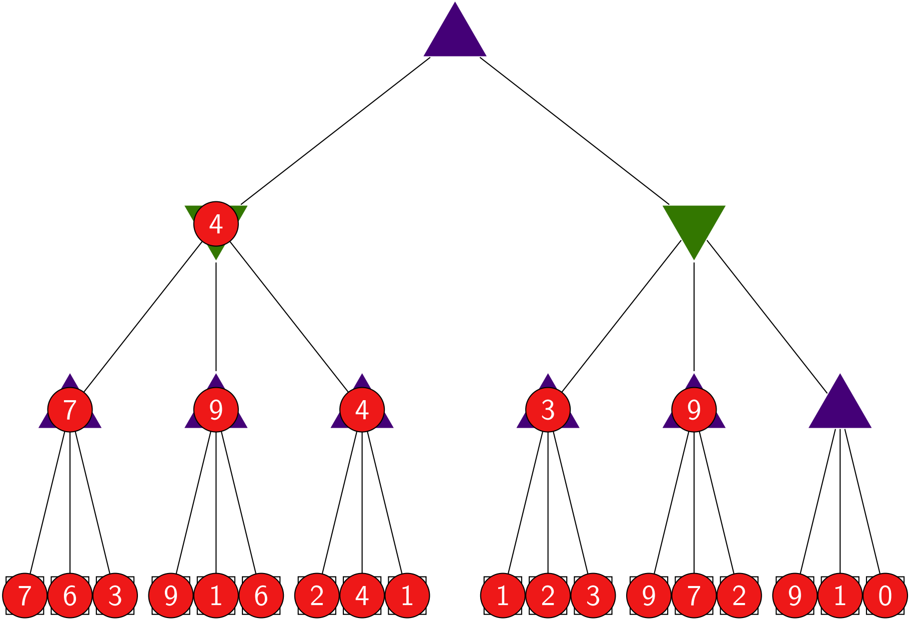

4
7
9
4
3
7 6 3 9 1 6 2 4 1 1 2 3 9 7

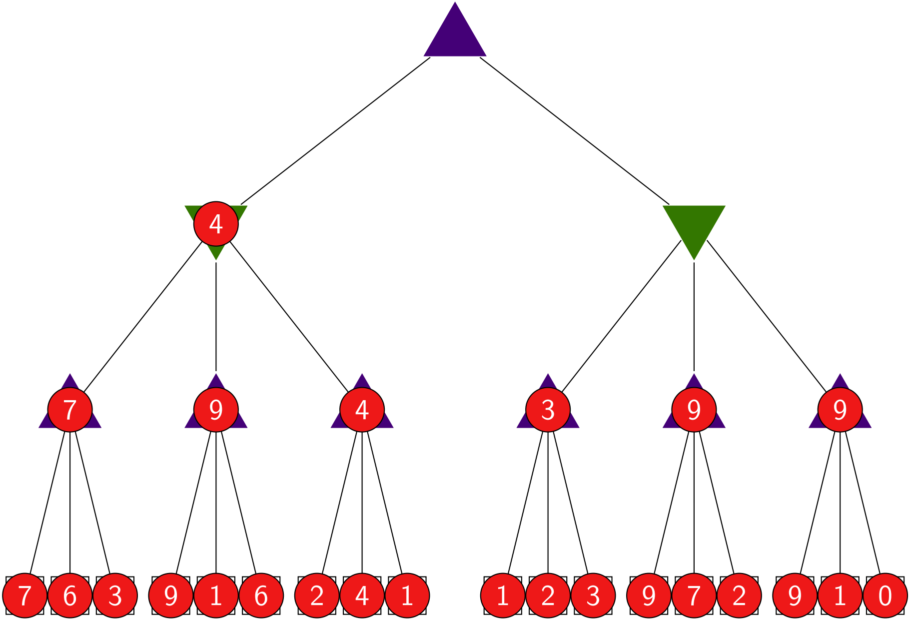

4
7
9
4
3
7
6
3
9
1
6
2
4
1
1
2
3
9
7
2

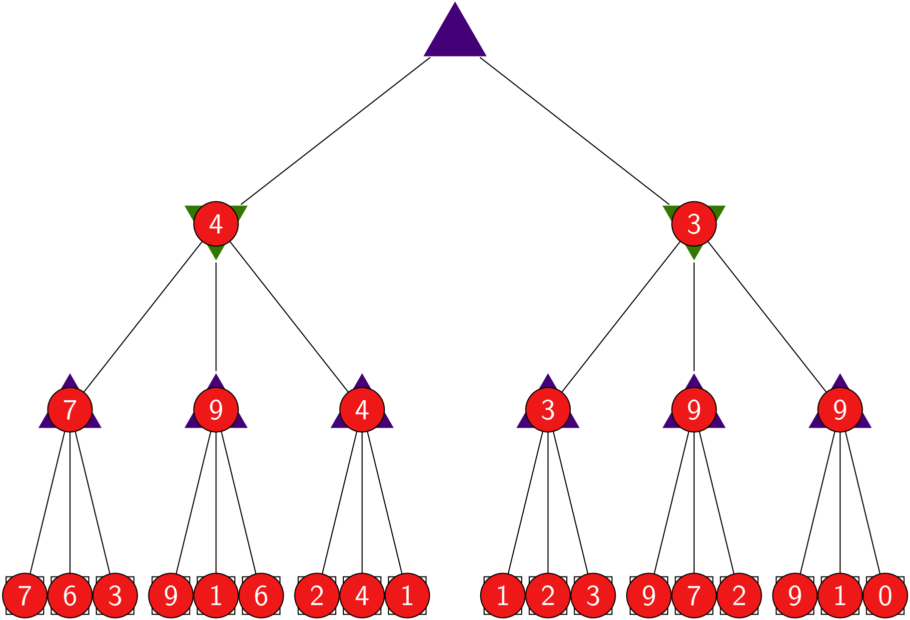

4
7
9
4
3
9
7
6
3
9
1
6
2
4
1
1
2
3
9
7
2

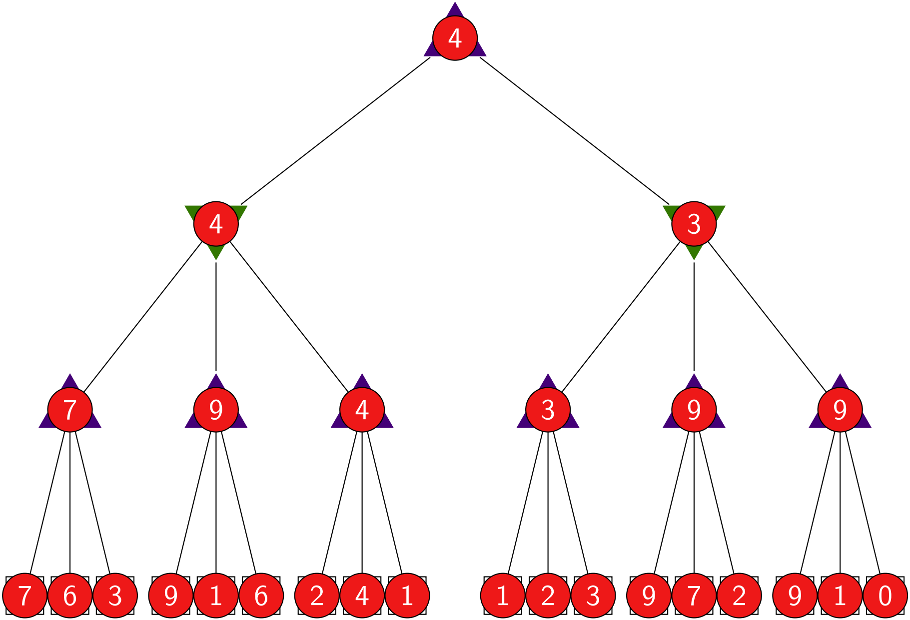

4
7
9
4
3
9
7 6 3
9 1 6
2 4 1
1 2 3
9 7 2
9


4
7
9
4
3
9
7 6 3 9 1 6 2 4 1 1 2 3 9 7 2 9 1

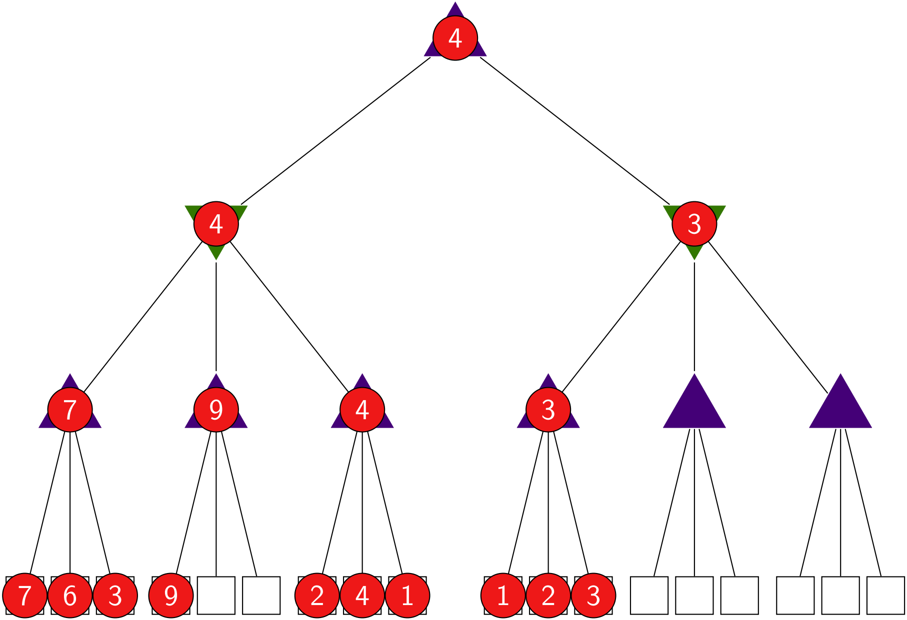

4
7
9
4
3
9
7 6 3 9 1 6 2 4 1 1 2 3 9 7 2 9 1 0

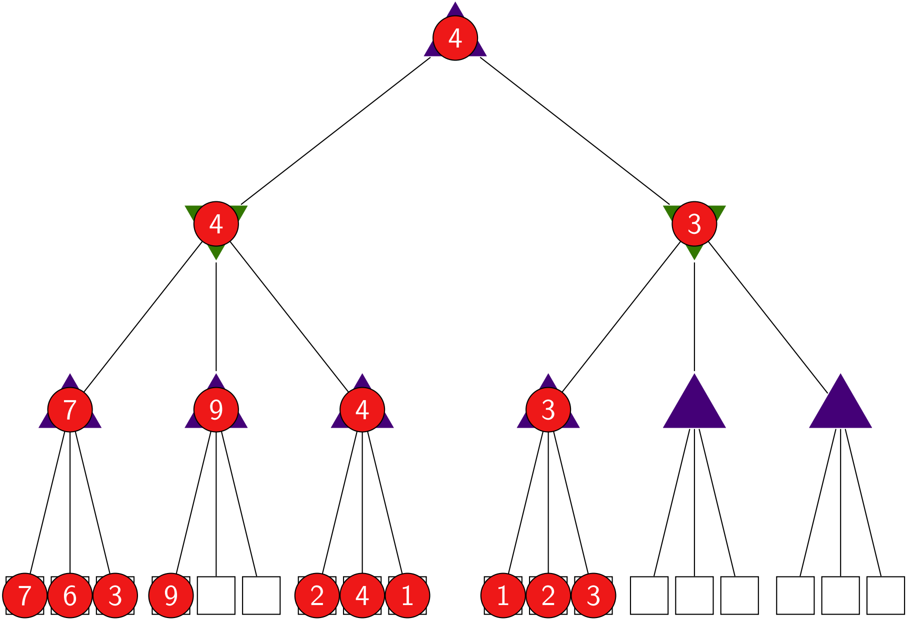

4
7
9
4
3
9
9
7 6 3 9 1 6 2 4 1 1 2 3 9 7 2 9 1 0


4
3
7
9
4
3
9
9
7 6 3 9 1 6 2 4 1 1 2 3 9 7 2 9 1 0



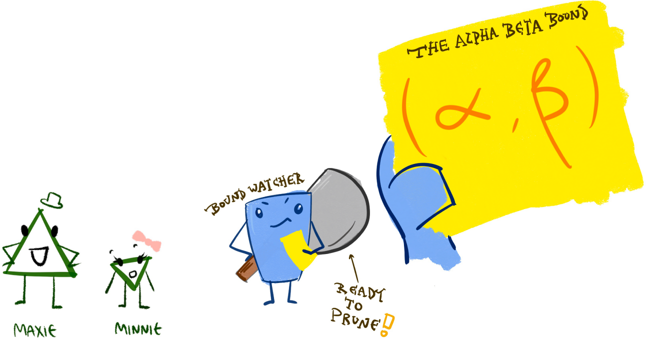
4
4
3
7
9
4
3
9
9
7 6 3
9 1 6
2 4 1
1 2 3
9 7 2
9 1 0


4
4
3
7
9
4
3
7
6
3
9
2
4
1
1
2
3


4
4
3
7
9
4
3
7
6
3
9
2
4
1
1
2
3

Idea

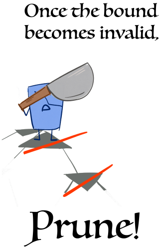
As we perform MiniMax, we want to keep track of "what can be guaranteed" to inform us when we're exploring an irrelevant subtree.

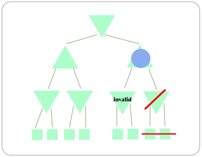


Idea

So, for every point along the minimax algorithm, there is some estimator guess value α, which represents the greatest value that **Maxie** can *guarantee*. Analogously, we'll keep track of some value β representing the least value that **Minnie** can guarantee. It must be the case that $\alpha \leq \beta$.

When **Minnie** encounters a node whose value is $\leq \alpha$, then she can "prune" the rest of the current subtree: **Maxie** won't let the game get to this point. If **Maxie** encounters a node whose value is $\geq \beta$, then she prunes.





Handy Chart

	$x \leq \alpha$	$\alpha < x < \beta$	$\beta \leq x$
Maxie	Ignore	Update α	Prune
Minnie	Prune	Update β	Ignore

(Alphabeta example)