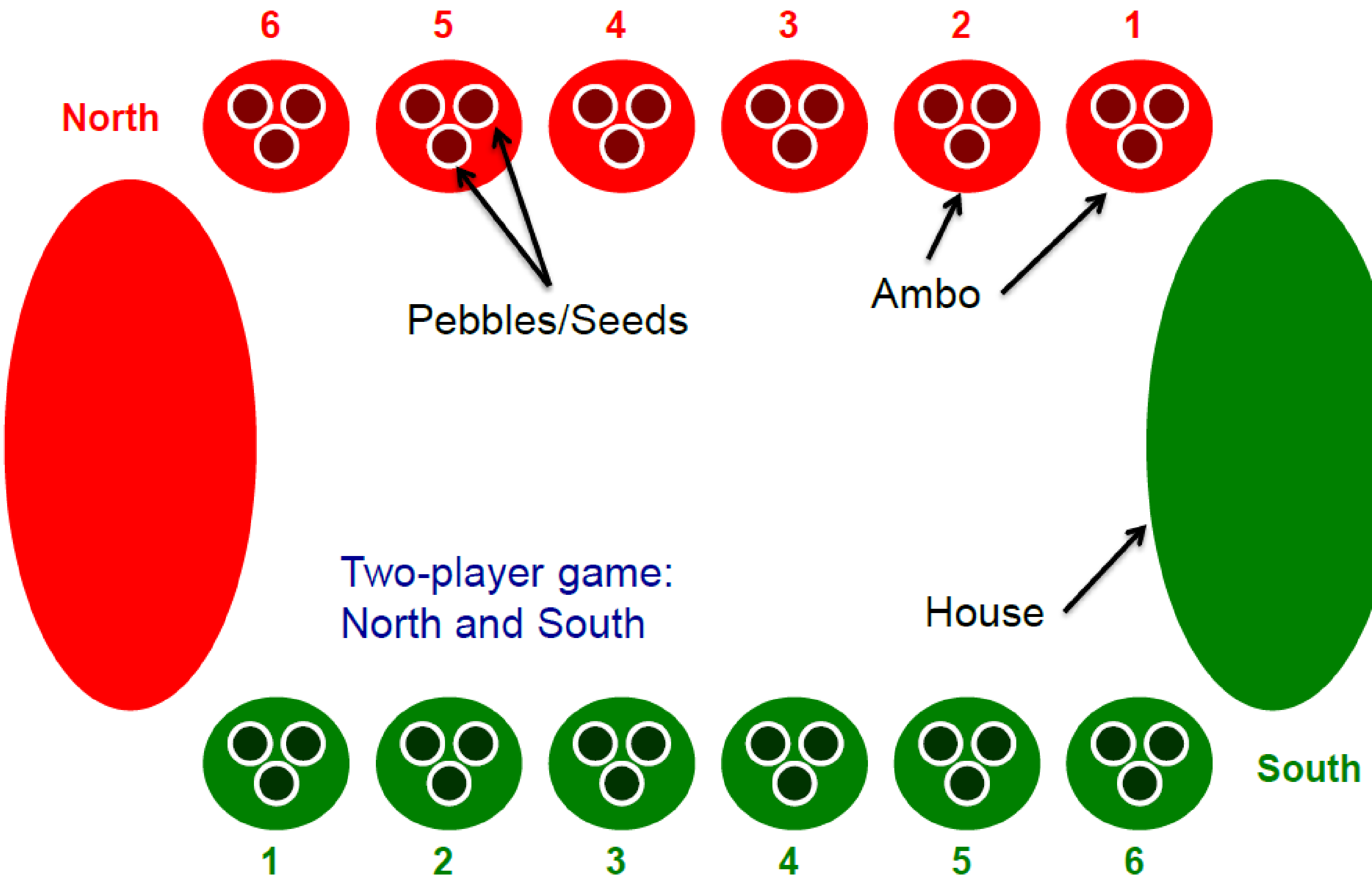
Assignment - Kalaha

APPLIED ARTIFICIAL INTELLIGENCE (DV2557)

1. Overview – Game



1. Overview – Grading

- Grade E: Minimax with ***Depth-First Search*** to depth level (>4)
- Grade D: Minimax with ***Iterative Deepening*** and time limit (5 seconds)
- Grade C: Minimax with ***Depth-First Search*** to depth level (>4), plus ***Alpha-Beta Pruning***
- Grade B: Minimax with ***Iterative Deepening*** and time limit (5 seconds), plus ***Alpha-Beta Pruning***
- Grade A: As Grade B, but with an opening book that has stored >100 game states that are relevant for deciding the first move. This shall be used instead of Minimax for deciding the first move from the AI.

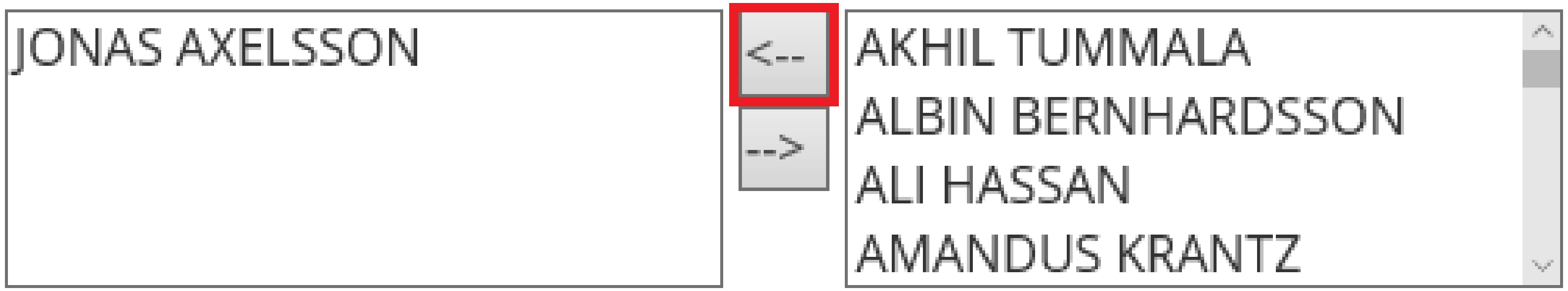
1. Overview – Submission

Assignment deadline: **1 October 201**7, 23:59

If you work in groups, note that:

Only one student needs to add the other group members to the it's learning submission page and submits the assignment solution.

Do not forget to add you group members before submitting!



2. Minimax

Minimax is a **recursive** method for finding optimal decisions. It is recommended for smaller game trees.

It needs:

- ***Complete information*** (Nothing in the game is hidden from any player)
- ***Optimal decisions*** (Players strive to win, making the best moves)
- ***Clear terminal states*** (The game ends after a certain goal is reached)

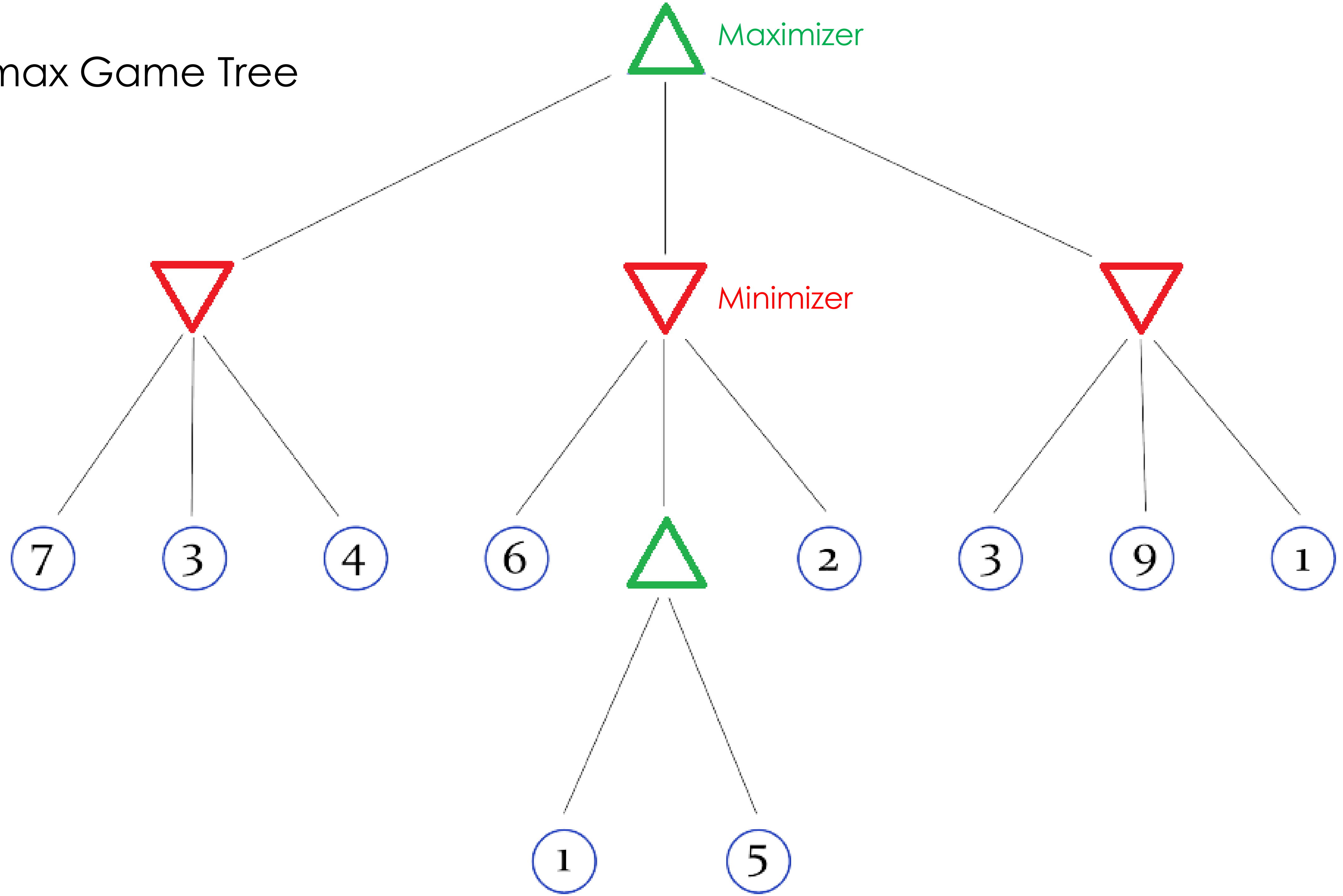
2. Minimax

Minimax works in the following way:

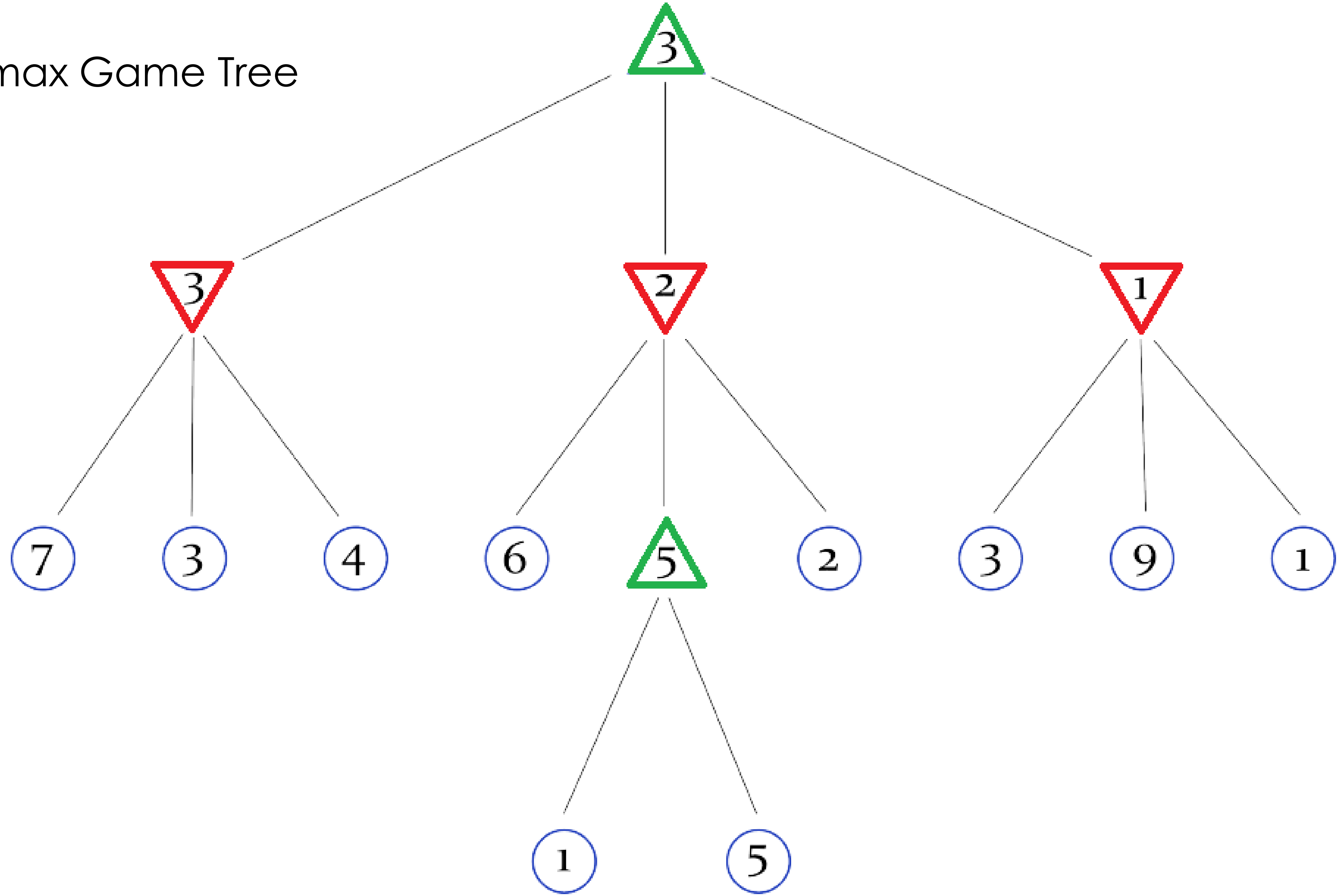
1. Expand through the ***game tree*** from an ***initial state***
2. When it reaches ***terminal nodes*** (sometimes called leaves), they are assigned a ***utility value*** depending on how "good" the node is
3. Propagate utility values upwards in the tree, using a ***Minimizer*** or a ***Maximizer*** (hence Minimax)
4. Once a utility value reaches the top node (initial state), we know the best path, and can return the best option

Easier to understand with an example!

Minimax Game Tree



Minimax Game Tree



2. Minimax

Elements of the minimax method:

- An ***Initial State*** (S_0)
- ***Player*** (who has the move in a state)
- ***Actions*** (actions available in a specific state)
- ***Result*** (how will a state look after a specific action is made)
- ***Terminal Test*** (checks if the game is over or not)
- ***Utility Function*** (Calculates the utility value of a given state and player)

2. Minimax

Algorithm example:

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

Taken from: (Russell et al., 2010) Artificial Intelligence a Modern Approach, 3rd edition pg. 164

Issues

Lets think about the Kalaha game tree for a while:

Most nodes have ***6 actions*** (pick an ambo from 1-6)

This means that by depth level 5, the tree could already contain ***9330*** nodes…

How will we traverse this tree efficiently?

Issues

The assignment proposes these two methods:

Depth-First Search and ***Iterative Deepening Search***

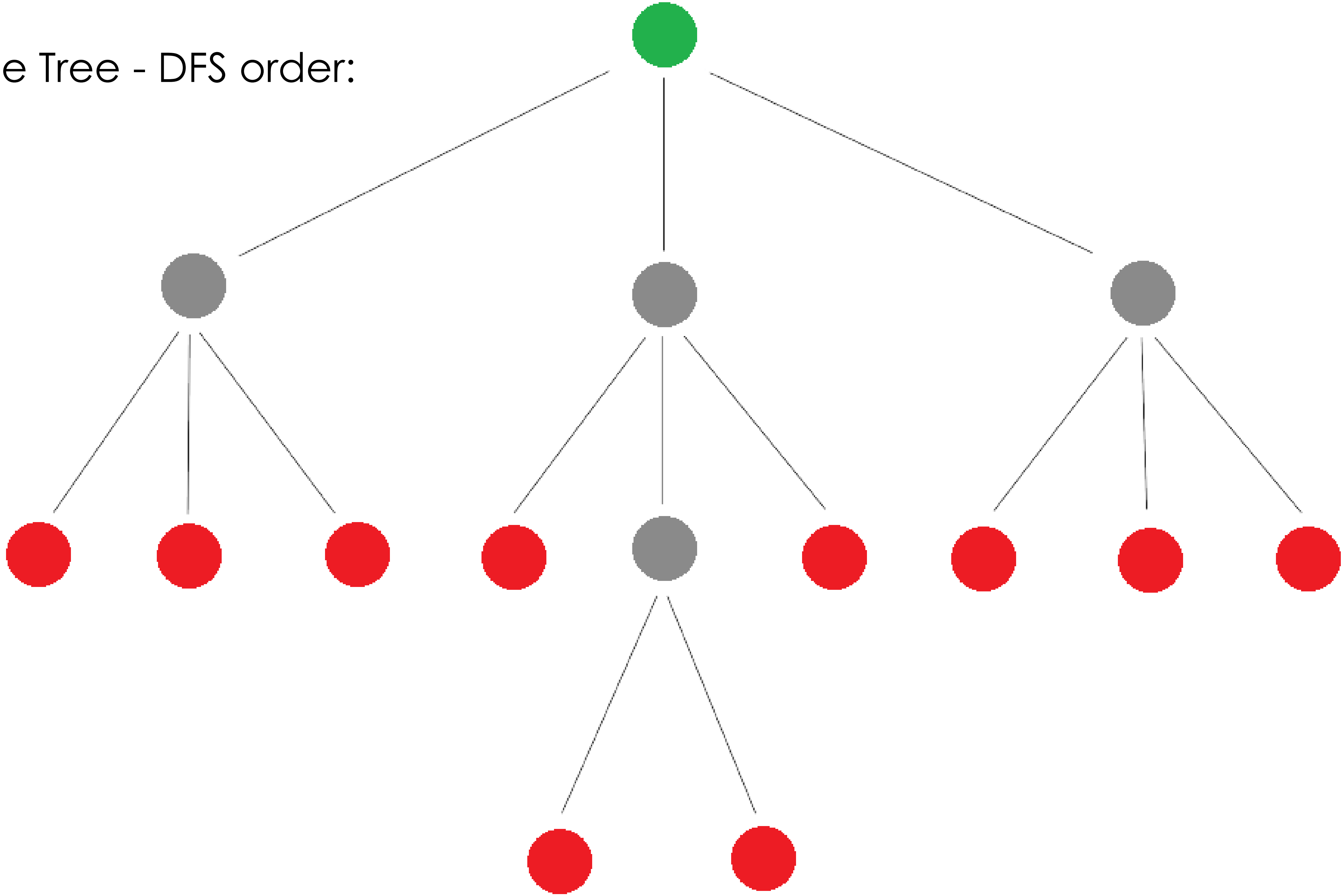
3. Depth-First Search

Depth-First Search (DFS):

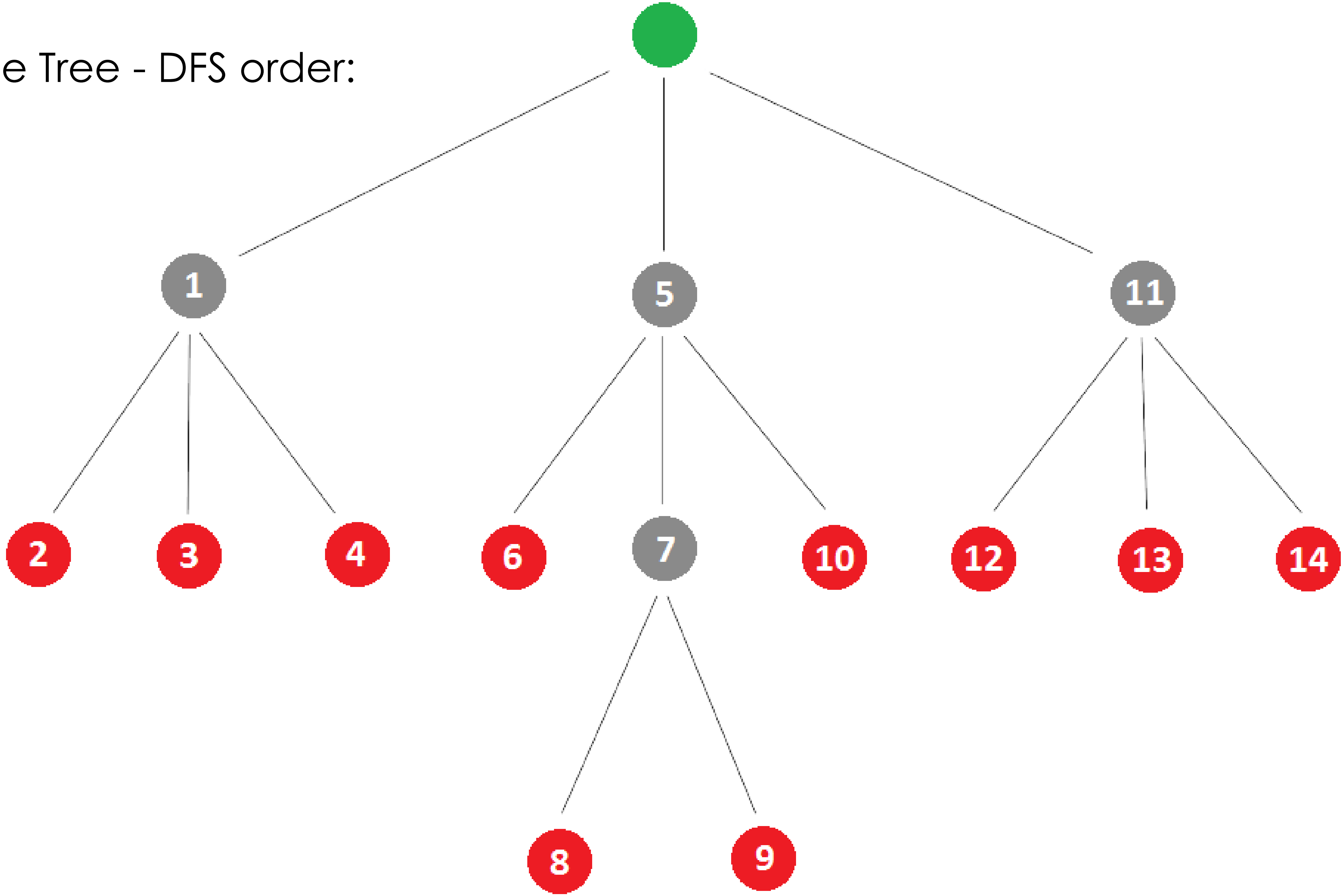
- Always expands from the deepest node.
- Uses a ***LIFO*** stack (Last In First Out), the latest node is expanded from.
- Commonly implemented with a ***recursive*** function.

Better understood with an example:

Game Tree - DFS order:



Game Tree - DFS order:



4. Iterative Deepening Search

Iterative Deepening Search (IDS):

- Combination of ***Depth-First*** and ***Breadth-First*** search.
- ***Gradual*** increase in depth level until a terminal node is found.
- The preferred method for an uniformed search when the space is large and the depth is unknown.

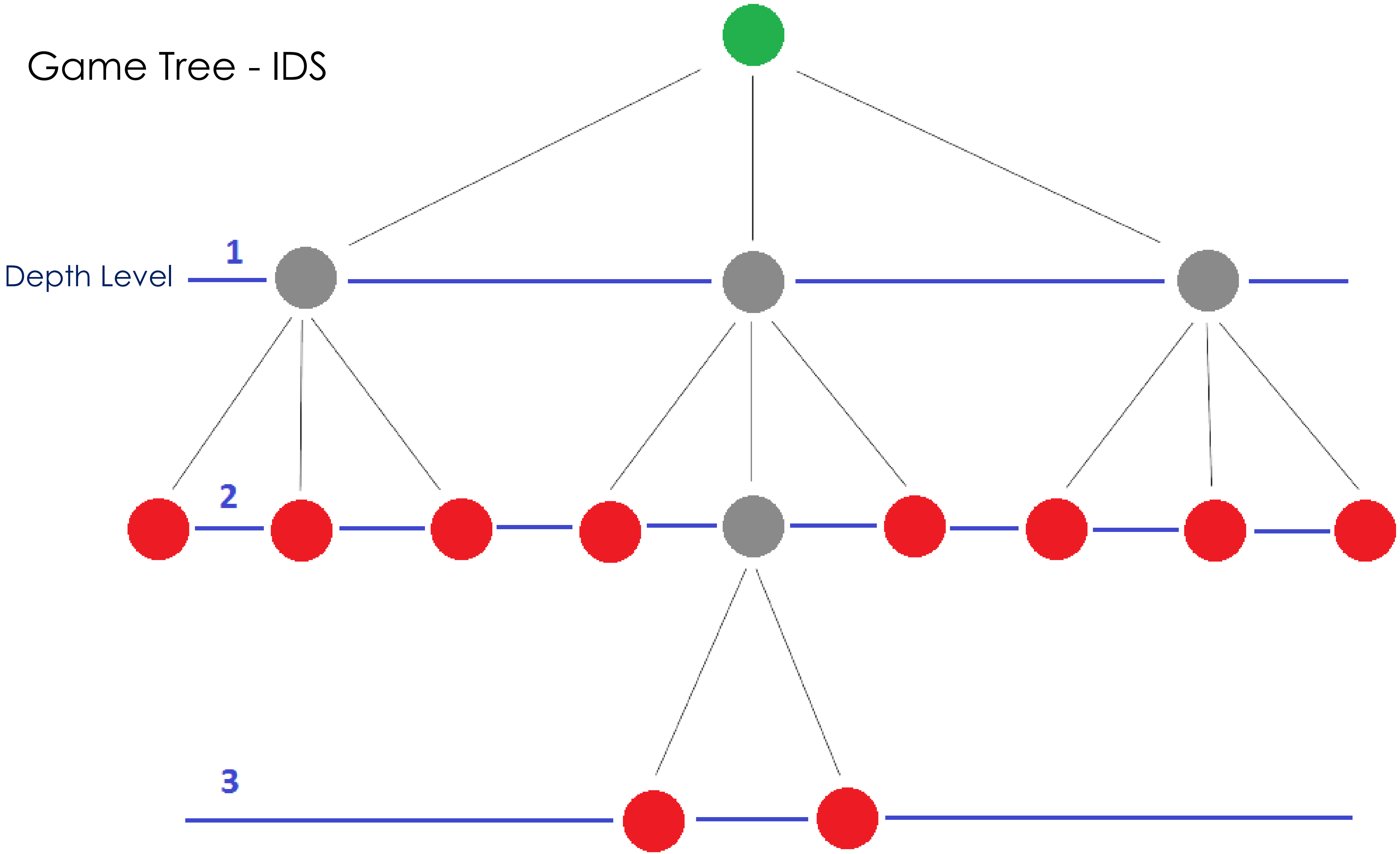
4. Iterative Deepening Search

How does it work:

1. Choose a ***depth level***.
2. Get the next node from the stack.
3. Is the node ***Terminal***? If so, set its ***utility value*** and get the next node.
4. Is the node at the ***depth level***?
   - Yes: Get the next node.
   - No: Expand and add new nodes to the stack.

If no solution is found, increase the depth level and repeat.

Game Tree - IDS



5. Alpha-Beta Pruning

Is there a better way?

Yes! We use ***Alpha-Beta Pruning***.

This method returns the same decision as Minimax would, but ***prunes*** branches of the tree would not have influenced the final decision while searching.

This reduces the amount of nodes we will have to search!

5. Alpha-Beta Pruning

Node components:

- ***Alpha value***: Best discovered path to the root of the maximizer.
- ***Beta value***: Best discovered path to the root of the minimizer.

Rules:

- Non terminal nodes are initialized with the worst possible value (depends on if the node is a maximizer or a minimizer) and then we expand.
- Terminal nodes are given the utility value as regular.

5. Alpha-Beta Pruning

When we retrieve a utility value:

The Maximizer:

1. If the retrieved value is higher than ***Alpha***, we update the ***Alpha***.
2. If the current value is higher than ***Beta***, we prune!

The Minimizer:

1. If the retrieved value is lower than ***Beta***, we update the ***Beta***.
2. If the current value is lower than ***Alpha***, we prune!

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

---

function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), 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*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

Taken from: (Russell et al., 2010) Artificial Intelligence a Modern Approach, 3rd edition pg. 170

5. Alpha-Beta Pruning

Can be difficult to understand at first glance, and is hard to explain.

For an in-depth example, try:

https://www.youtube.com/watch?v=xBXHtz4Gbdo

6. Assignment Code

The assignment is composed of 4 packages and 10 classes:

- ***AI Package***
  - ***AIClient.java***: This is where the Minimax will be implemented.
- ***Client Package***
  - ***BadClient.java:*** Always makes the worst possible action.
  - ***HumanClient.java:*** Enables the GUI to control the game.
  - ***RandomClient.java:*** Selects a random action.

6. Assignment Code

- ***Kalaha Package***
  - ***Commands.java:*** Strings that can be sent to and from the server.

    [Move, Hello, Board, Player, New, Winner]
  - ***Errors.java:*** Errors that can be returned from the server.

    [Game Full, Game not full, Command not found, Invalid parameters, Invalid move, Wrong player, Ambo empty]
  - ***GameState.java:*** Represents the Kalaha board.
  - ***KalahaMain.java:*** Starts the application and GUI.

6. Assignment Code

- ***Server Package***
  - ***KalahaServer.java:*** Creates new Kalaha server.
  - ***ServerGUI.java:*** Creates and controls the GUI for the server.

7. Useful Functions

Mostly you will be working in ***AIClient.java*** class, but you will find many useful functions in the ***GameState.java*** class:

- ***GameState():*** creates a start game state.
- ***GameState(int[ ] board, int nextPlayer):*** creates a game state from a board representation.
- ***GameState(String boardStr):*** creates a state from a string representation (server).
- ***clone():*** Creates a copy of a game state and returns a new GameState object.

7. Useful Functions

- ***createBoard():*** Creates a start game state with an specific number of seeds (6 by default).
- ***makeMove(int ambo):*** checks if the action is valid, sows pebbles in specific ambo and updates the game state.
- ***getSeeds(int ambo, int player):*** Retrieve the number of seeds from and specific ambo and player.
- ***gameEnded():*** Checks if the game is over or not.
- ***getWinner():*** Check if there is a winner, a tie or the game is still running.
- ***getScore(int player):*** Returns the score from an specific player.

Good Luck!