Algorithme *Minimax* et élagage αβ

Gauthier Picard

SMA/G2I/ENS Mines Saint-Etienne

gauthier.picard@emse.fr

9 novembre 2011

Sommaire

Introduction

Contexte

- Basé sur le théorème du *Minimax* de von Neumann
- Règle de décision pour les jeux à somme nulle, à l'origine
- Objectif : *mini*miser la perte potentielle *maxi*male (ou l'inverse)
- Initialement formulé pour des jeux à somme nulle à deux joueurs jouant alternativement
- Des extensions existent pour couvrir des jeux plus complexes également

Introduction

Contexte

- Basé sur le théorème du *Minimax* de von Neumann
- Règle de décision pour les jeux à somme nulle, à l'origine
- Objectif : *mini*miser la perte potentielle *maxi*male (ou l'inverse)
- Initialement formulé pour des jeux à somme nulle à deux joueurs jouant alternativement
- Des extensions existent pour couvrir des jeux plus complexes également

Objectifs

1. Implémenter un algorithme *Minimax* pour un jeu bien connu

Introduction

Contexte

- Basé sur le théorème du *Minimax* de von Neumann
- Règle de décision pour les jeux à somme nulle, à l'origine
- Objectif : *mini*miser la perte potentielle *maxi*male (ou l'inverse)
- Initialement formulé pour des jeux à somme nulle à deux joueurs jouant alternativement
- Des extensions existent pour couvrir des jeux plus complexes également

Objectifs

1. Implémenter un algorithme *Minimax* pour un jeu bien connu
2. Adapter l'algorithme pour permettre à une machine de jouer contre un joueur humain

Introduction

Contexte

- Basé sur le théorème du *Minimax* de von Neumann
- Règle de décision pour les jeux à somme nulle, à l'origine
- Objectif : *mini*miser la perte potentielle *maxi*male (ou l'inverse)
- Initialement formulé pour des jeux à somme nulle à deux joueurs jouant alternativement
- Des extensions existent pour couvrir des jeux plus complexes également

Objectifs

1. Implémenter un algorithme *Minimax* pour un jeu bien connu
2. Adapter l'algorithme pour permettre à une machine de jouer contre un joueur humain
3. Améliorer l'algorithme grâce à un élagage αβ

Théorème du *Minimax*

Théorème

Pour tout jeu à deux joueurs, à somme nulle, avec un nombre fini de stratégies, il existe une valeur V et une stratégie mixte pour chaque joueur telle que

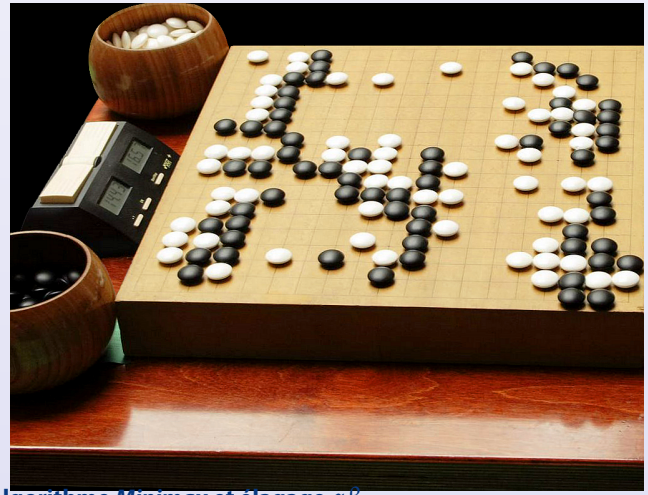
(a) étant donnée la stratégie du joueur 2, le meilleur gain possible pour le joueur A est V, et

(b) étant donnée la stratégie du joueur B, le meilleur gain possible pour le joueur est $-V$.

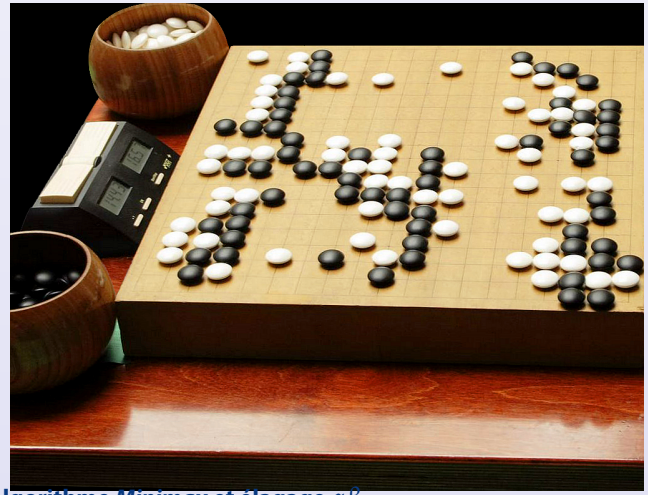
Remarques

- On est capable de borner ses propres bénéfices et ceux de l'adversaire
- Chaque joueur minimise le gain maximum de l'adversaire, et comme le jeu est à somme nulle, il maximise également son gain minimum

Exemples



Exemples



Algorithme *Minimax*

Survol

- Théorie des jeux combinatoires (actions, sommes, choix finis)
- Mouvements alternés (un joueur après l'autre, jusqu'à la fin du jeu)
- A chaque coup, les joueurs tentent de maximiser leur gain minimum et donc de minimiser le gain maximal de leur adversaire
- On appelle *Minimax* à la fois l'algorithme et la valeur obtenue par exécution de cet algorithme

Algorithme *Minimax*

Survol

- Théorie des jeux combinatoires (actions, sommes, choix finis)
- Mouvements alternés (un joueur après l'autre, jusqu'à la fin du jeu)
- A chaque coup, les joueurs tentent de maximiser leur gain minimum et donc de minimiser le gain maximal de leur adversaire
- On appelle *Minimax* à la fois l'algorithme et la valeur obtenue par exécution de cet algorithme

→ L'algorithme Minimax est utilisé pour décider du prochain coup à jouer

- L'ordinateur va calculer la valeur *Minimax* de chaque coup possible à un instant donné et choisir celui qui maximise le gain minimum

Algorithme *Minimax*

Quelques définitions

- Le joueur *A* est appelé le *joueur maximisant* (ou *joueur*), le joueur *B* est appelé le *joueur minimisant* (ou *adversaire*)
  - par ex : l'ordinateur est le joueur maximisant puisqu'il va exécuter l'algorithme *Minimax* afin de jouer contre le joueur humain
- On appelle *état* du jeu une configuration du jeu
  - par ex : l'état d'un jeu d'échec est l'enregistrement de toutes les positions des pièces sur l'échiquier
- On appelle *état initial* l'état de jeu avant que les joueurs aient joué, et *état final* un état mettant fin au jeu (gagné, perdu, égalité)
- On appelle *gain d'un état* la valeur gagnée par un joueur s'il atteint l'état donné
- On appelle *coup* une action permettant de passer le jeu d'un état à un autre
  - par ex : déplacer un pion de *X* en *Y*
- On appelle *fils d'un état* e_i, notés $f(e_i)$ les états atteignables depuis l'état e_i

  → exploration d'un *arbre de jeu*

Algorithme *Minimax*

Heuristique

- Pour déterminer les gains, on définit une fonction *heuristique*, notée h, qui attribue une valeur à un état du jeu
  - empirique (donc souvent la partie la plus complexe à définir)
  - non générique (à définir pour chaque jeu)
- Comment interpréter une heuristique ?
  - « C'est la valeur que l'on peut espérer gagner en jouant le coup »
- Quand utiliser une heuristique ?
  - impossible de déterminer les gains réels d'un état
  - trop coûteux de calculer les gains d'un état (combinatoire)
- Comment définir une heuristique ?
  - Si l'état e_i est gagnant pour le joueur évaluant, $h(e_i) = +\infty$
  - Si l'état e_i est perdant pour le joueur évaluant, $h(e_i) = -\infty$
  - Sinon, donner une valeur entière déterminée de manière empirique
    - par ex : aux échecs une heuristique simpliste consiste à calculer la différence entre le nombre de pièces blanches et le nombre de pièces noires

Algorithme *Minimax*

Principe

- On visite l'arbre de jeu pour faire remonter à la racine la valeur *Minimax*
- La valeur est calculée récursivement comme suit
  - $Minimax(e) = h(e)$, si e est une feuille de l'arbre
  - $Minimax(e) = max(Minimax(e_1), \ldots, Minimax(e_n))$, si e est un nœud Joueur (maximisant) avec les fils $f(e) = \{e_1, \ldots, e_n\}$
  - $Minimax(e) = min(Minimax(e_1), \ldots, Minimax(e_n))$, si e est un nœud adversaire (minimisant) avec les fils $f(e) = \{e_1, \ldots, e_n\}$

Profondeur limite

- Pour limiter les calculs, on définit une *profondeur limite* au calcul (ou *horizon*)
- Les nœuds ne sont plus développés à partir de cette profondeur, on calcule alors leur valeur heuristique
- une *feuille de l'arbre* est donc soit un état final (gagné, perdu, égalité), soit un état non final mais à une profondeur limite

Algorithme *Minimax*

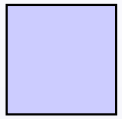
Fonction $Minimax(e, d)$

Entrées : nœud e, profondeur d

Sorties : Valeur Minimax du nœud e

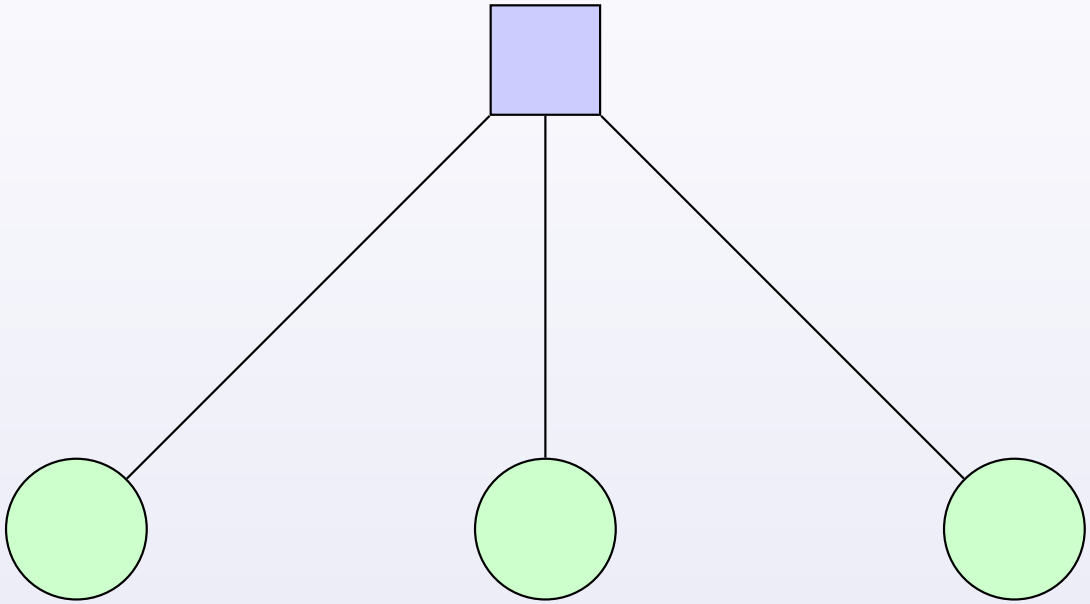
si *final ?(e) ou (d == 0)* **alors**
  return $h(e)$
sinon
  si *joueur ?(e)* **alors**
    return $max\{Minimax(e_i, d-1) \mid e_i \in f(e)\}$
  sinon
    return $min\{Minimax(e_i, d-1) \mid e_i \in f(e)\}$

Algorithme *Minimax*



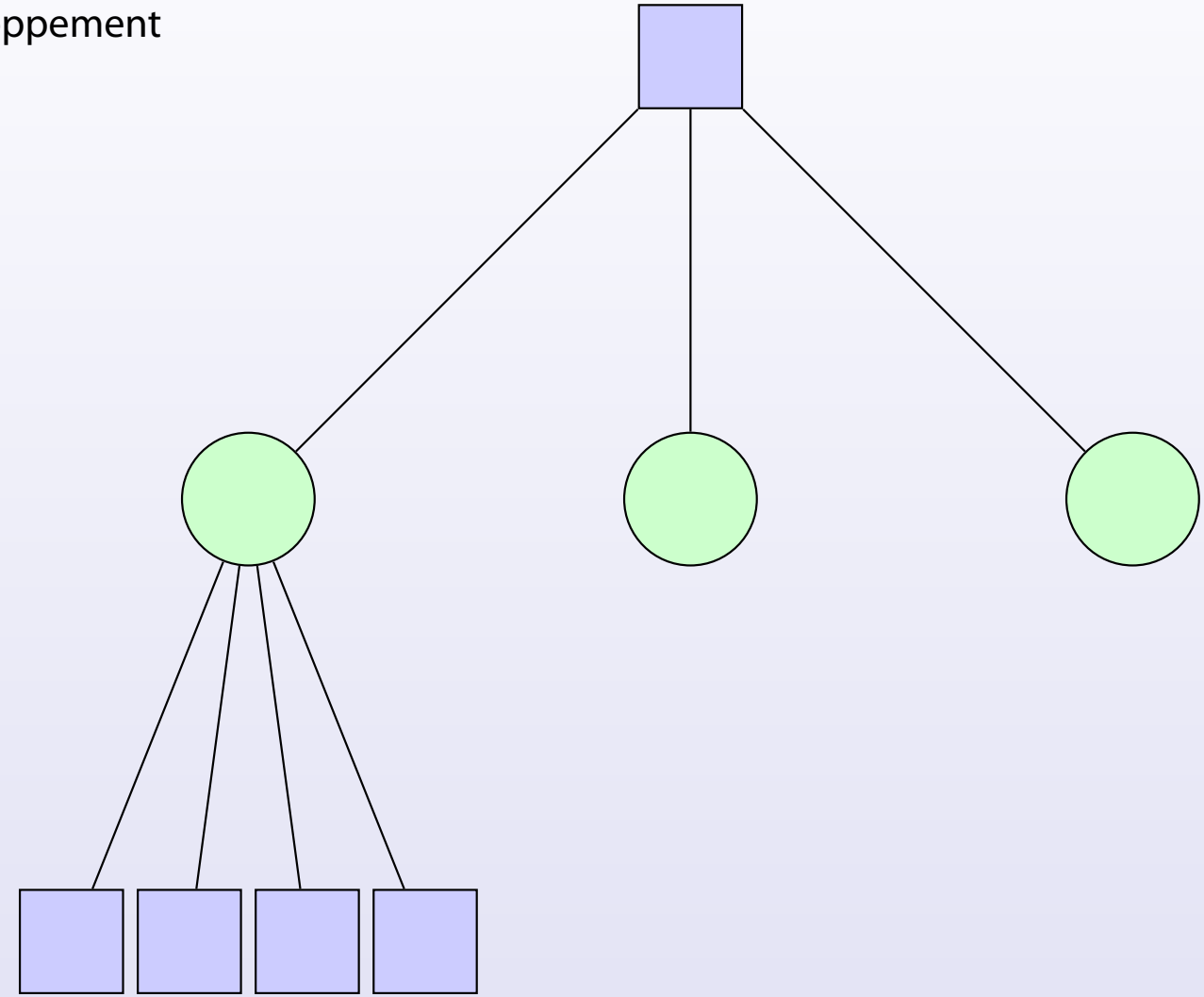
Algorithme *Minimax*

Développement
$f(e)$



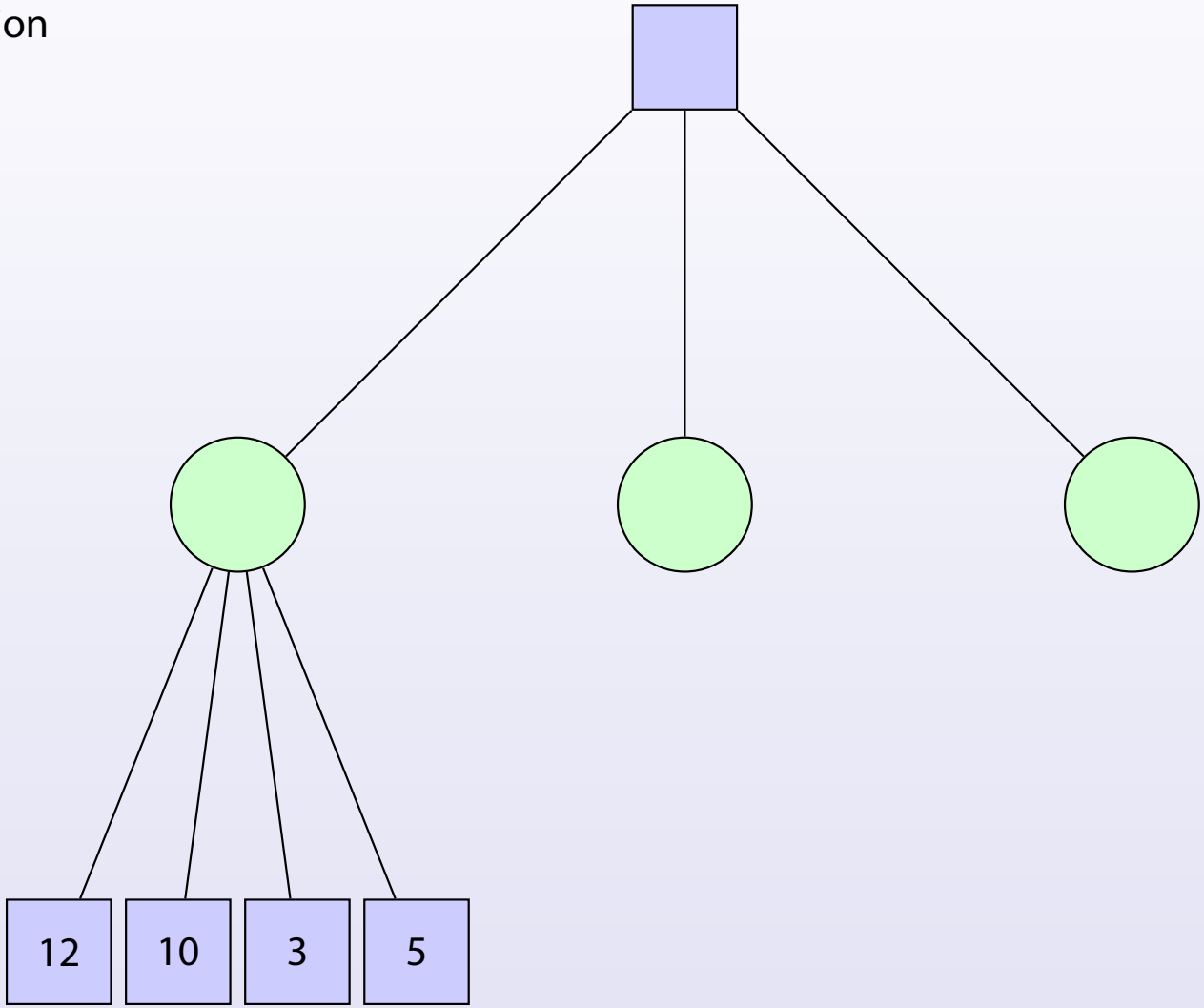
Algorithme *Minimax*

Développement
$f(e)$



Algorithme *Minimax*

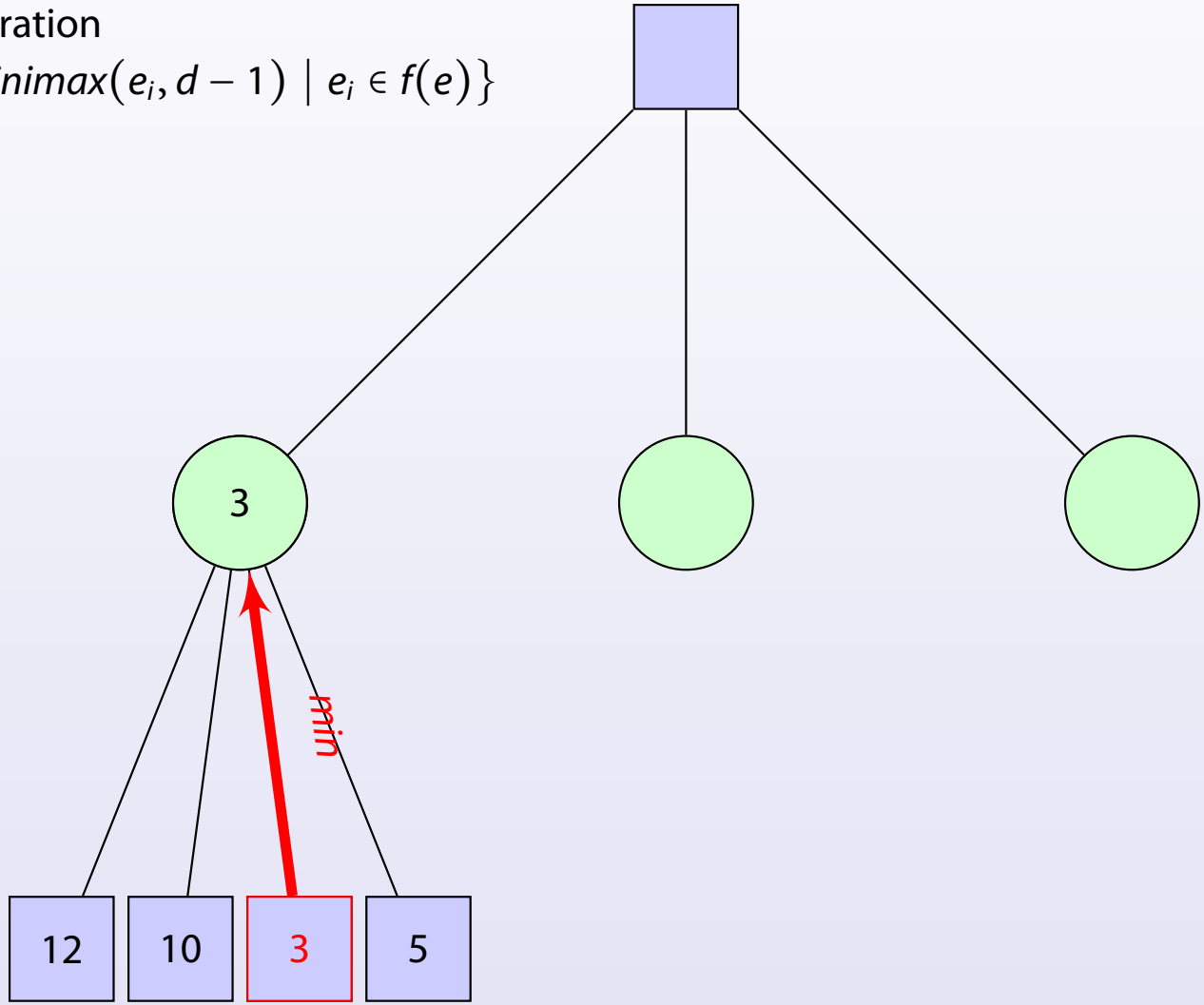
Evaluation
$h(e)$



Algorithme *Minimax*

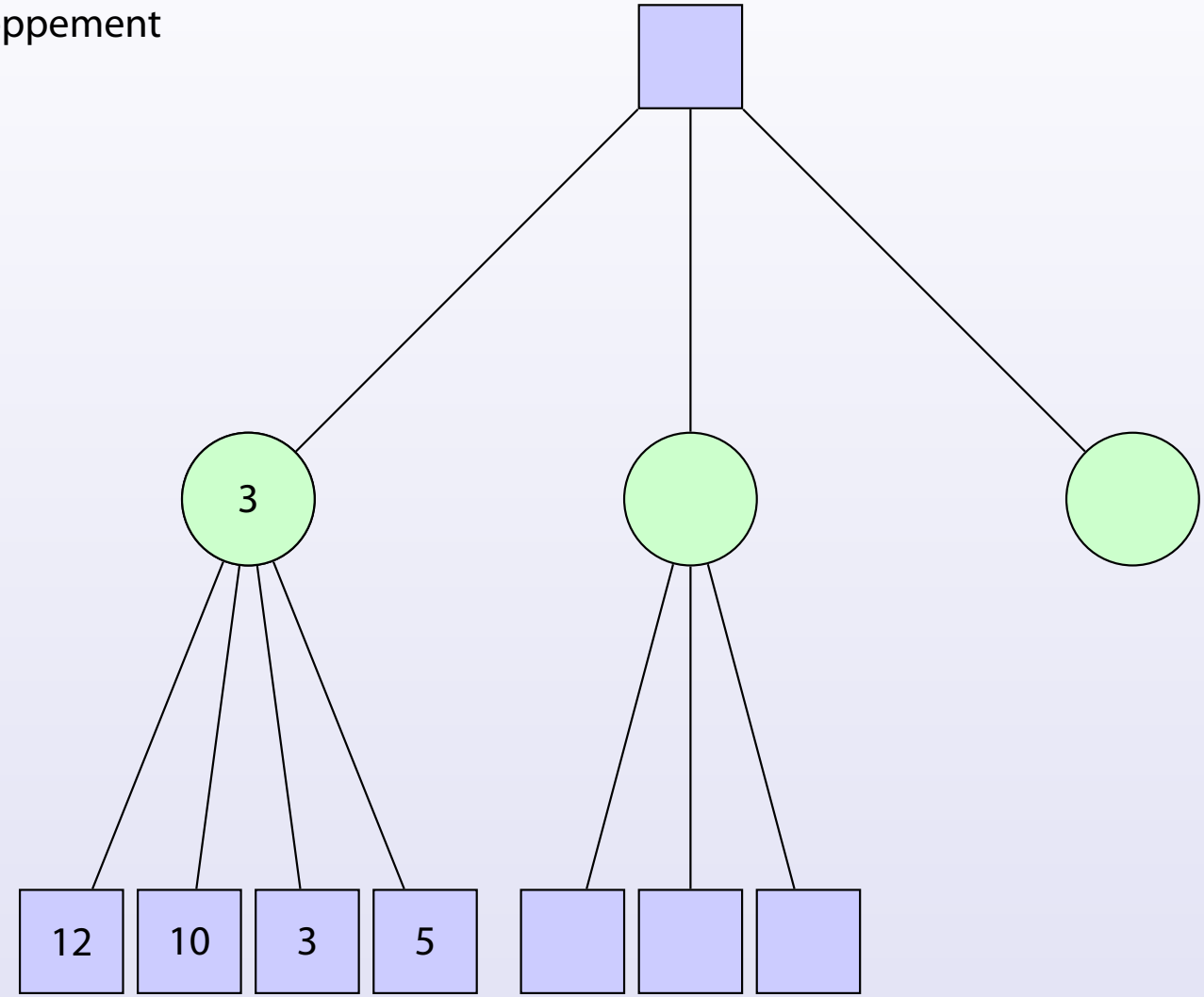
Récupération

$min\{Minimax(e_i, d-1) \mid e_i \in f(e)\}$



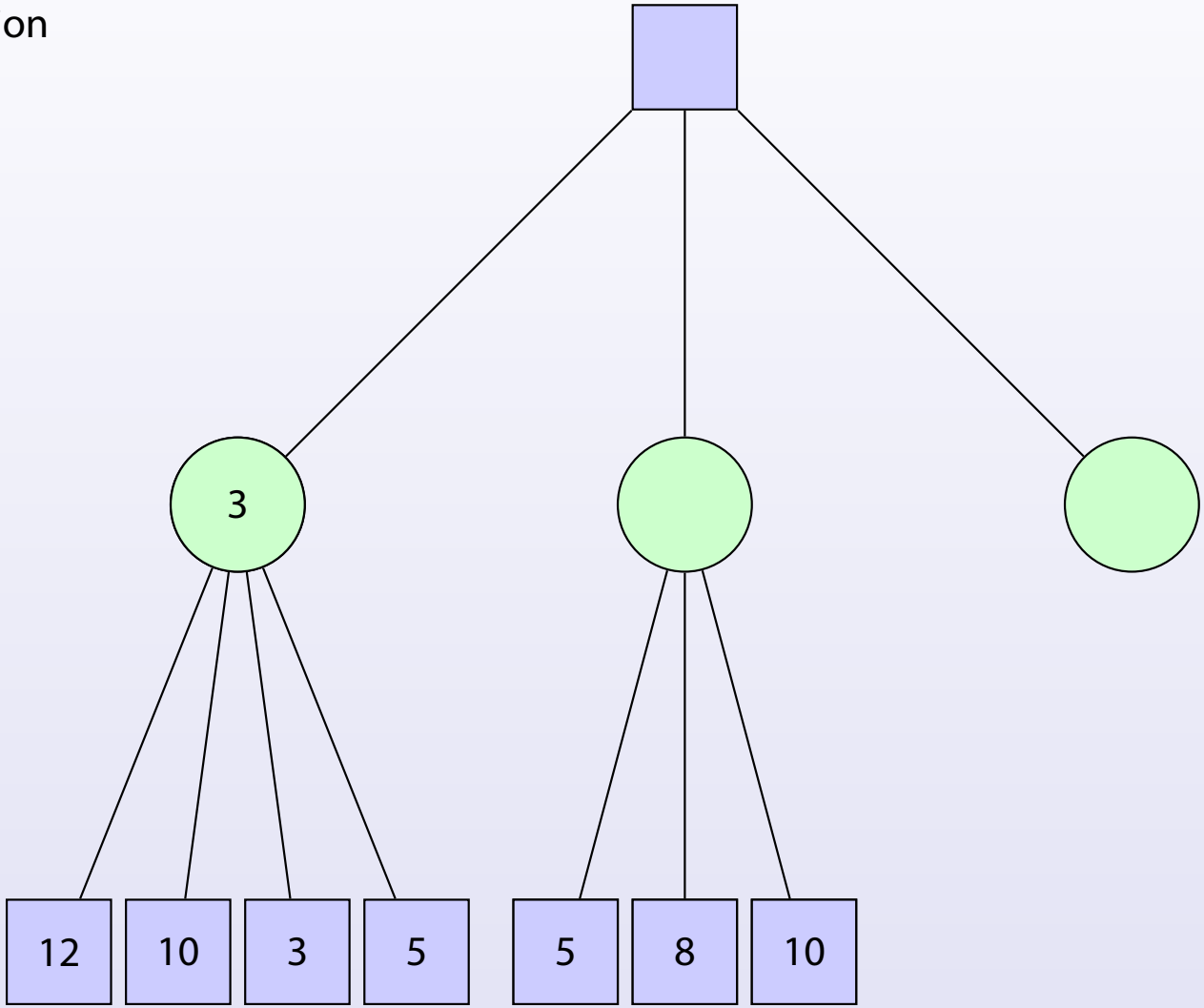
Algorithme *Minimax*

Développement
$f(e)$



Algorithme *Minimax*

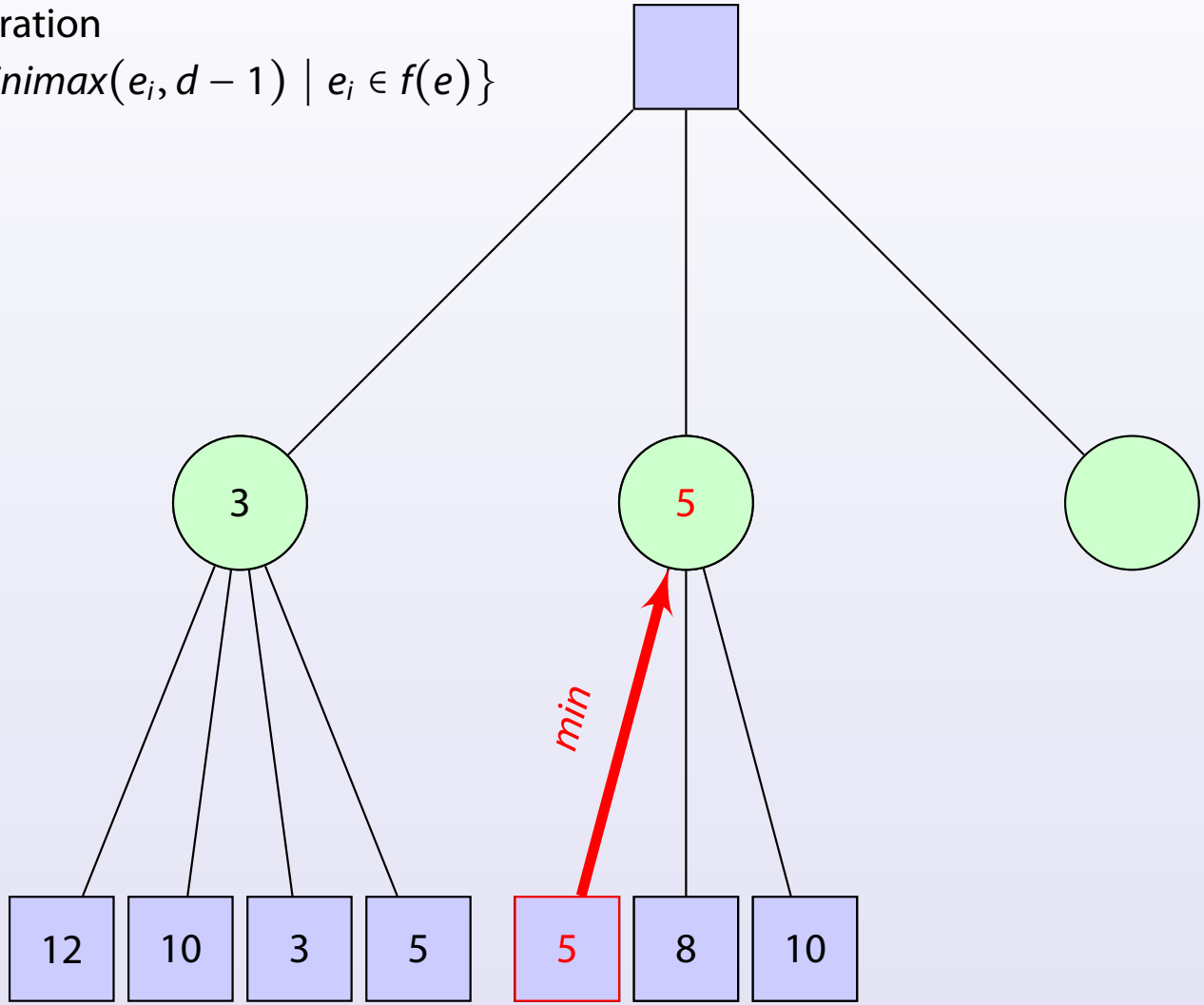
Evaluation
$h(e)$



Algorithme *Minimax*

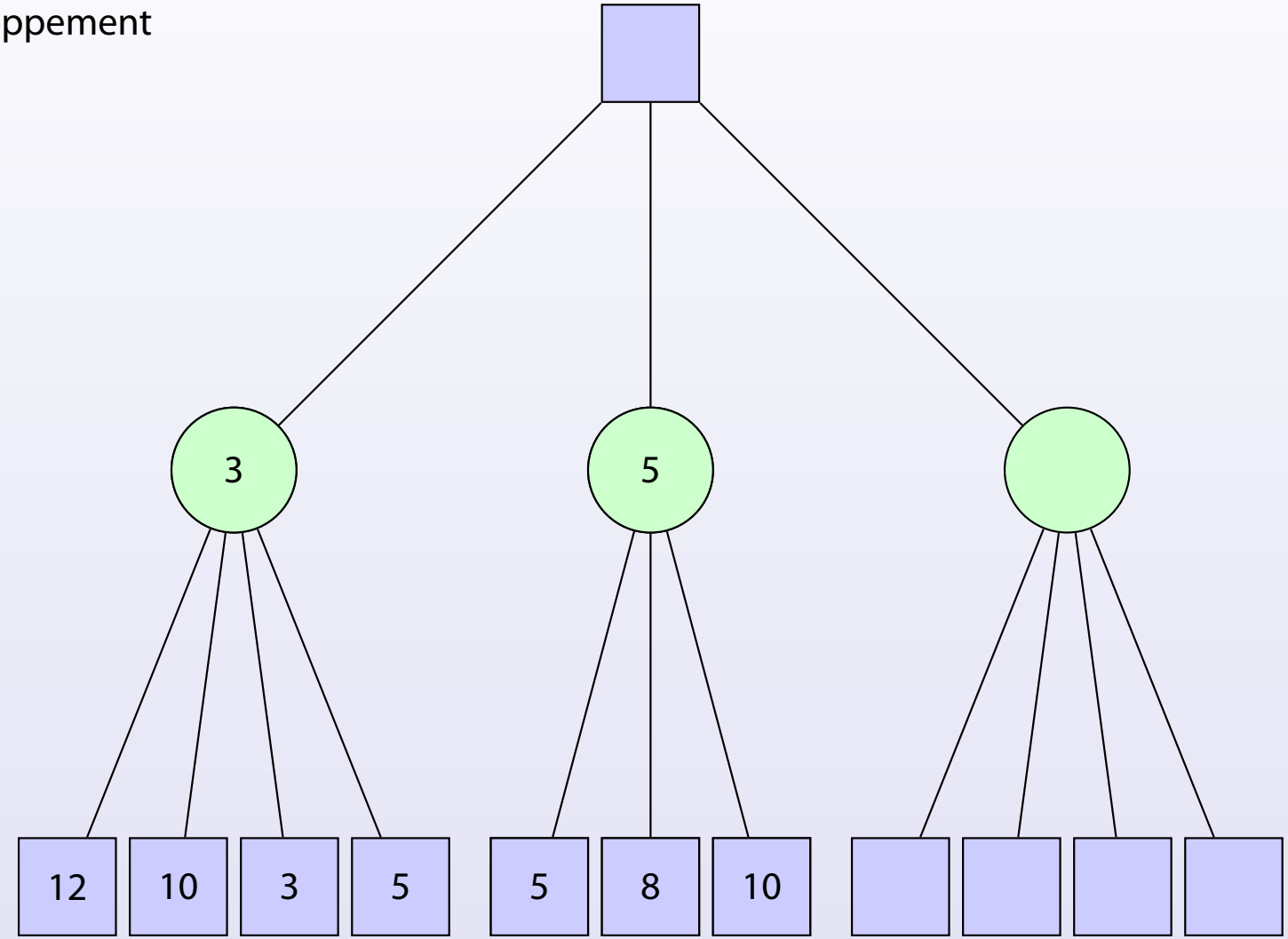
Récupération

$min\{Minimax(e_i, d-1) \mid e_i \in f(e)\}$



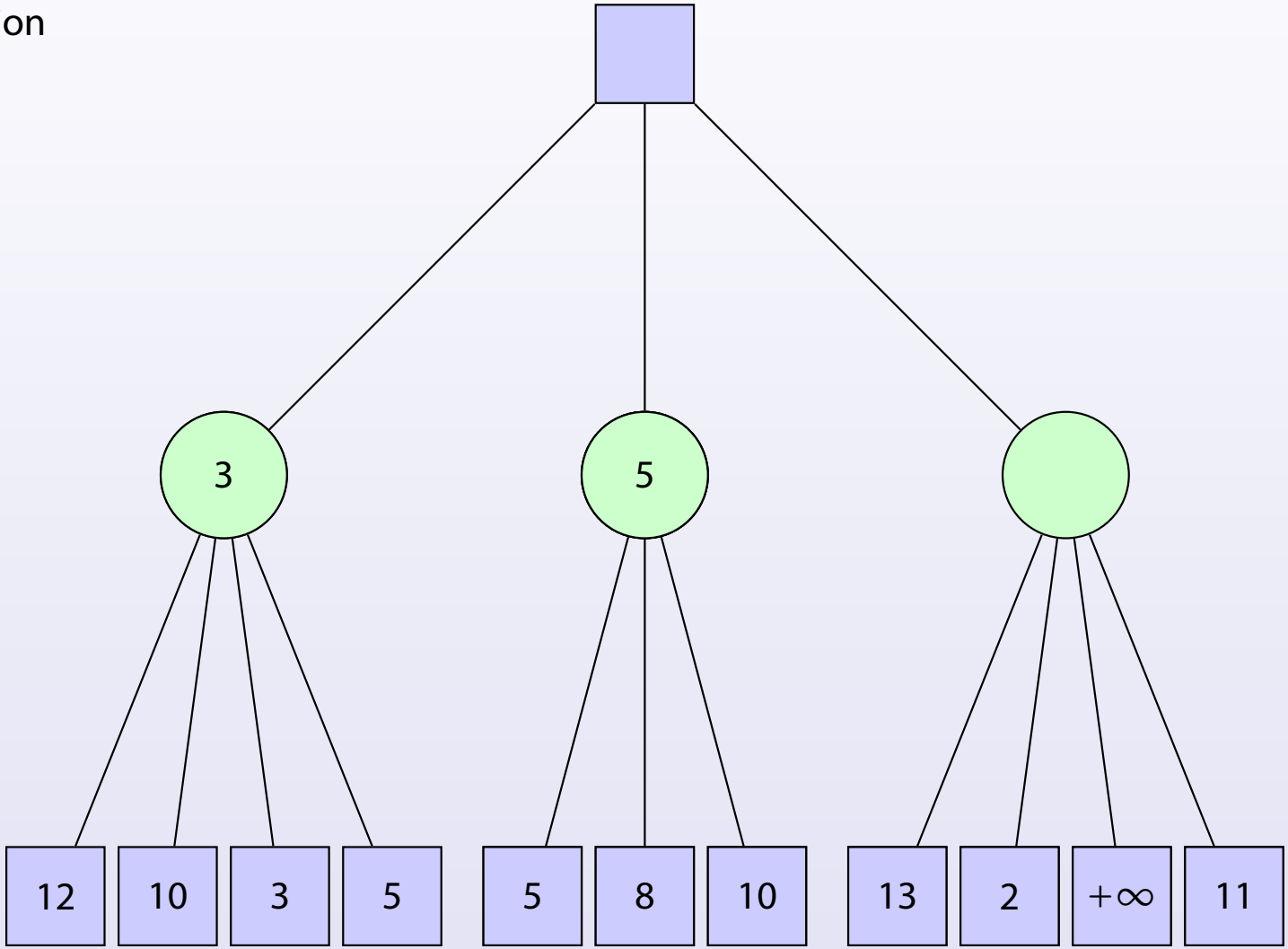
Algorithme *Minimax*

Développement
$f(e)$



Algorithme *Minimax*

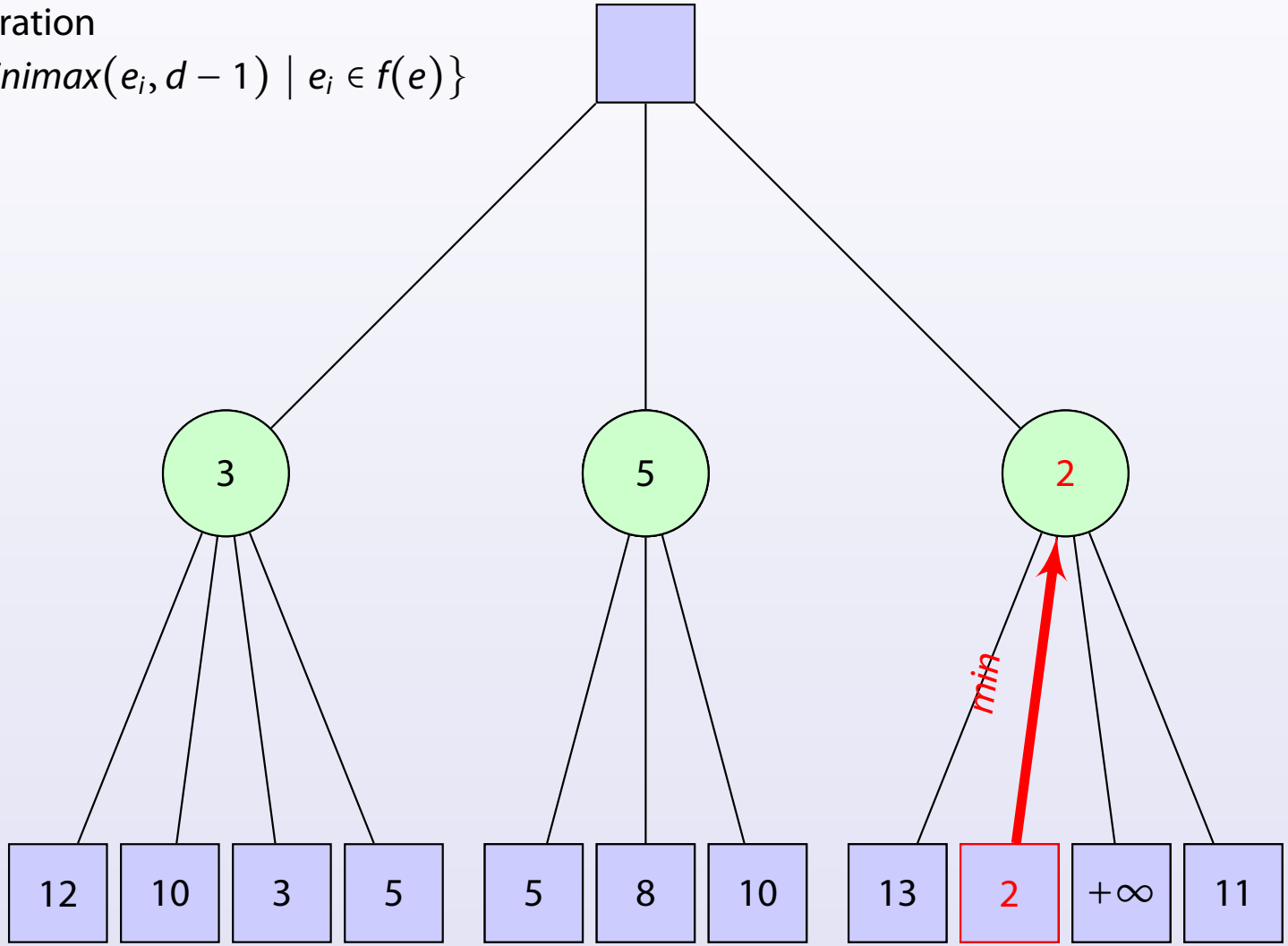
Evaluation
$h(e)$



Algorithme *Minimax*

Récupération

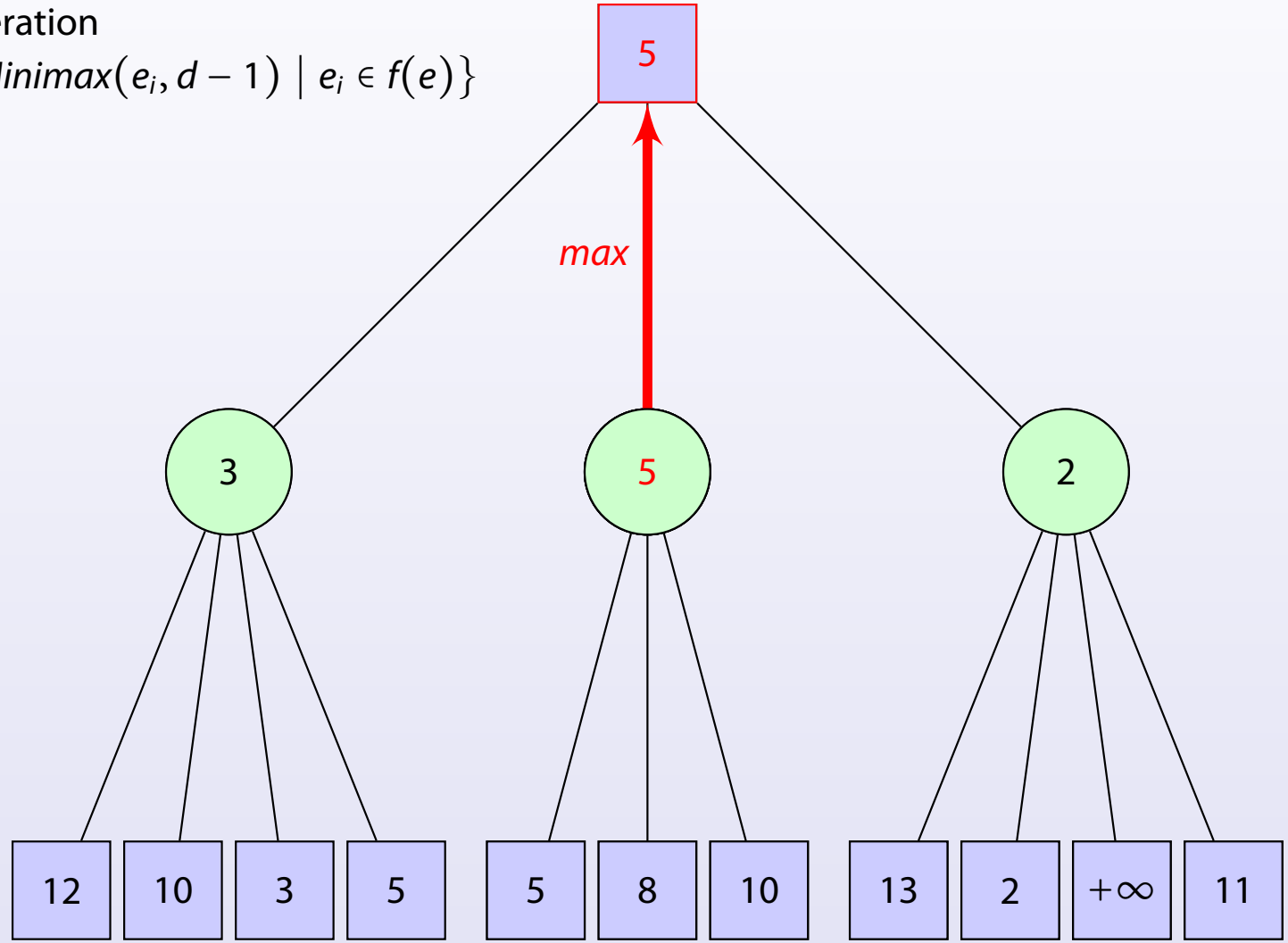
$min\{Minimax(e_i, d-1) \mid e_i \in f(e)\}$



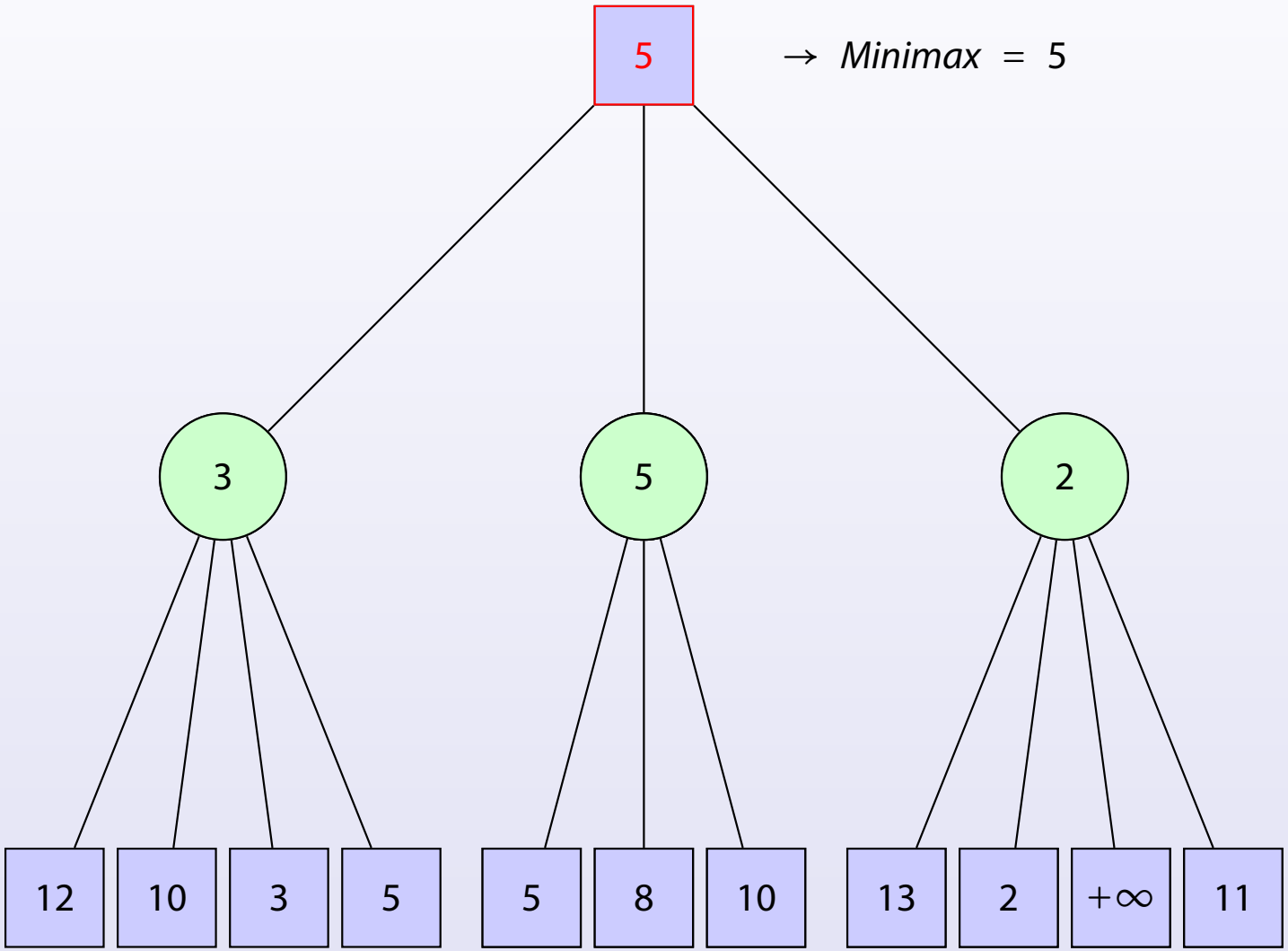
Algorithme *Minimax*

Récupération

$max\{Minimax(e_i, d-1) \mid e_i \in f(e)\}$



Algorithme *Minimax*



Élagage αβ

Problème

- L'algorithme *Minimax* développe **toutes** les feuilles à une profondeur limite
- Il arrive pourtant qu'on sache pertinemment qu'une feuille sera inintéressante

Solution : les bornes α et β

Pour élaguer certaines branches de l'arbre de jeu, on définit deux bornes, α et β

α est une approximation de la borne inférieure de la valeur du nœud

- $\alpha = h(e)$ sur les feuilles, et initialisée à $\alpha = -\infty$ ailleurs
- sur les nœuds *joueurs*, elle est maintenue égale à la plus grande valeur obtenue sur les fils visités jusque-là
- sur les nœuds *adversaires*, elle est égale à la valeur α de son prédécesseur

β est une approximation de la borne supérieure de la valeur du nœud

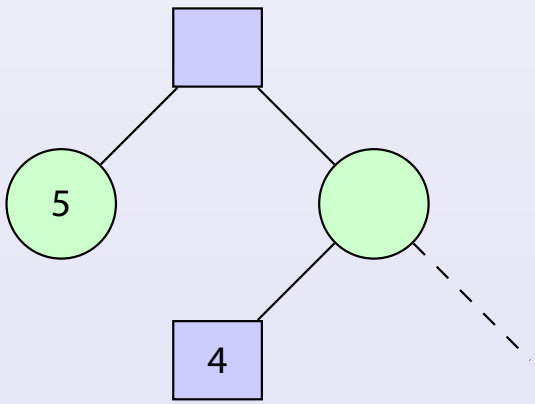
- $\beta = h(e)$ sur les feuilles, et initialisée à $\beta = +\infty$ ailleurs
- sur les nœuds *adversaires*, elle est maintenue égale à la plus petite valeur obtenue sur les fils visités jusque-là,
- sur les nœuds *joueurs* elle est égale à la valeur β de son prédécesseur

Élagage αβ

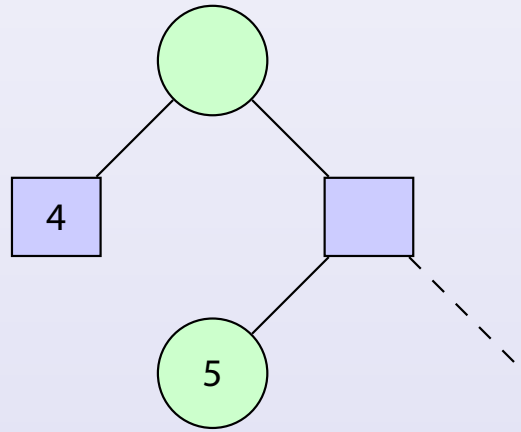
Principe

- L'algorithme est identique à *Minimax*, de signature $AlphaBeta(e, \alpha, \beta, d)$
- On ajoute les paramètres α et β (initialement $-\infty$ et $+\infty$)
- Les nœuds élagués sont ceux tels que $h(e) \in [\alpha, \beta]$ et $\alpha \geq \beta$
- Les nœuds non élagués sont ceux tels que :

$$[\alpha, \beta] = \begin{cases} [-\infty, +\infty] & \text{ou} \\ [-\infty, b] & \text{avec } b \neq +\infty \text{ ou} \\ [a, +\infty] & \text{avec } a \neq -\infty \end{cases}$$



Coupure α



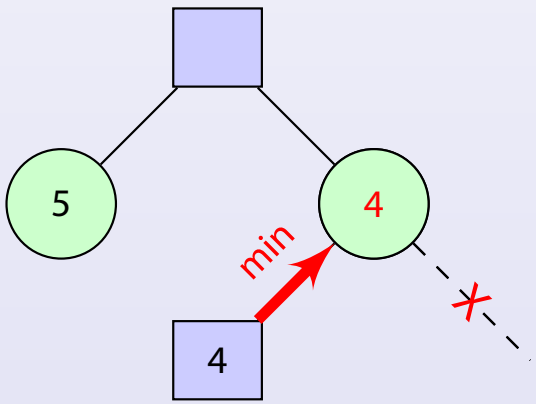
Coupure β

Élagage αβ

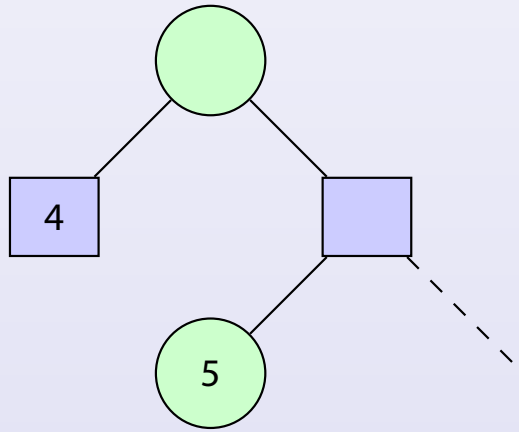
Principe

- L'algorithme est identique à *Minimax*, de signature $AlphaBeta(e, \alpha, \beta, d)$
- On ajoute les paramètres α et β (initialement $-\infty$ et $+\infty$)
- Les nœuds élagués sont ceux tels que $h(e) \in [\alpha, \beta]$ et $\alpha \geq \beta$
- Les nœuds non élagués sont ceux tels que :

$$[\alpha, \beta] = \begin{cases} [-\infty, +\infty] & \text{ou} \\ [-\infty, b] & \text{avec } b \neq +\infty \text{ ou} \\ [a, +\infty] & \text{avec } a \neq -\infty \end{cases}$$



Coupure α



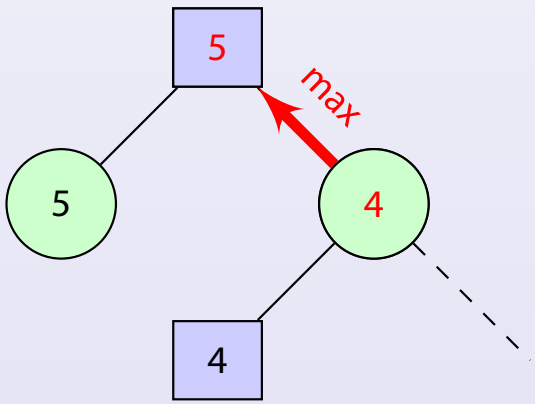
Coupure β

Élagage αβ

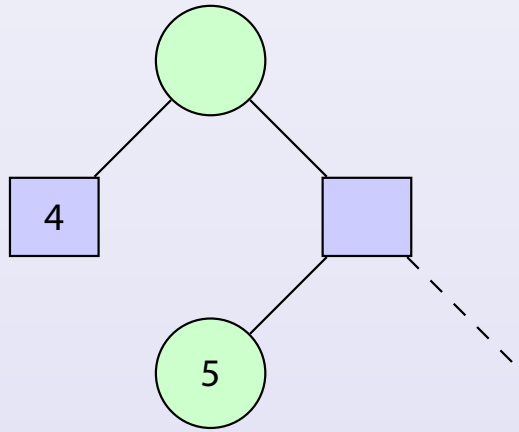
Principe

- L'algorithme est identique à *Minimax*, de signature $AlphaBeta(e, \alpha, \beta, d)$
- On ajoute les paramètres α et β (initialement $-\infty$ et $+\infty$)
- Les nœuds élagués sont ceux tels que $h(e) \in [\alpha, \beta]$ et $\alpha \geq \beta$
- Les nœuds non élagués sont ceux tels que :

$$[\alpha, \beta] = \begin{cases} [-\infty, +\infty] & \text{ou} \\ [-\infty, b] & \text{avec } b \neq +\infty \text{ ou} \\ [a, +\infty] & \text{avec } a \neq -\infty \end{cases}$$



Coupure α



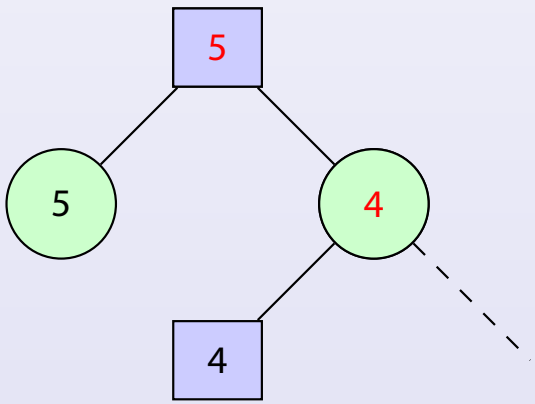
Coupure β

Élagage αβ

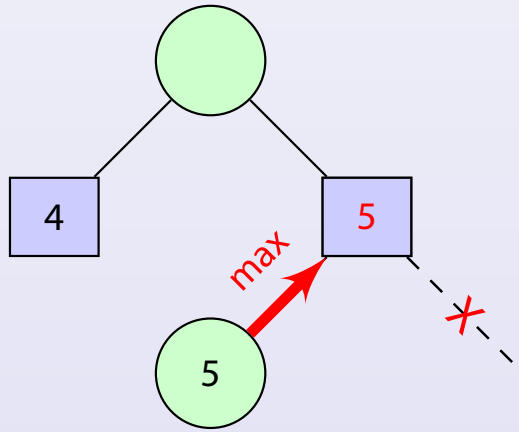
Principe

- L'algorithme est identique à *Minimax*, de signature $AlphaBeta(e, \alpha, \beta, d)$
- On ajoute les paramètres α et β (initialement $-\infty$ et $+\infty$)
- Les nœuds élagués sont ceux tels que $h(e) \in [\alpha, \beta]$ et $\alpha \geq \beta$
- Les nœuds non élagués sont ceux tels que :

$$[\alpha, \beta] = \begin{cases} [-\infty, +\infty] & \text{ou} \\ [-\infty, b] & \text{avec } b \neq +\infty \text{ ou} \\ [a, +\infty] & \text{avec } a \neq -\infty \end{cases}$$



Coupure α



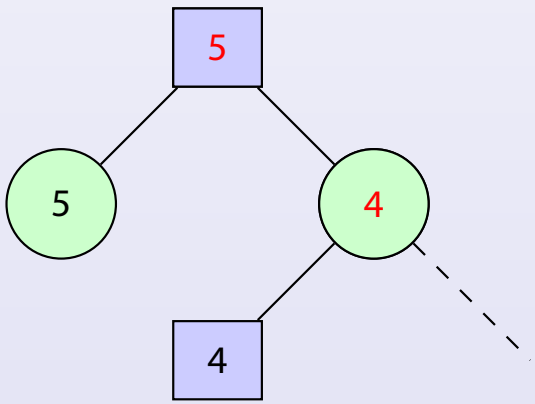
Coupure β

Élagage αβ

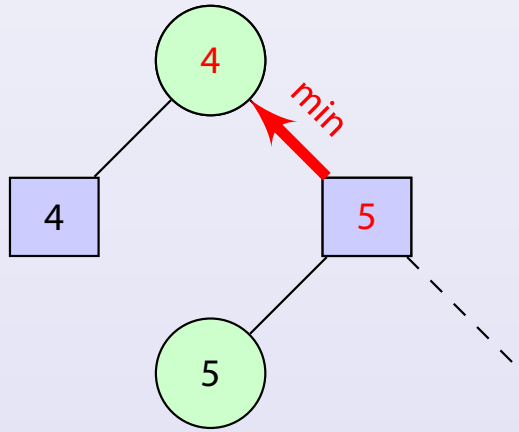
Principe

- L'algorithme est identique à *Minimax*, de signature $AlphaBeta(e, \alpha, \beta, d)$
- On ajoute les paramètres α et β (initialement $-\infty$ et $+\infty$)
- Les nœuds élagués sont ceux tels que $h(e) \in [\alpha, \beta]$ et $\alpha \geq \beta$
- Les nœuds non élagués sont ceux tels que :

$$[\alpha, \beta] = \begin{cases} [-\infty, +\infty] & \text{ou} \\ [-\infty, b] & \text{avec } b \neq +\infty \text{ ou} \\ [a, +\infty] & \text{avec } a \neq -\infty \end{cases}$$



Coupure α



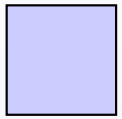
Coupure β

Élagage αβ

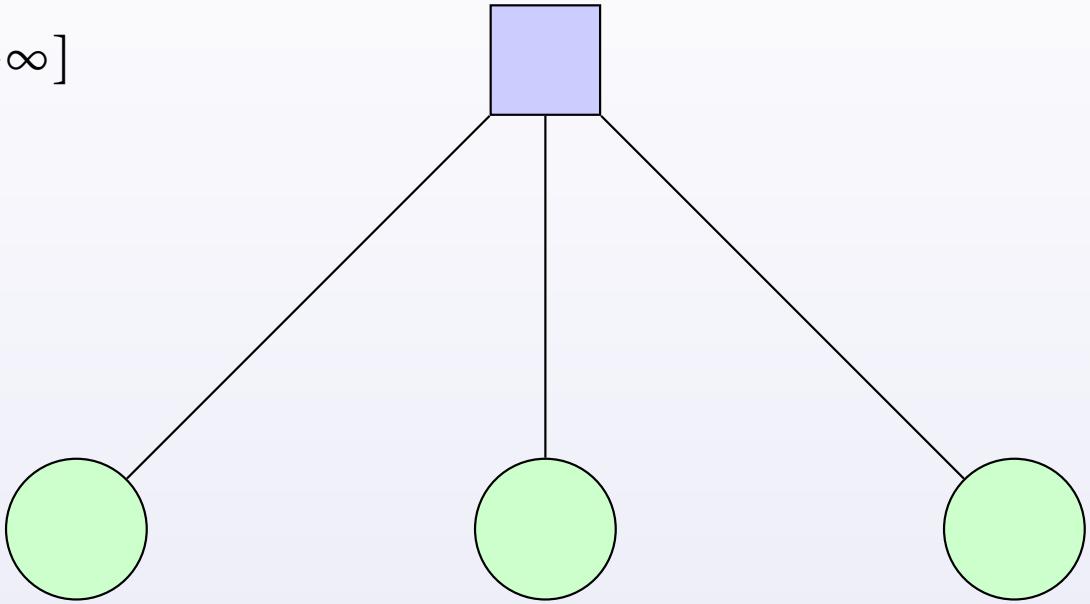
Fonction *Alphabeta*(e, d, α, β)

si *final ?(e) ou (d == 0)* **alors**
    return $h(e)$
sinon
    si *joueur ?(e)* **alors**
        $v = -\infty$
        pour $f_i \in f(e)$ **faire**
            $v = max(v, Alphabeta(f_i, d-1, \alpha, \beta))$
            si $v > \beta$ **alors** return v
            $\alpha = max(\alpha, v)$
    sinon
        $v = +\infty$
        pour tous les $f_i \in f(e)$ **faire**
            $v = min(v, Alphabeta(f_i, d-1, \alpha, \beta))$
            si $\alpha > v$ **alors** return v
            $\beta = min(\beta, v)$
    return v

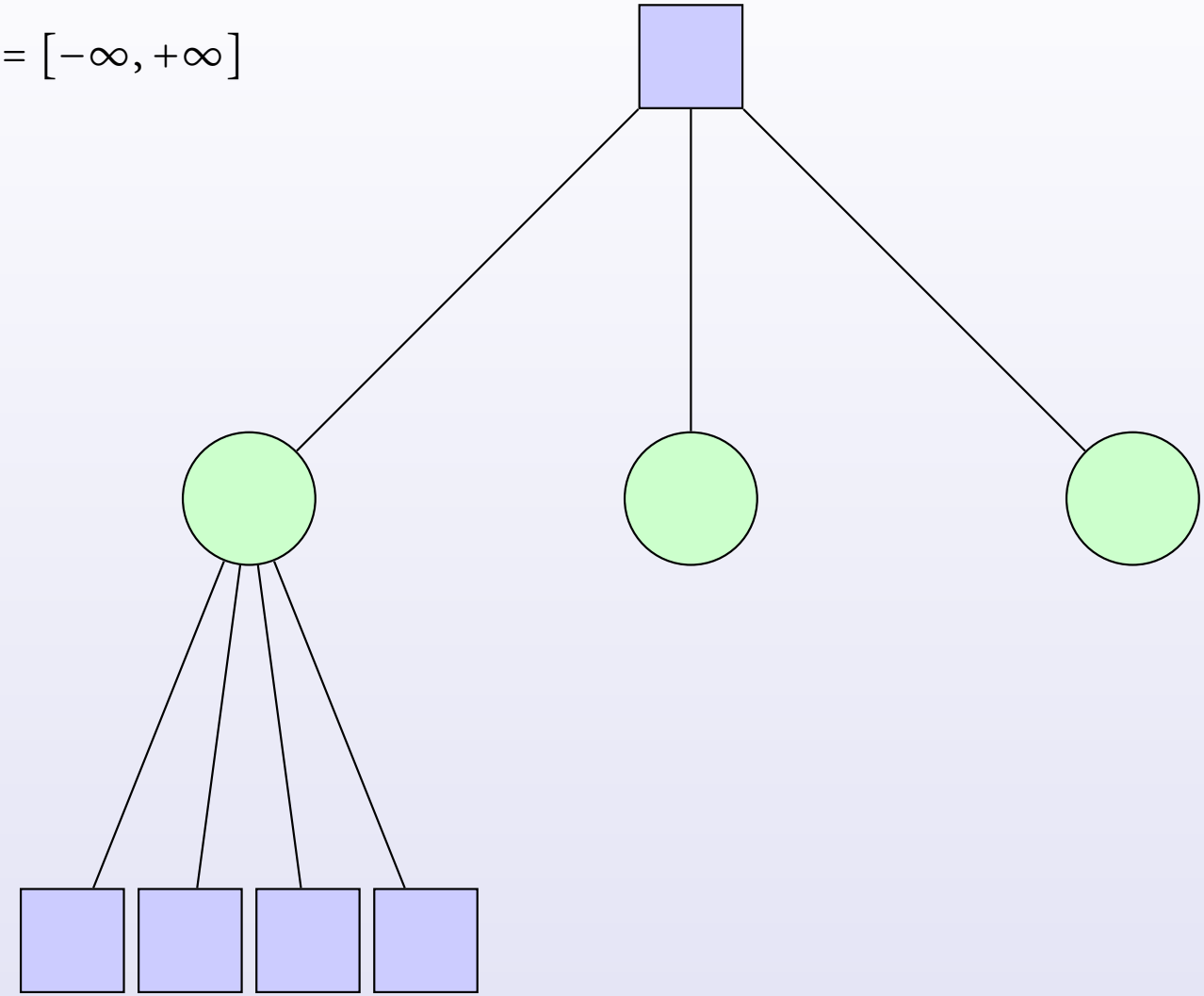
$$[\alpha, \beta] = [-\infty, +\infty]$$



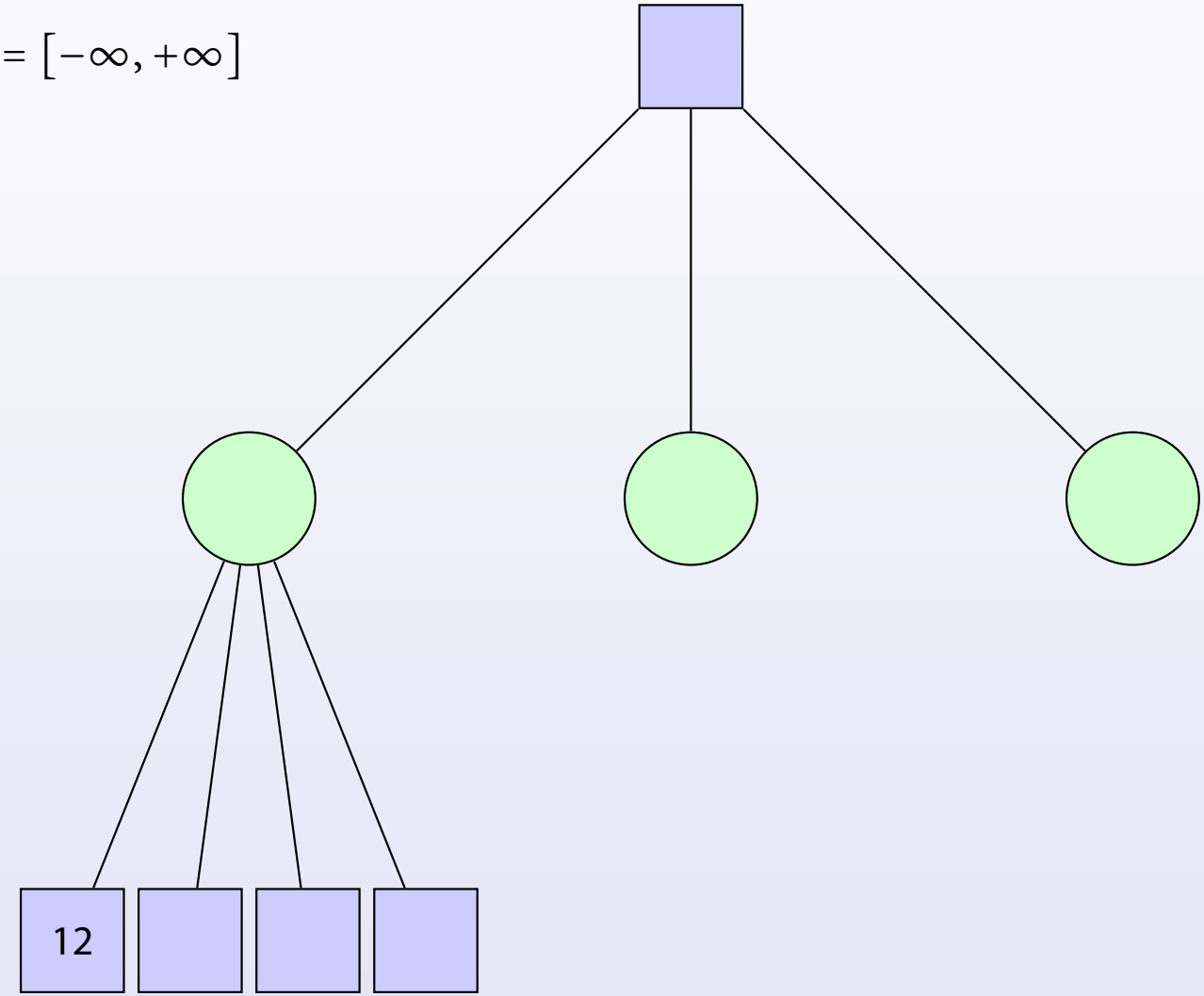
$$[\alpha, \beta] = [-\infty, +\infty]$$



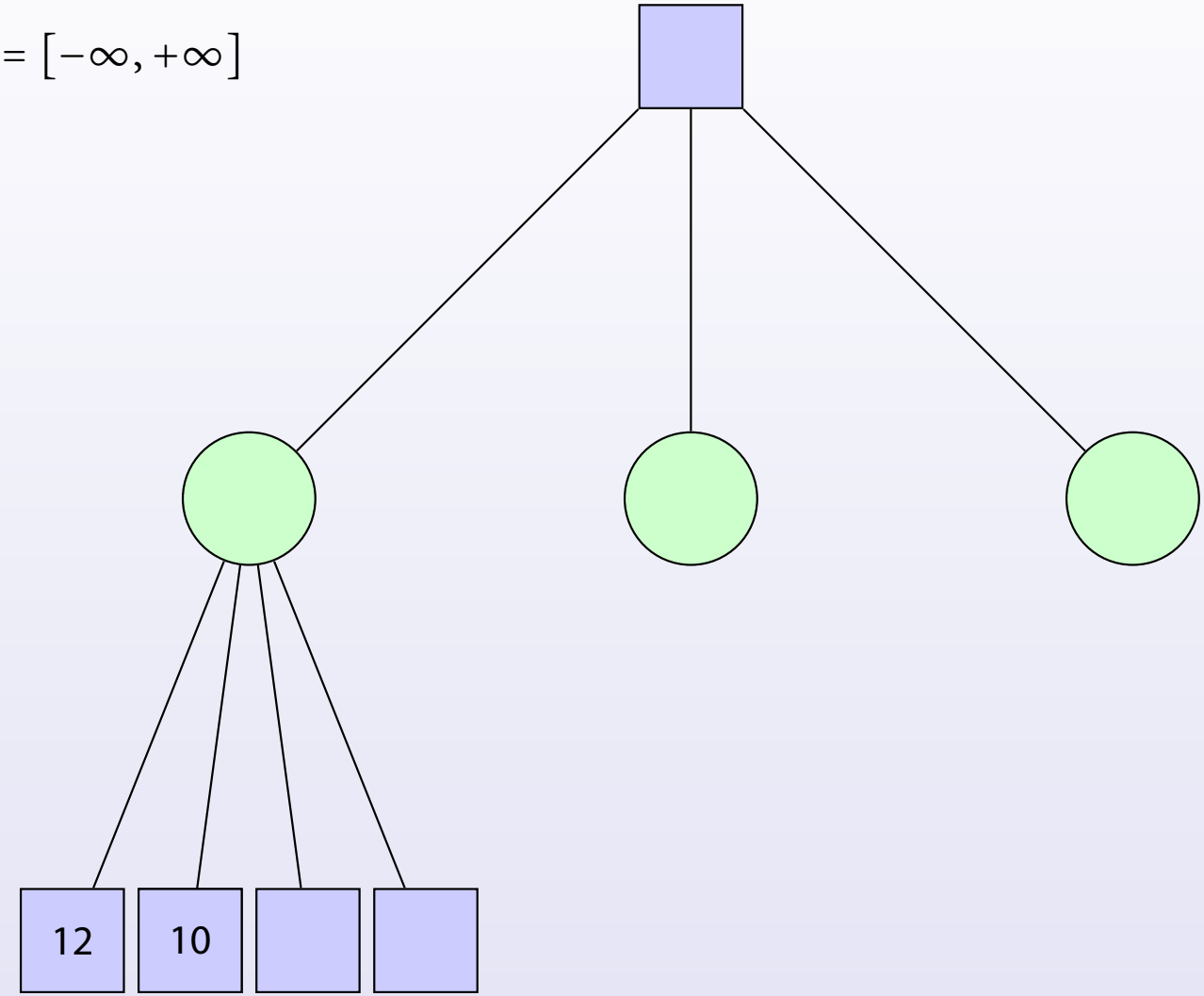
$$[\alpha, \beta] = [-\infty, +\infty]$$



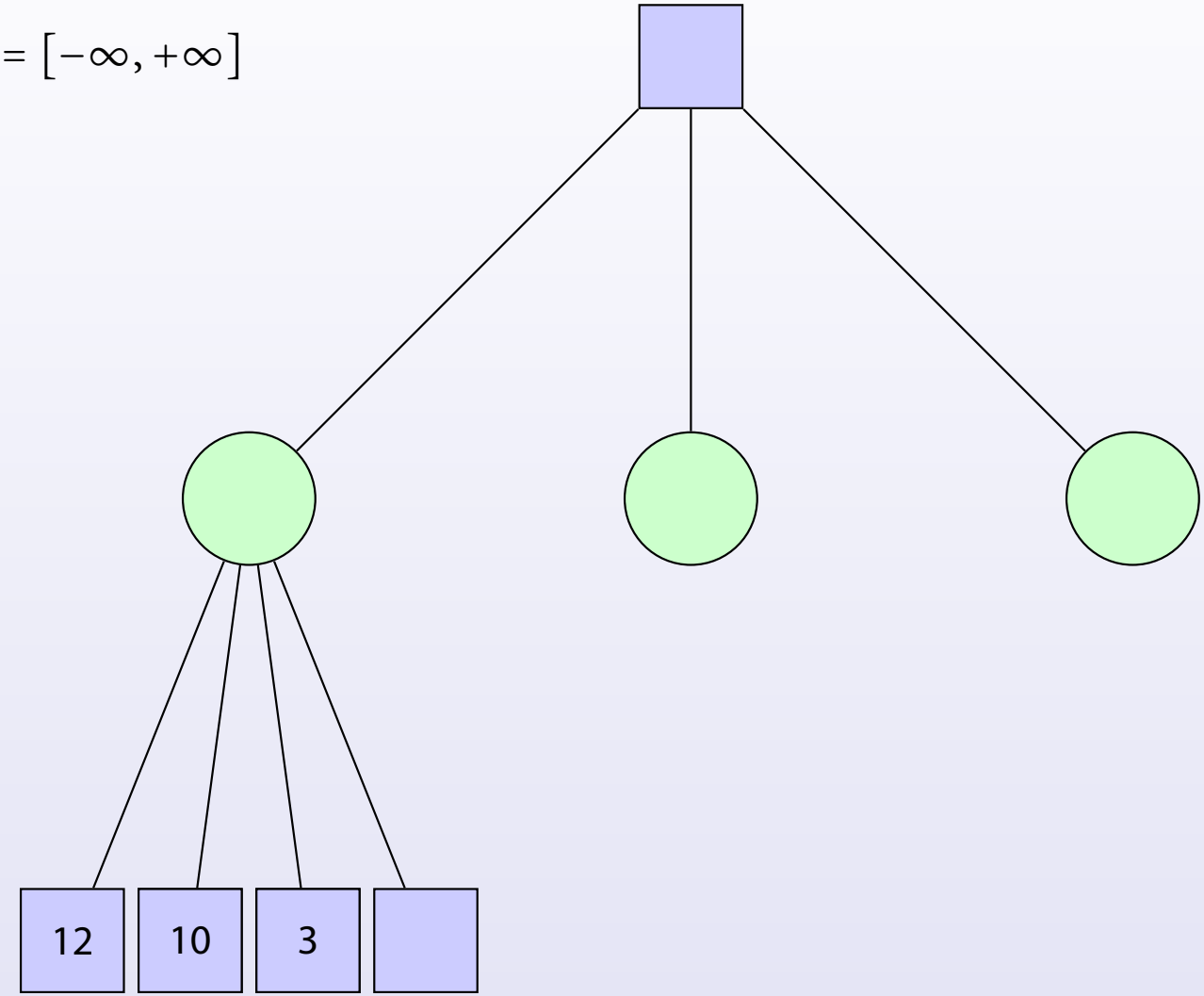
$$[\alpha, \beta] = [-\infty, +\infty]$$



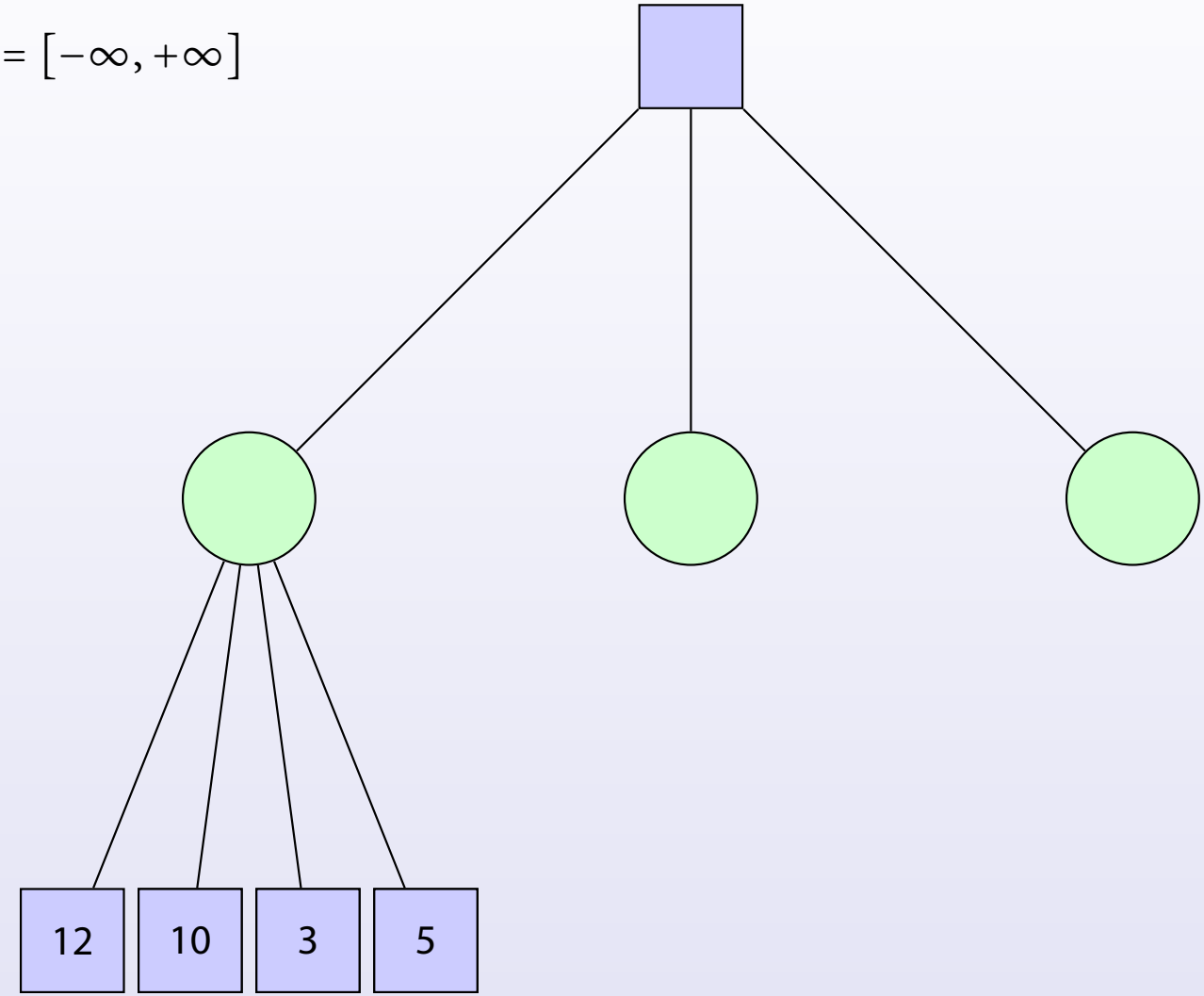
$$[\alpha, \beta] = [-\infty, +\infty]$$



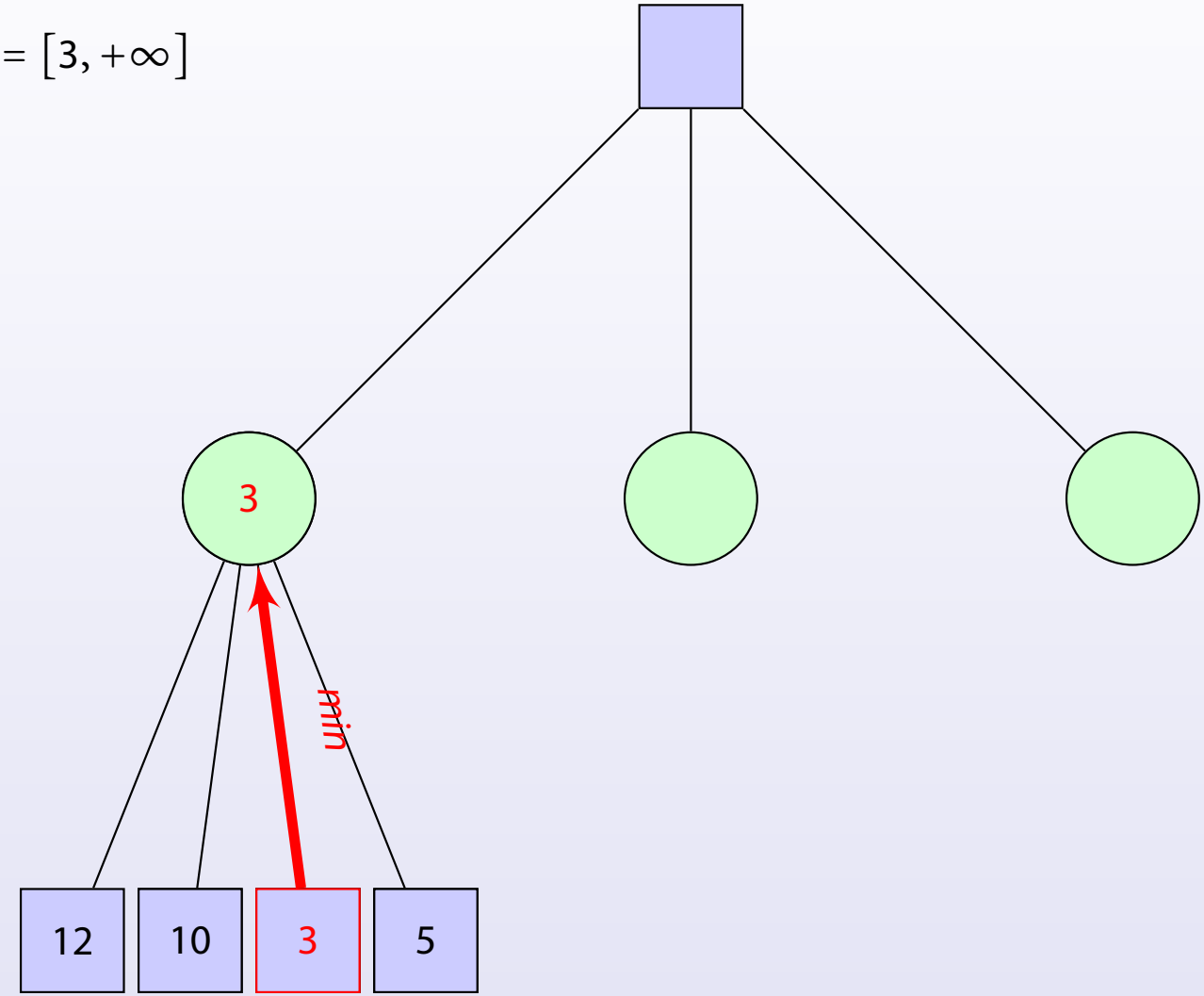
$$[\alpha, \beta] = [-\infty, +\infty]$$



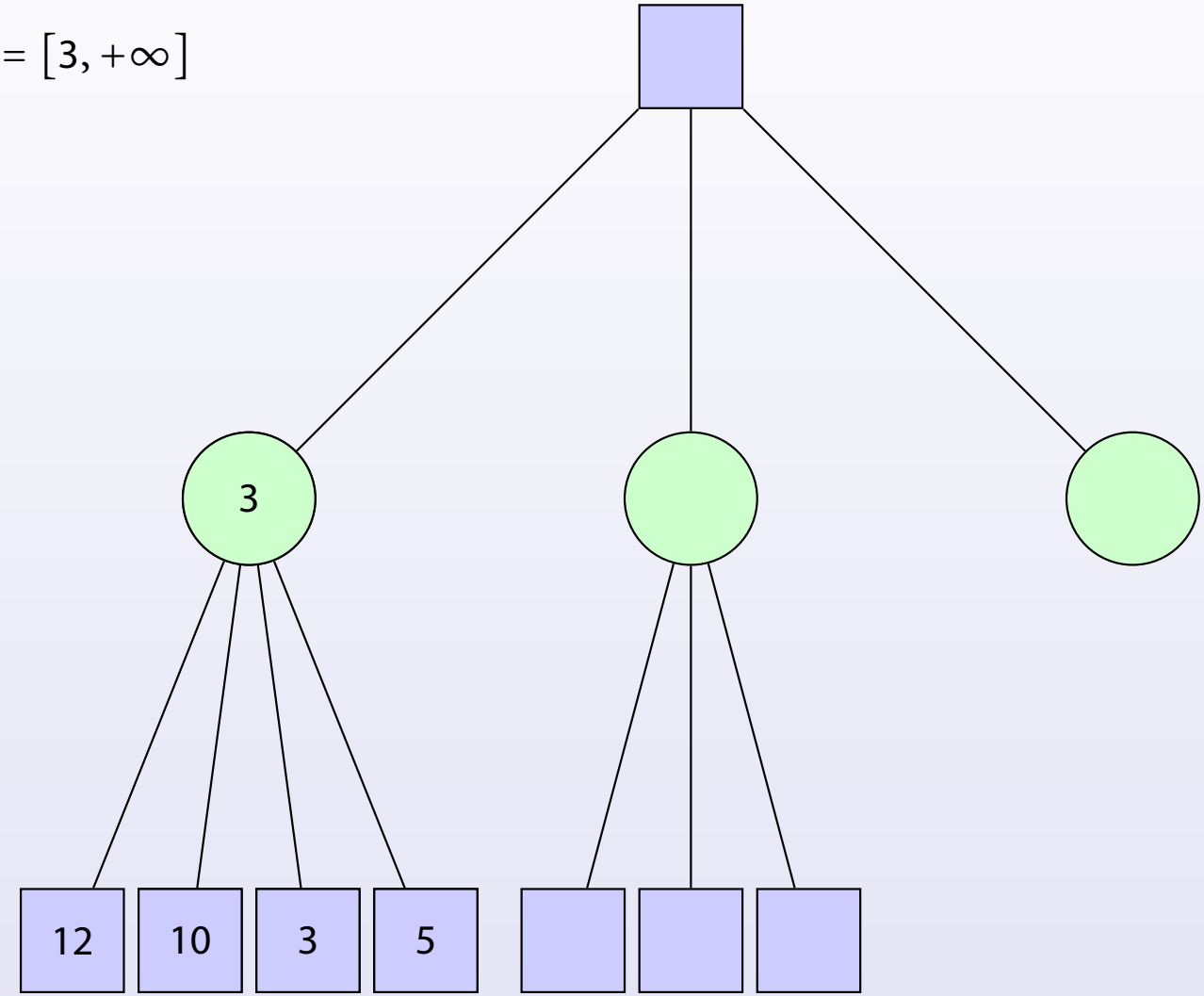
$$[\alpha, \beta] = [-\infty, +\infty]$$



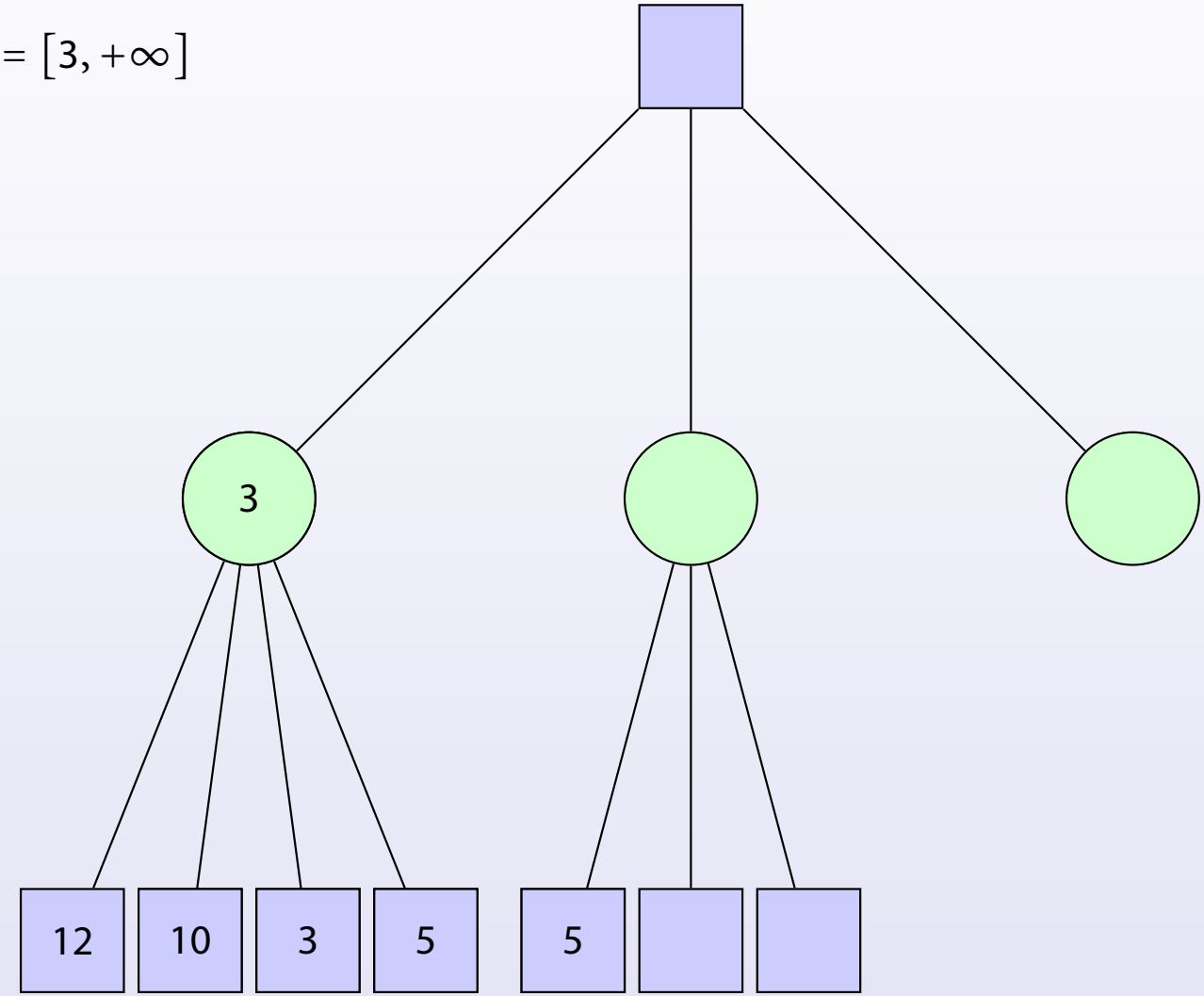
$[\alpha, \beta] = [3, +\infty]$



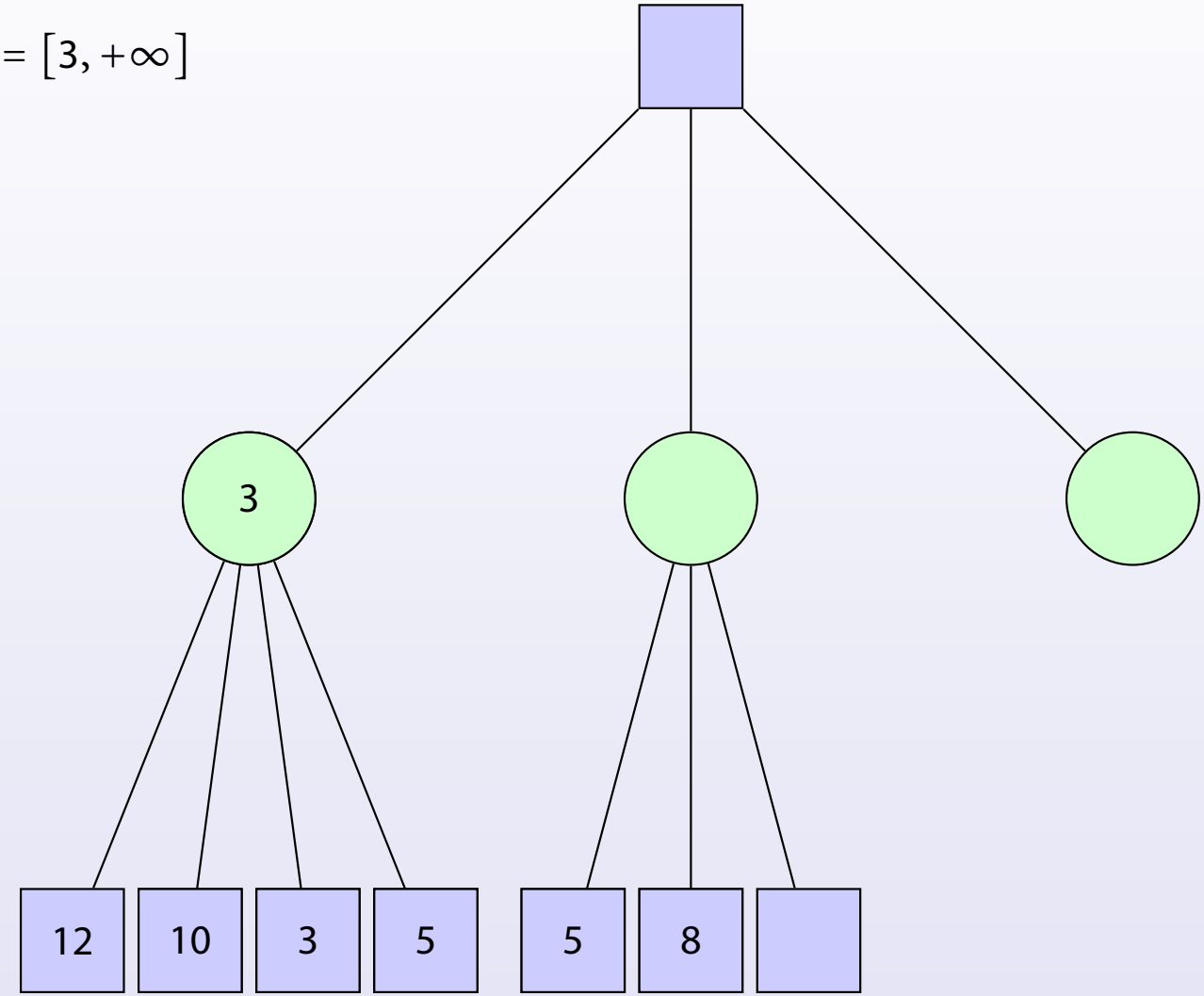
$[\alpha, \beta] = [3, +\infty]$



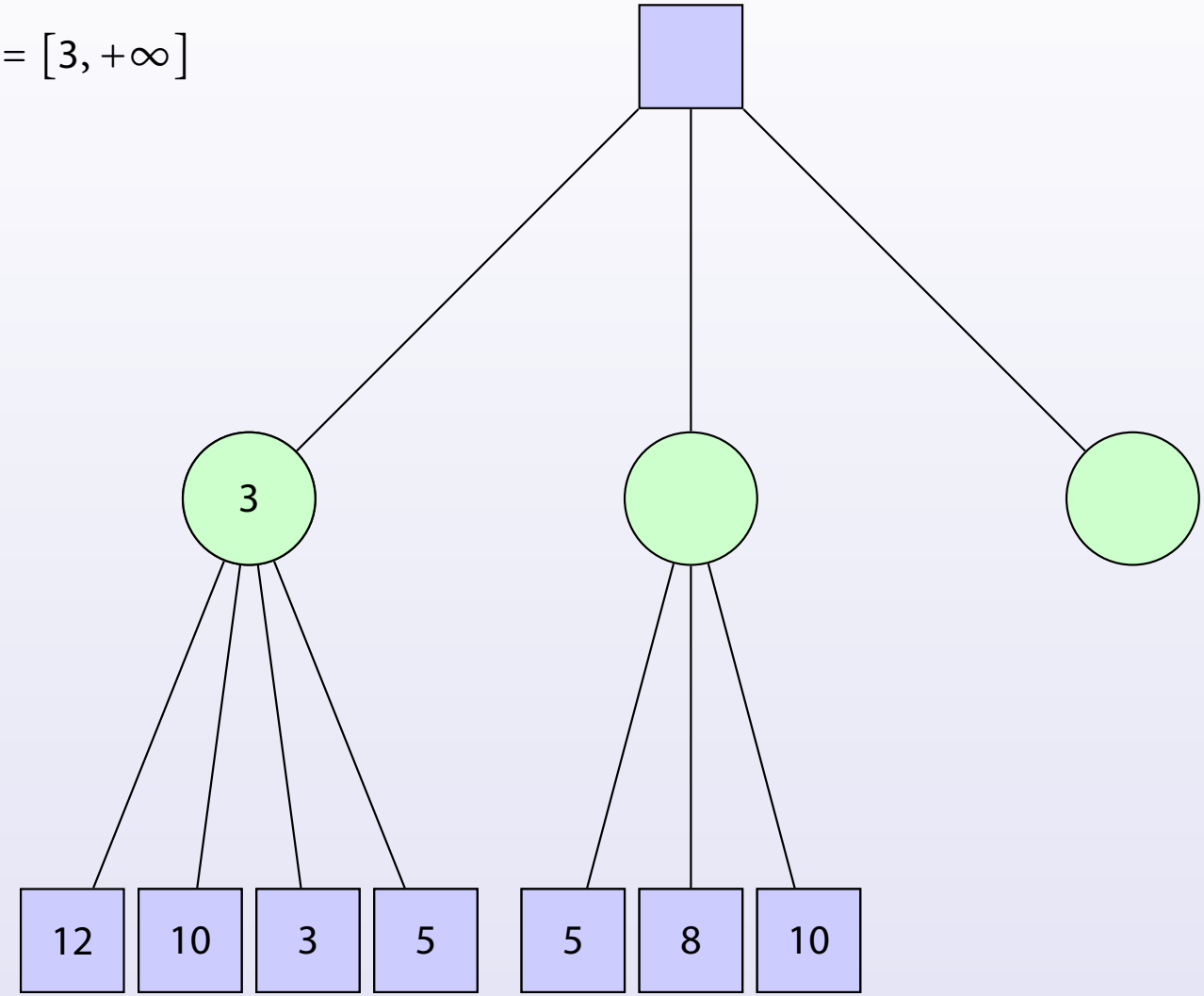
$[\alpha, \beta] = [3, +\infty]$



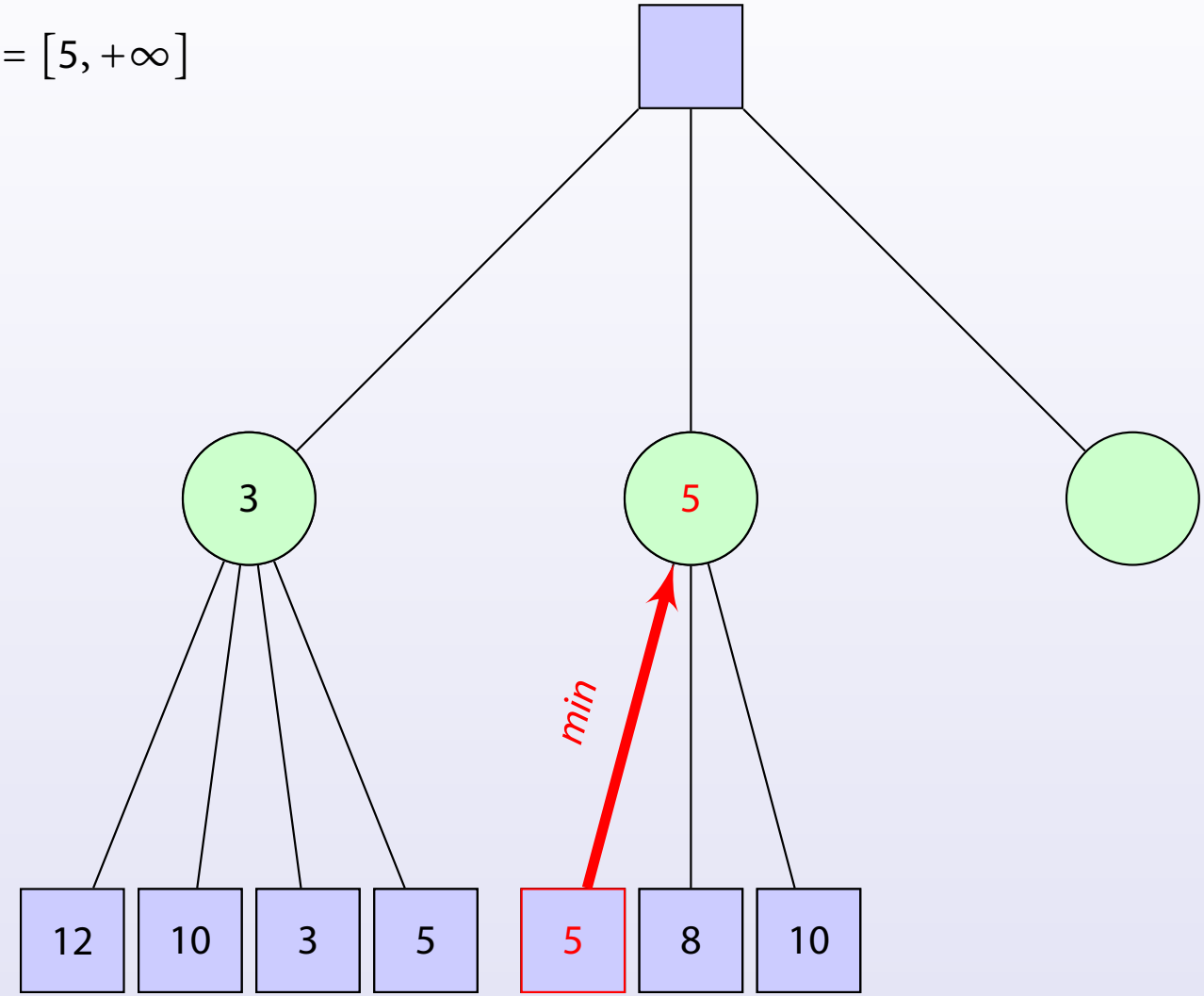
$[\alpha, \beta] = [3, +\infty]$



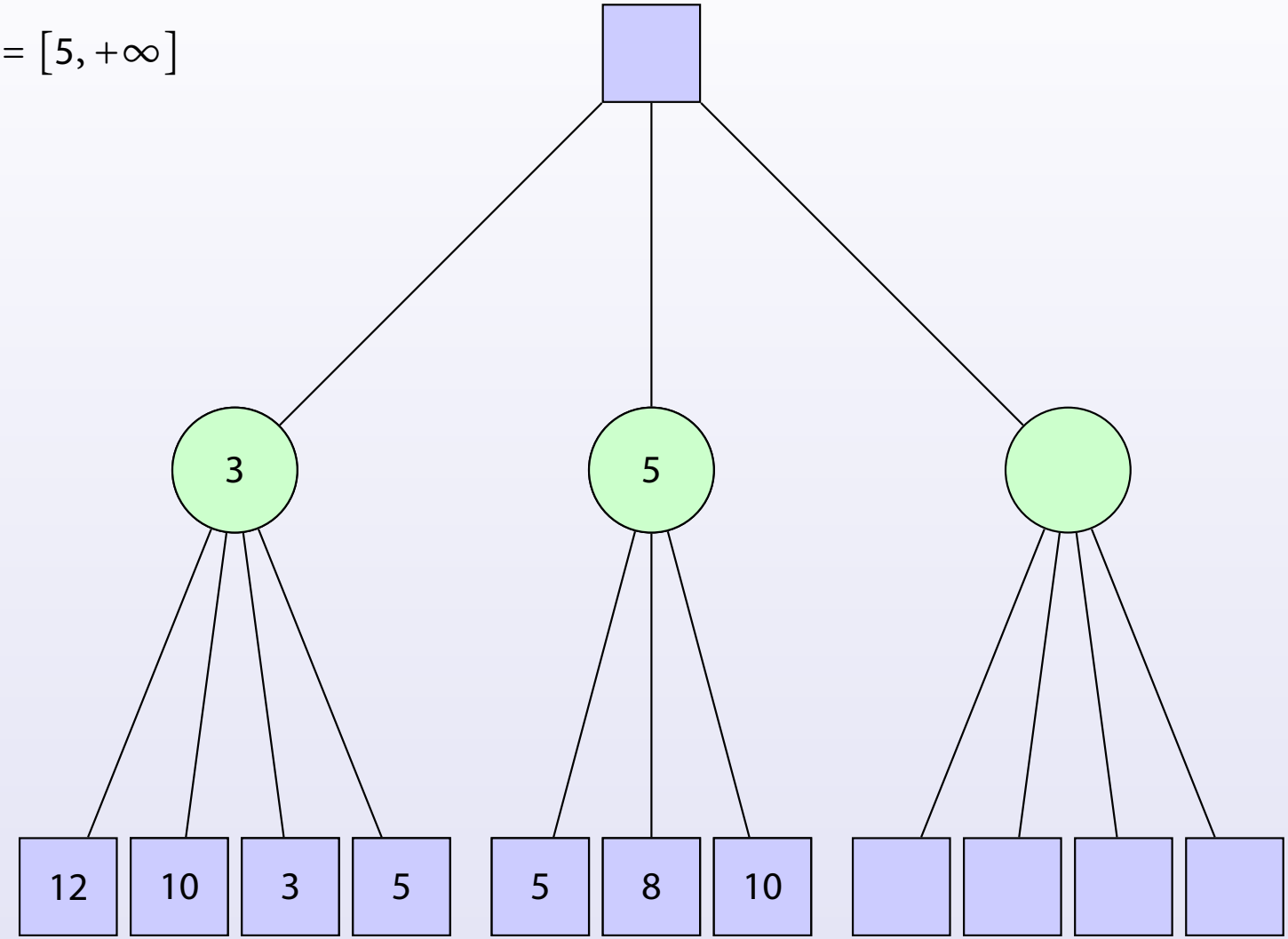
$[\alpha, \beta] = [3, +\infty]$



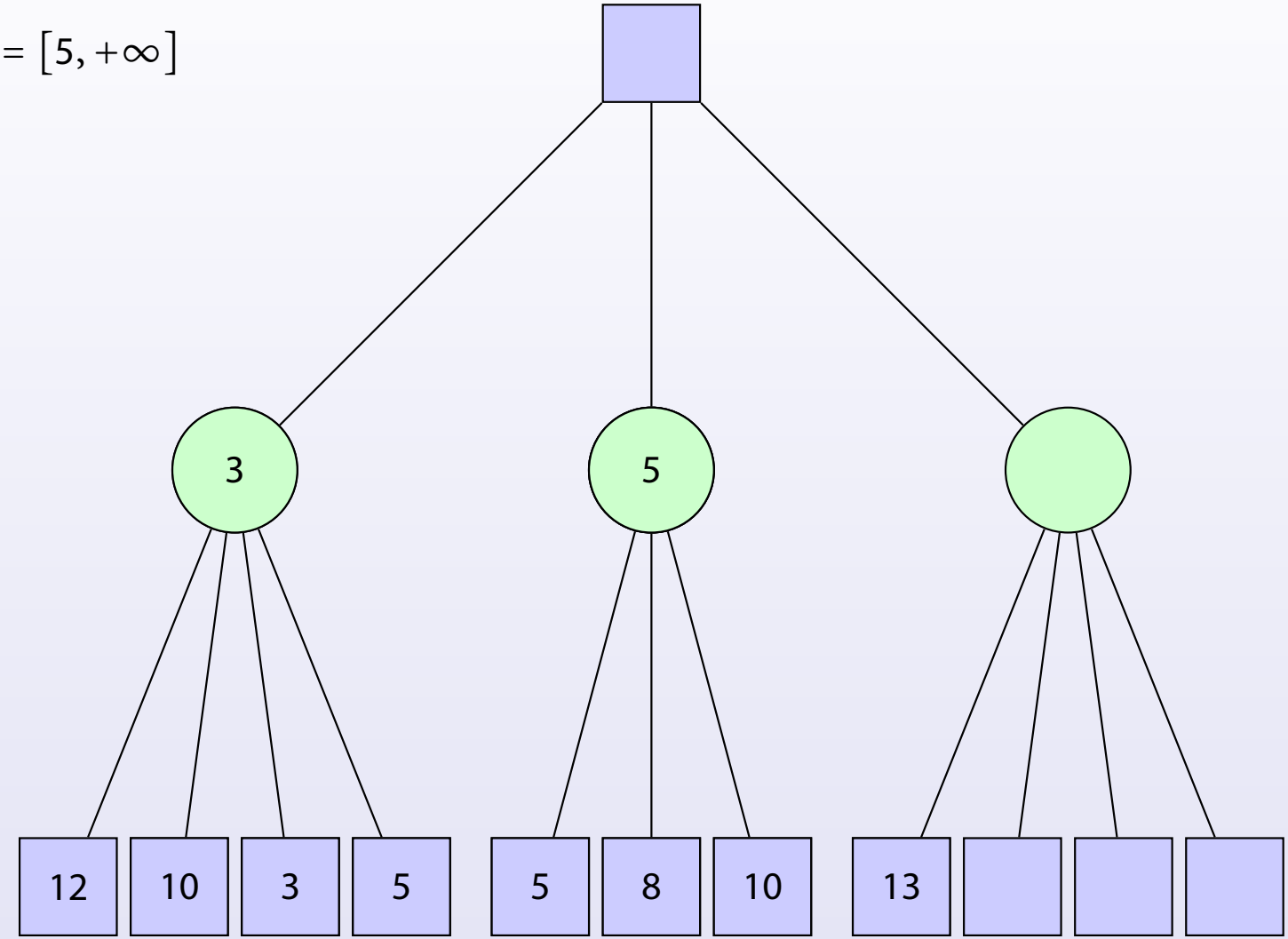
$[\alpha, \beta] = [5, +\infty]$



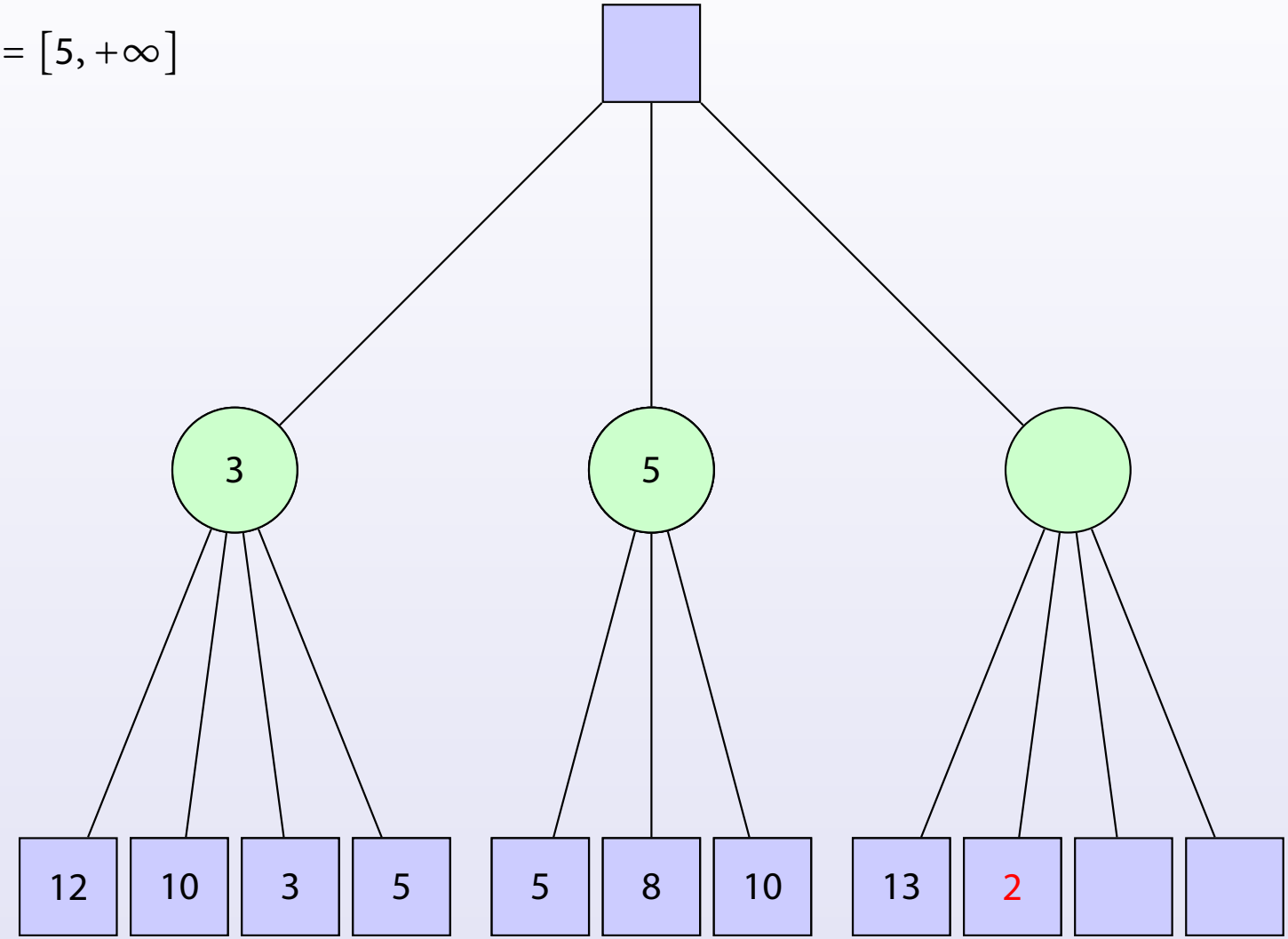
$[\alpha, \beta] = [5, +\infty]$



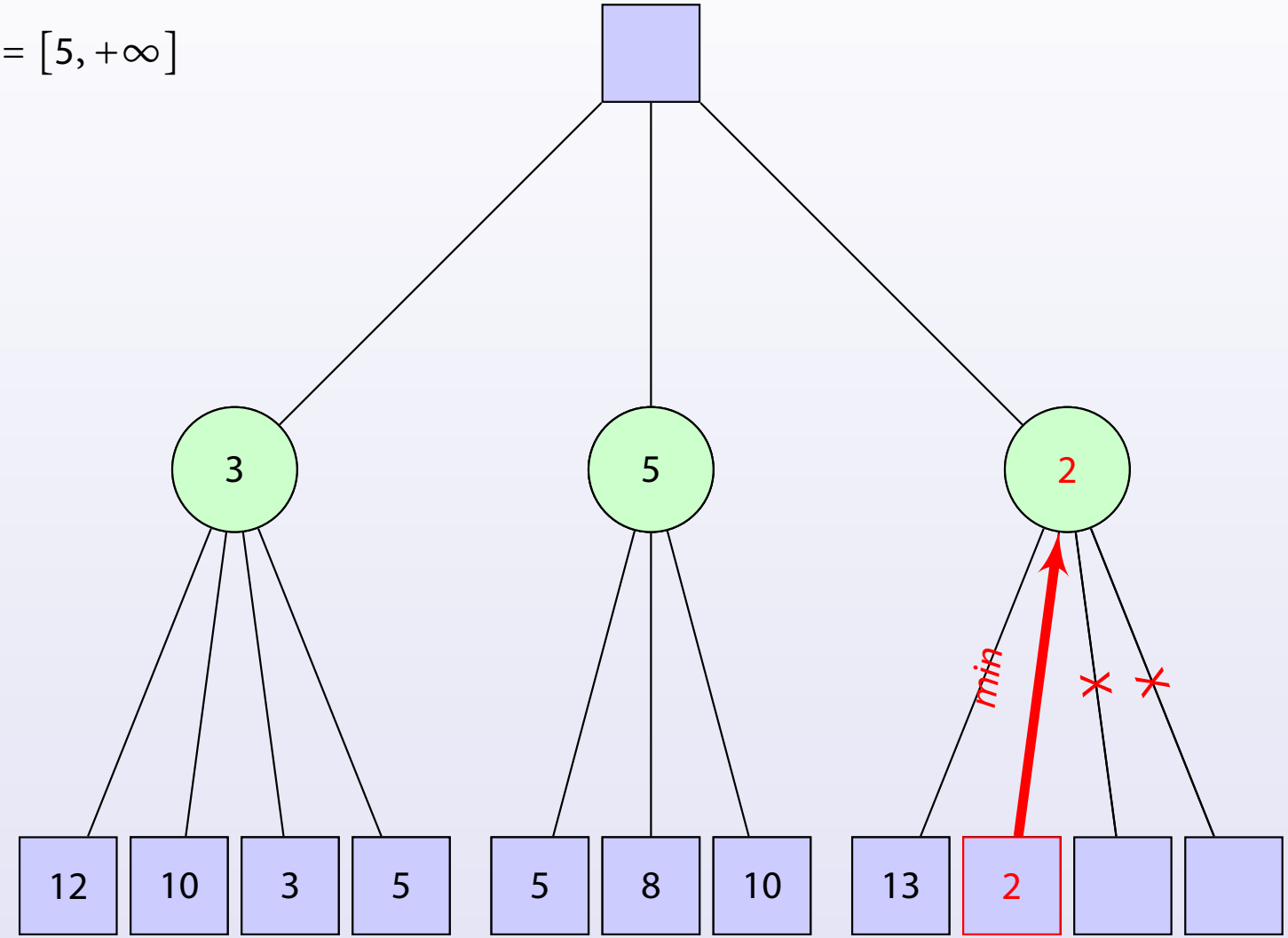
$[\alpha, \beta] = [5, +\infty]$



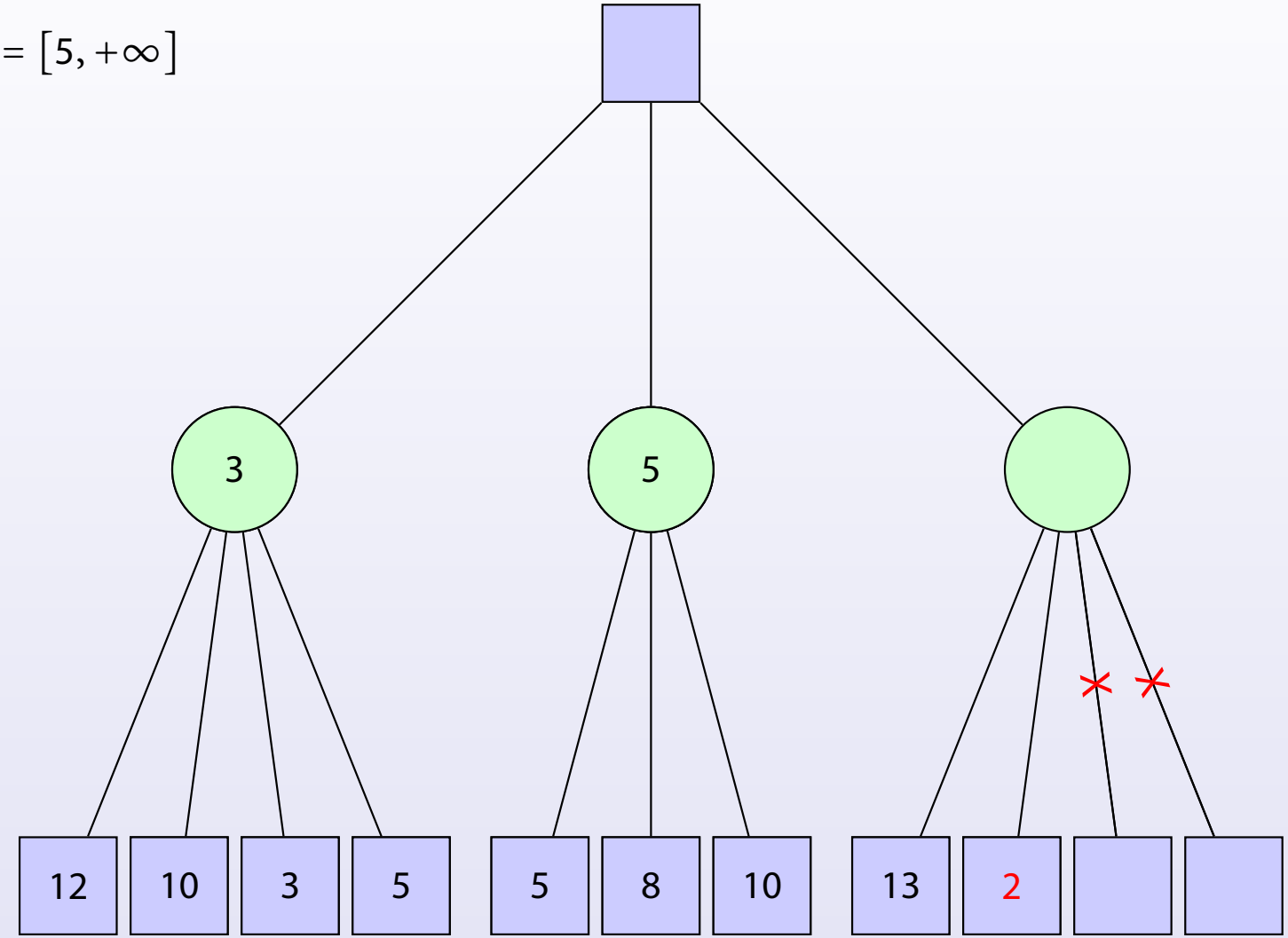
$[\alpha, \beta] = [5, +\infty]$

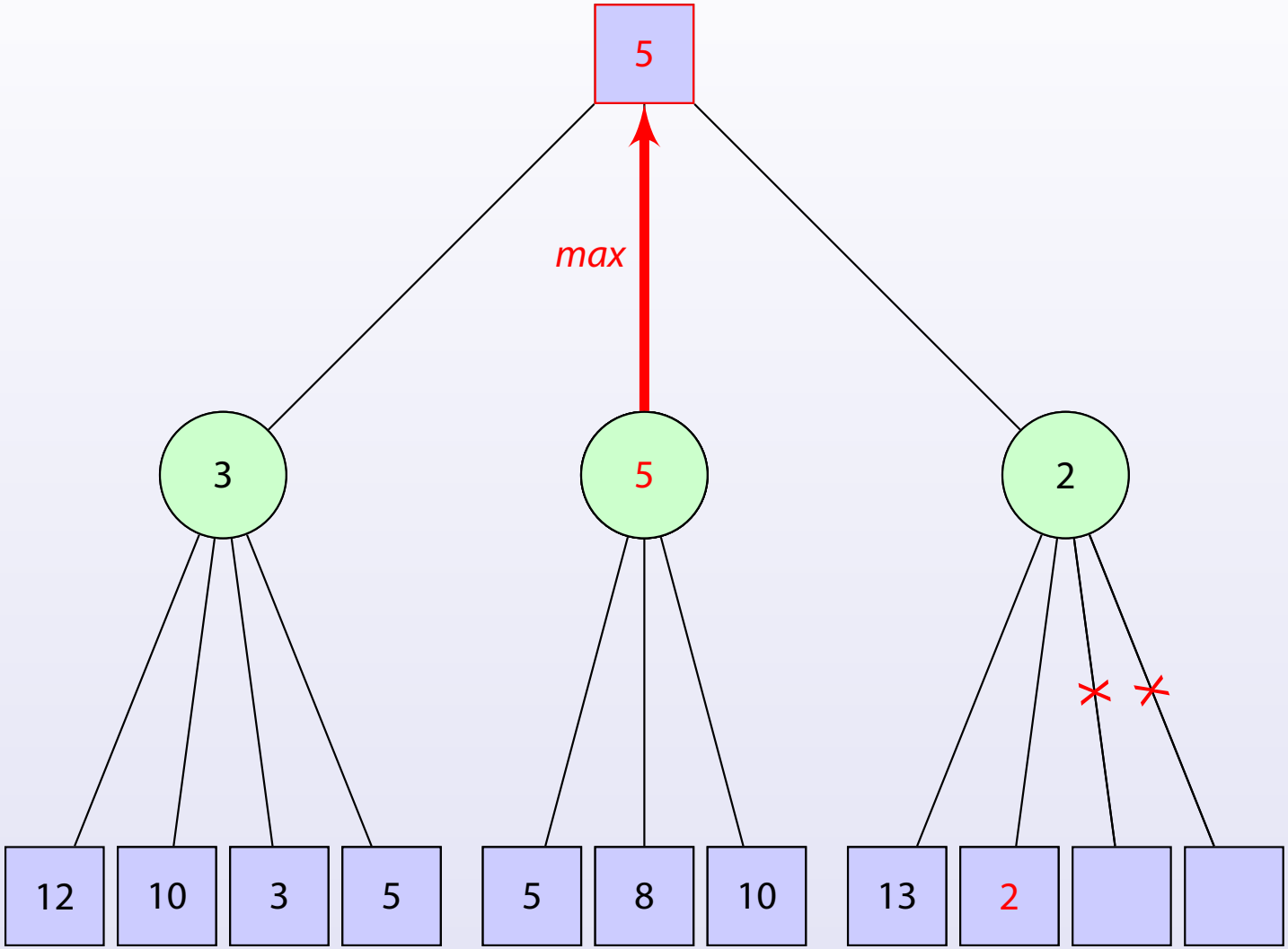


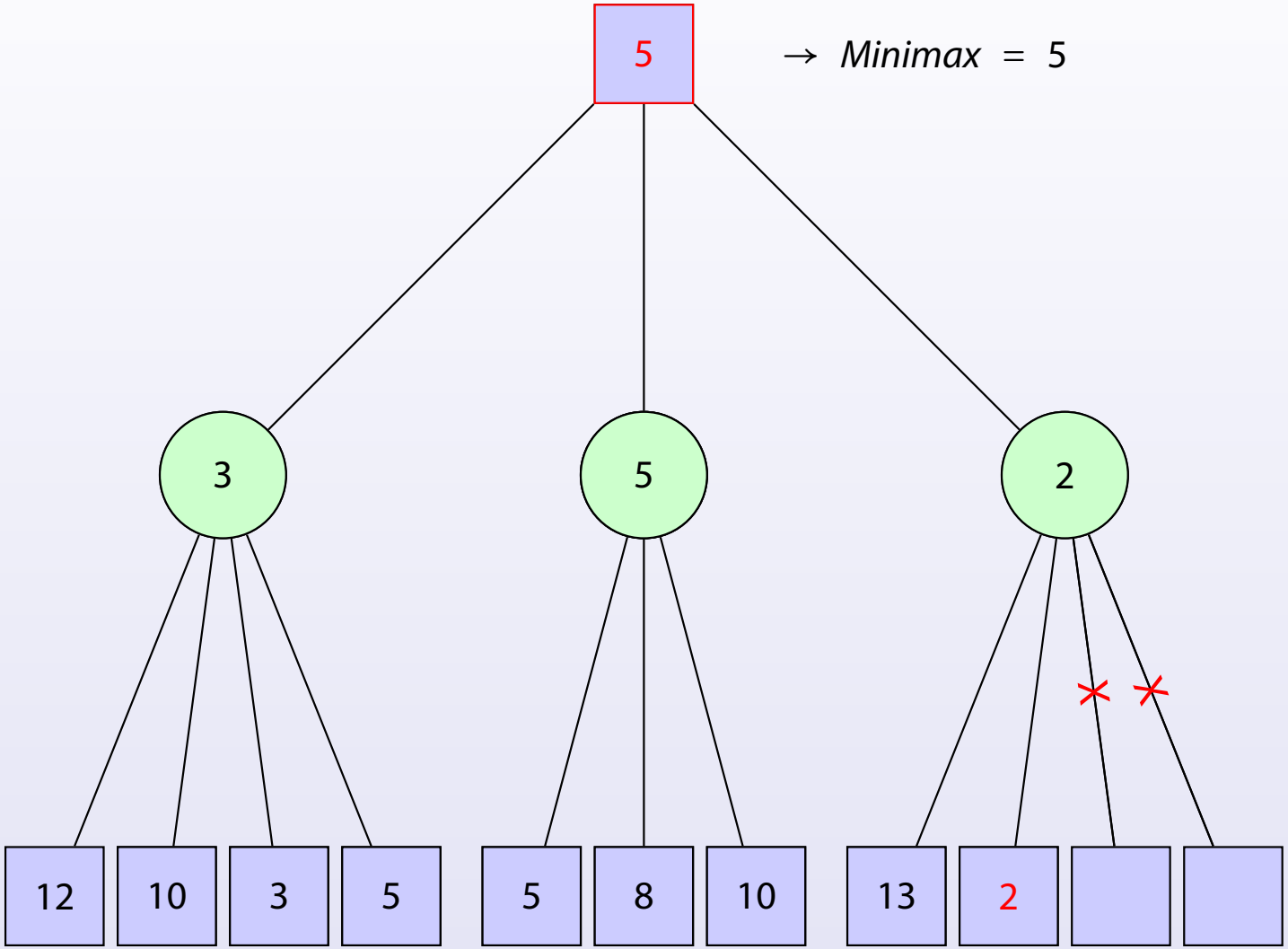
$[\alpha, \beta] = [5, +\infty]$



$[\alpha, \beta] = [5, +\infty]$




5
max
3
5
2
12
10
3
5
5
8
10
13
2


5
→ Minimax = 5
3
5
2
12
10
3
5
5
8
10
13
2

Instructions et précautions pour le projet

Méthodologie

1. Définir la structure représentant les états (données + fonctions)
2. Implémenter l'affichage de jeu et la sauvegarde/le chargement de partie
3. Implémenter la boucle de jeu pour 2 joueurs humains
4. Implémenter l'algorithme *Minimax* basique
5. Remplacer un joueur humain par l'ordinateur avec l'algorithme *Minimax*
6. Implémenter et intégrer l'élagage αβ
7. Améliorer les heuristiques et la difficulté du jeu

Implémenter l'algorithme *Minimax*, ça revient à implémenter...

- la fonction *f*, pour déterminer les fils d'un état
- la fonction *h*, pour évaluer les états
- la fonction *final*? pour déterminer si un état est final
- ...

Instructions et précautions pour le projet

Précautions en vrac

- Penser générique
  - idéalement la fonction *Minimax* est indépendante du jeu
  - seules les fonctions *f*, *h*, *final*?, etc. sont dépendantes du domaine
- Ne pas prendre le pseudo-code pour du code C
  - initialement *Minimax* renvoie une valeur entière
  - MAIS il faut pouvoir **également** renvoyer le coup à jouer
- La représentation des données est primordiale
  - La structure d'arbre n'est pas directement manipulée par le programme : ce sont des appels récursifs
  - Les listes (fils, coups, etc.) vont être représentées par des pointeurs
- Être rigoureux
  - Essayer d'abord des heuristiques simples, puis les améliorer
  - une fonction dès qu'elle est codée → jeux de tests
  - Utiliser le debugger C