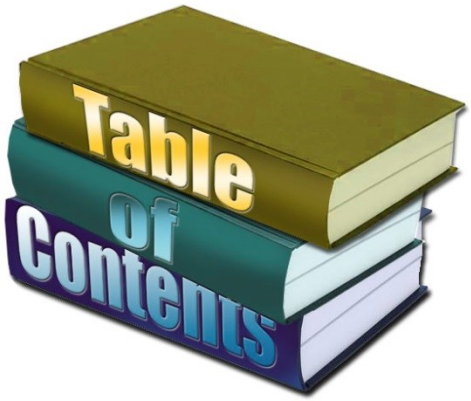
# OSNOVE UMETNE INTELIGENCE
## 2018/19

*igranje iger*

*planiranje*

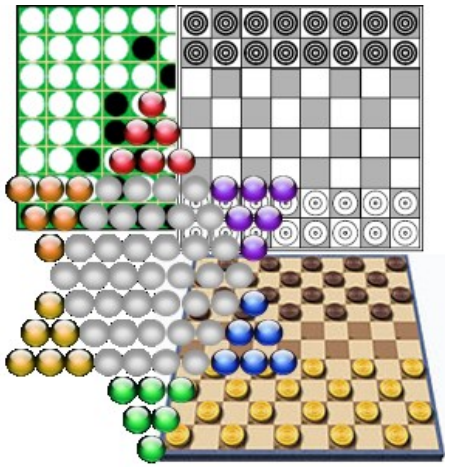
# Pregled




- igranje iger
  - predstavitev problema
  - algoritem MINIMAX
  - rezanje alfa-beta
- planiranje
  - planiranje s "klasičnim" preiskovanjem prostora stanj
  - planiranje s sredstvi in cilji
  - planiranje z regresiranjem ciljev

# Igranje iger

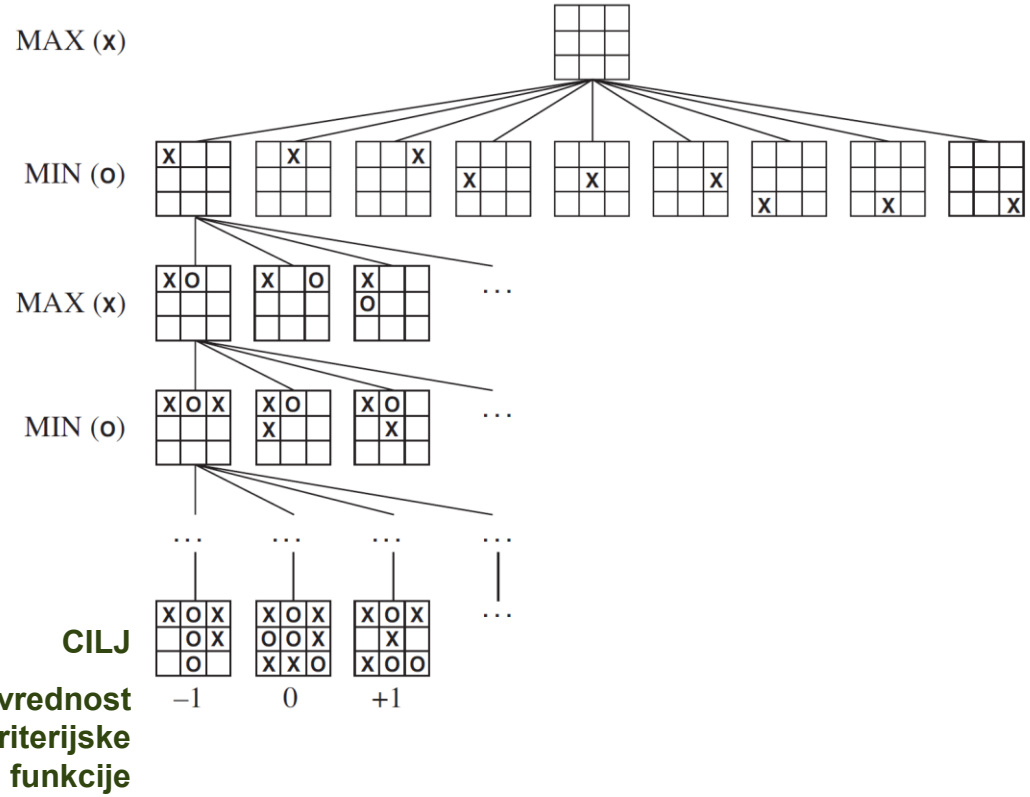
- preiskovanje prostora med dvema nasprotnikoma (angl. *adversarial search*)
- več-agentno tekmovalno okolje, kjer mora vsak agent upoštevati vpliv akcij drugega agenta na svojo uspešnost
- večina iger: deterministične, izmenične poteze, dva igralca, transparentne (s popolno informacijo)
  - primeri iger s popolno informacijo: šah, dama, go
  - primeri iger z nepopolno informacijo: potapljanje ladjic, poker, scrabble
- rešitev igre je **strategija**, ki za vsako možno potezo nasprotnika predvidi akcijo



- izziv:
  - iskanje rešitev je lahko kompleksno, velik prostor stanj
  - primer: šah ima faktor vejanja okoli 35, igra vsebuje okoli 50 potez vsakega igralca → to pomeni $35^{100}$ $(= 10^{54})$ stanj

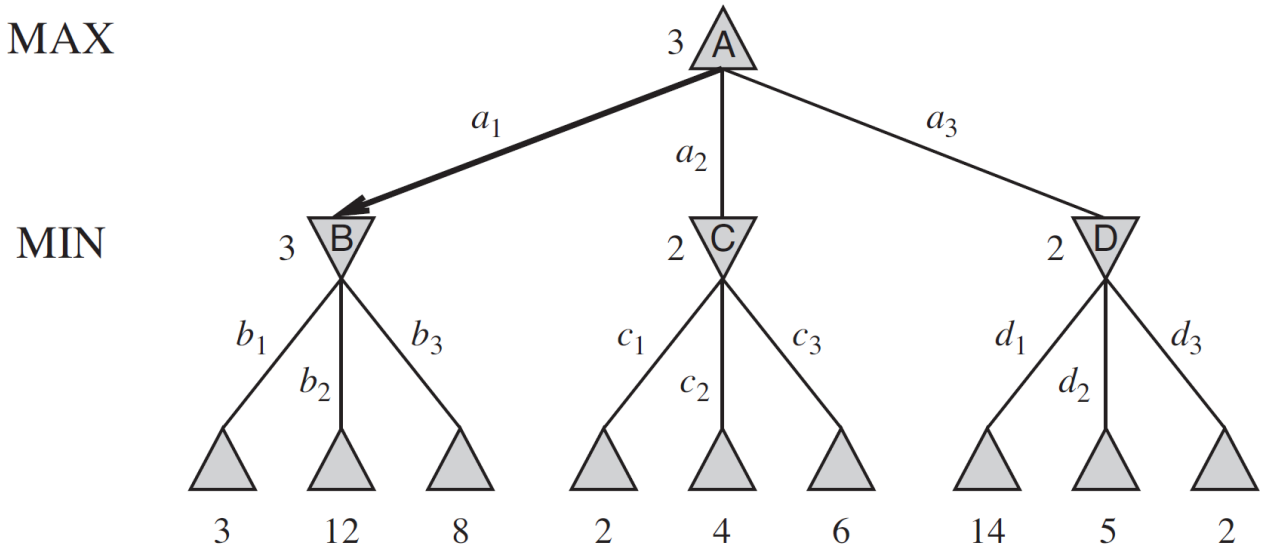
# Igranje iger

- potek igre predstavimo z **igralnim drevesom**, v katerem si poteze izmenjujeta igralca **MAX** in **MIN**
- ciljna stanja vrednotimo s **kriterijsko funkcijo** (pozitivne vrednosti so ugodne za MAX, negativne za MIN)
  - pojem: igra s konstantno vsoto kriterijske funkcije (šah: 1+0, 0+1, ½+½)



# Igranje iger

- celotna igralna drevesa so lahko velika (križci in krožci – 362.880 ciljnih vozlišč, šah - $10^{40}$ ciljnih vozlišč)
- iskalno drevo vsebuje podmnožico vseh možnih stanj igralnega drevesa, ki razkriva dovolj informacije za izvedbo poteze
- ne zadošča iskanje končnega vozlišča, ker na pot vpliva nasprotni igralec (MIN)
- podobno predstavitvi z grafi AND/OR
  - OR: izbira poteze s strani igralca MAX
  - AND: predvideti je potrebno vse poteze nasprotnika MIN



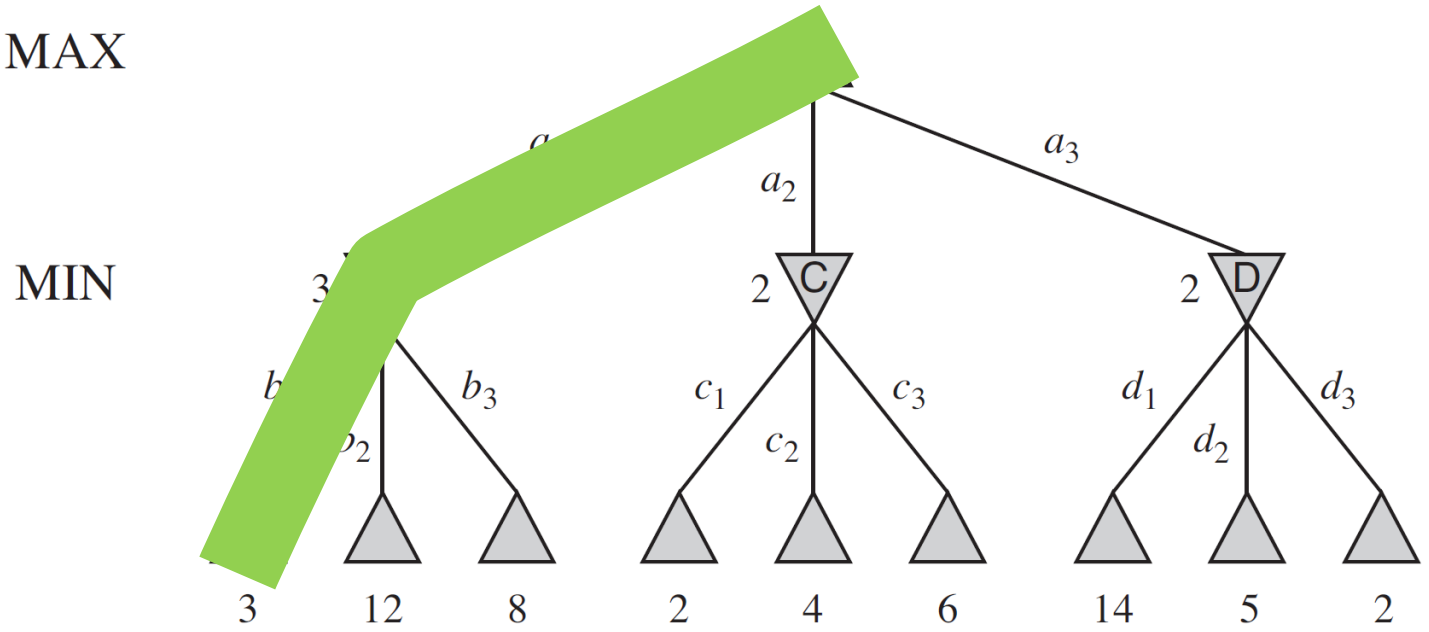
# Igranje iger

- optimalno strategijo določa *MINIMAX* vrednost vozlišča, ki je enaka vrednosti kriterijske funkcije (za MAX), če oba igralca igrata optimalno
  - MAX preferira zvišanje vrednosti kriterijske funkcije (najboljša lastna poteza)
  - MIN preferira znižanje vrednosti kriterijske funkcije (najboljša protipoteza)
  - predpostavimo, da MIN igra optimalno

- algoritem *MINIMAX*: namenjen izračunu *MINIMAX* vrednosti za vozlišča
  - potek algoritma: rekurzivni spust v globino do listov drevesa, izračun vrednosti po formuli ob vračanju

$$MINIMAX(v) = \begin{cases} kriterijska\_funkcija(v) & \text{če je } v \text{ končno stanje} \\ \max\limits_{a \in akcija(v)} MINIMAX(rezultat(v,a)) & \text{če je igralec } MAX \\ \min\limits_{a \in akcija(v)} MINIMAX(rezultat(v,a)) & \text{če je igralec } MIN \end{cases}$$

# Algoritem *MINIMAX*

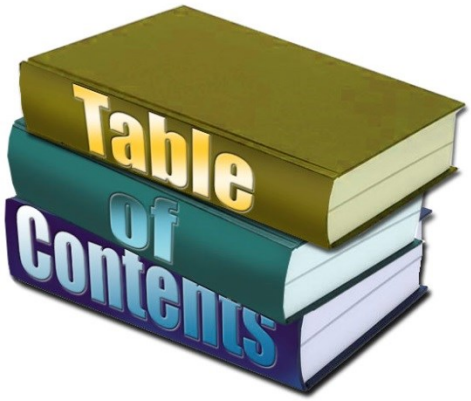


# Algoritem *MINIMAX*

- **popolnost algoritma:**
  - da, če je prostor stanj končen (ta je definiran s pravili igre)
- **optimalnost algoritma**: da, če nasprotnik igra optimalno strategijo
  - kaj, če ne?
- **časovna zahtevnost**: $O(b^m)$
- **prostorska zahtevnost**: $O(bm)$ ali $O(m)$
  - od česa je zgornje odvisno?

- ali je potrebno preiskati celoten prostor stanj?
  - rezanje drevesa (alfa-beta rezanje)

# Pregled

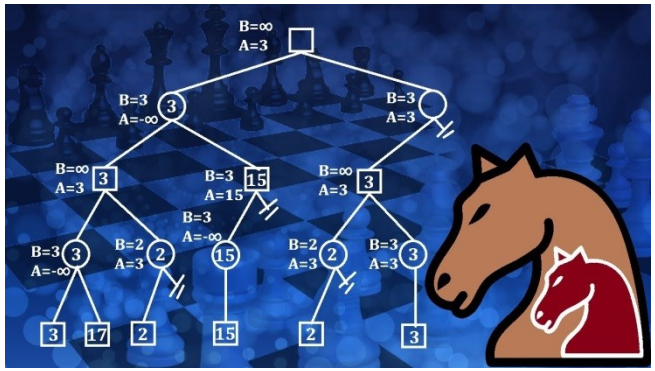



- igranje iger
  - predstavitev problema
  - algoritem *MINIMAX*
  - rezanje alfa-beta
- planiranje
  - planiranje s "klasičnim" preiskovanjem prostora stanj
  - planiranje s sredstvi in cilji
  - planiranje z regresiranjem ciljev

# Rezanje alfa-beta

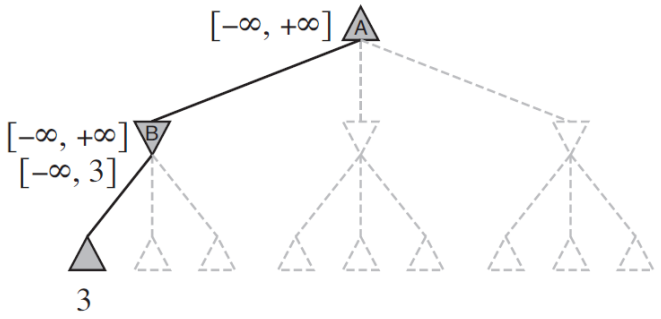
- vrne isto zaporedje potez kot bi algoritem MINIMAX s to razliko, da ne upošteva vej, ki ne vplivajo na končno odločitev
- za vsako vozlišče spremljamo vrednosti $[\alpha, \beta]$:

  $\alpha$ – najboljša do sedaj najdena rešitev za vozlišča MAX (najvišji že najdeni maksimum)
  $\beta$ – najboljša do sedaj najdena rešitev za vozlišča MIN (najnižji že najdeni minimum)
- algoritem:
  - podoben kot MINIMAX, izvaja nastavljanje vrednosti MIN in MAX s preiskovanjem v globino; za začetno vozlišče velja $[\alpha, \beta] = [-\infty, +\infty]$
  - na vsakem koraku v globino prenaša vrednosti $[\alpha, \beta]$
  - ob vračanju posodablja vrednosti $[\alpha, \beta]$ nadrejenega vozlišča
  - če v nekem vozlišču velja $\alpha \geq \beta$, lahko prekinemo preiskovanje ostalih poddreves (izvedemo rezanje)

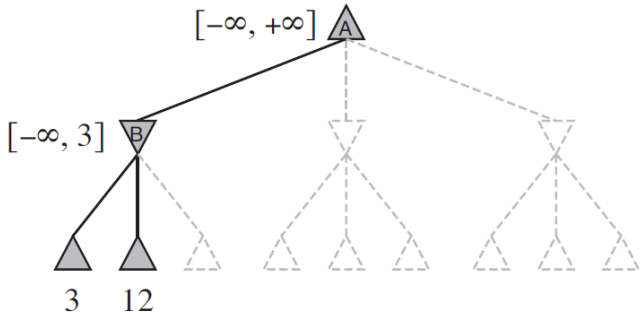


# Primer

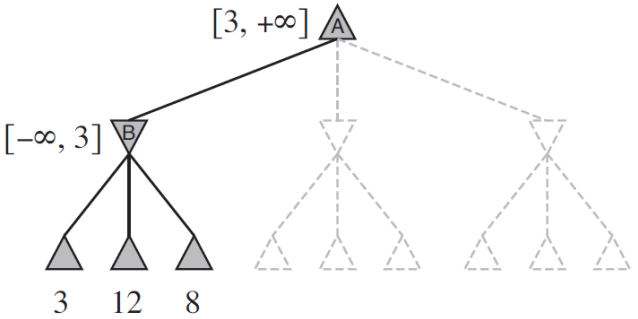
Propagiranje $[\alpha, \beta]$ navzdol. Ob vračanju za vozlišče B velja $\beta$=3 (najnižji najdeni minimum)



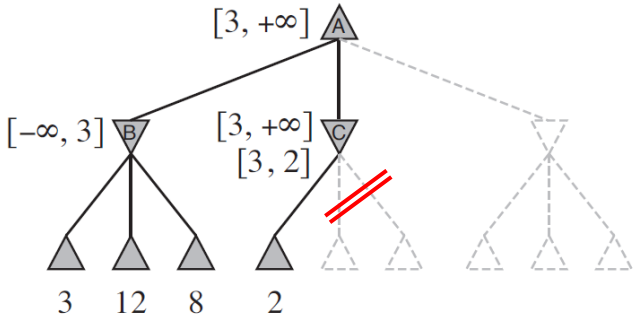

Naslednji naslednik B (vrednost 12) ne spremeni vrednosti najdenega minimuma za B. Ostane $\beta$=3.



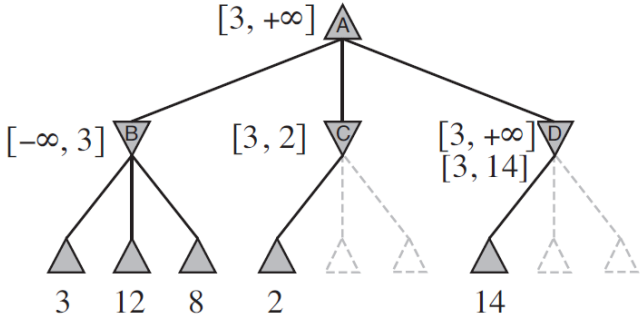

Tudi tretji naslednik B (vrednost 8) ohrani $\beta$=3.
Za vozlišče A to pomeni, da je $\alpha$=3 (najvišji najdeni maksimum).
Preiskujemo naprej (iščemo višji maksimum).



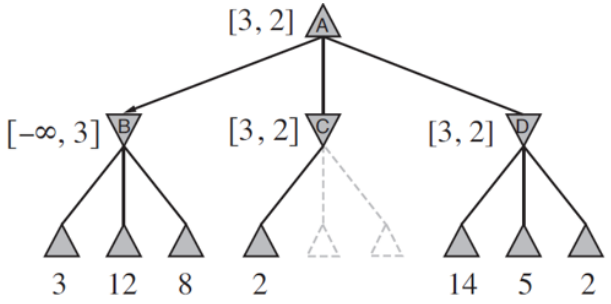

$[3, +\infty]$ se ob preiskovanju propagira k nasledniku C. Ob vračanju za C velja $\beta$=2, $[\alpha, \beta]$ = [3,2]. Ker $\alpha \geq \beta$, preiskovanje ostalih poddreves ne bi vplivalo na vrednost vozlišča A. Porežemo.



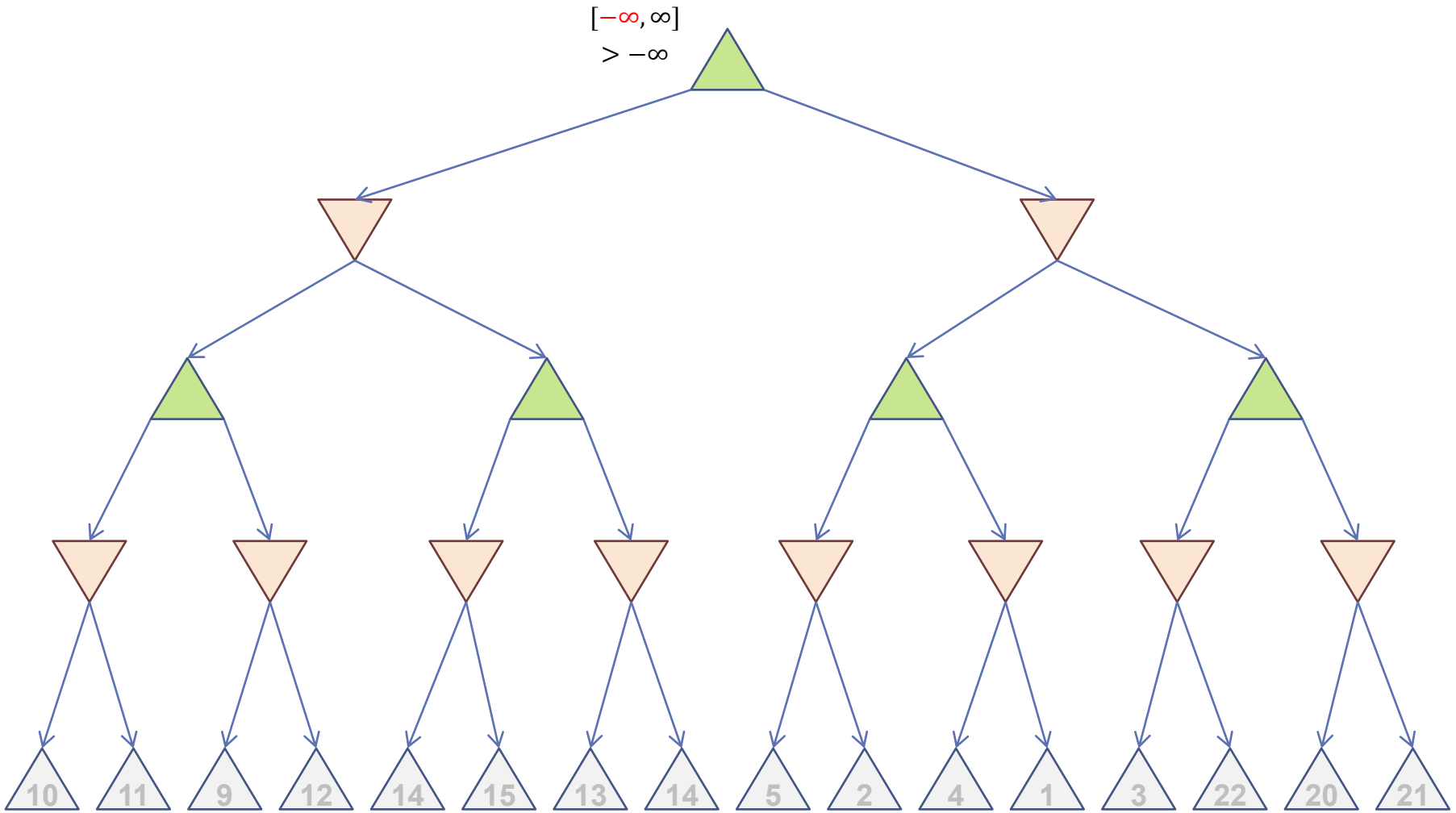

$[3, +\infty]$ se ob preiskovanju propagira k nasledniku D. Ob vračanju za D velja $\beta$=14, torej [3,14]. Preiskujemo naprej.




Preiskovanje ostalih poddreves D ne spremeni $[\alpha, \beta]$ v vozlišču D. Rezultat se propagira navzgor. Konec iskanja.

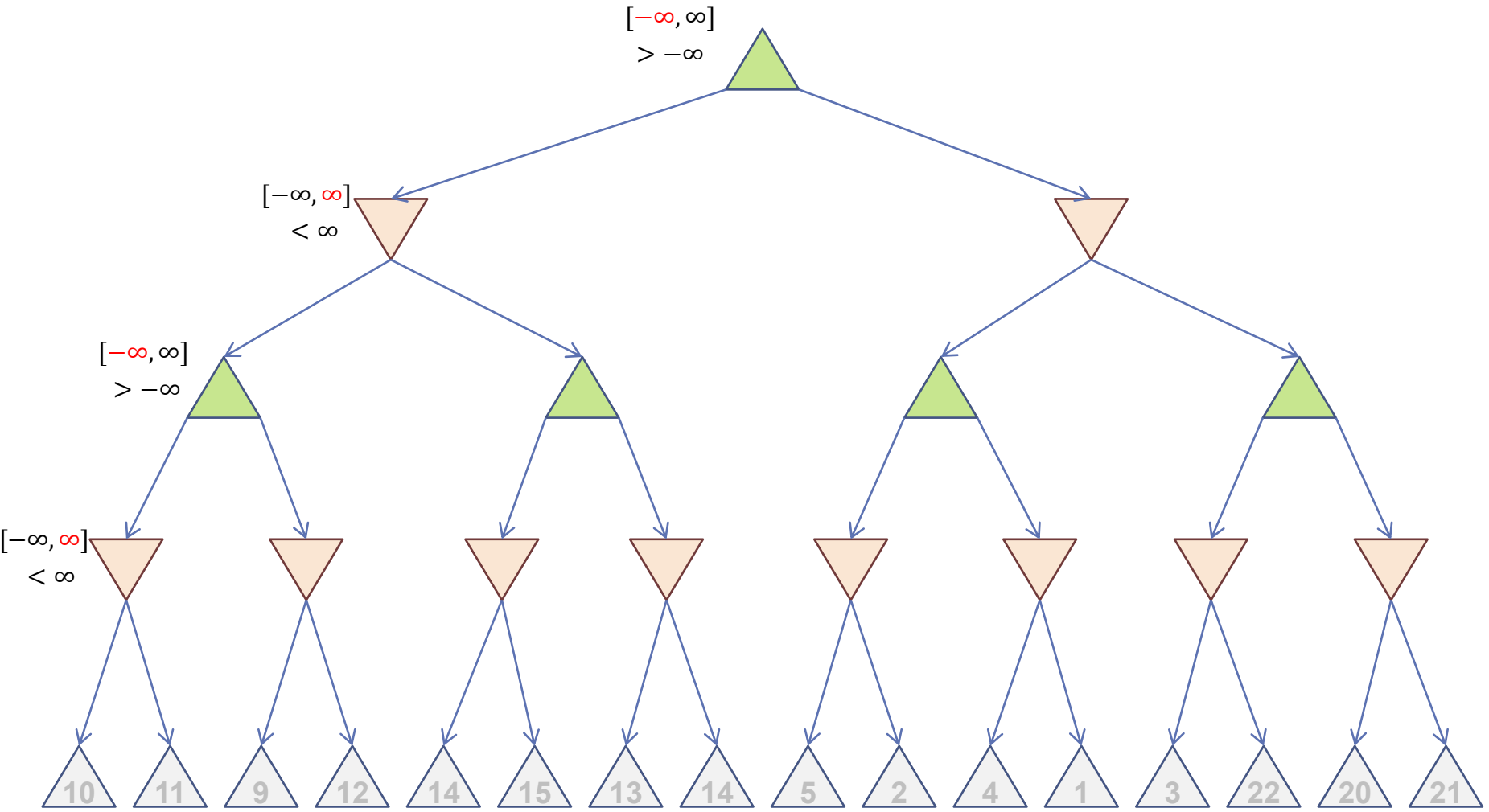


# Drugi primer (poenostavitev zapisa α in β)

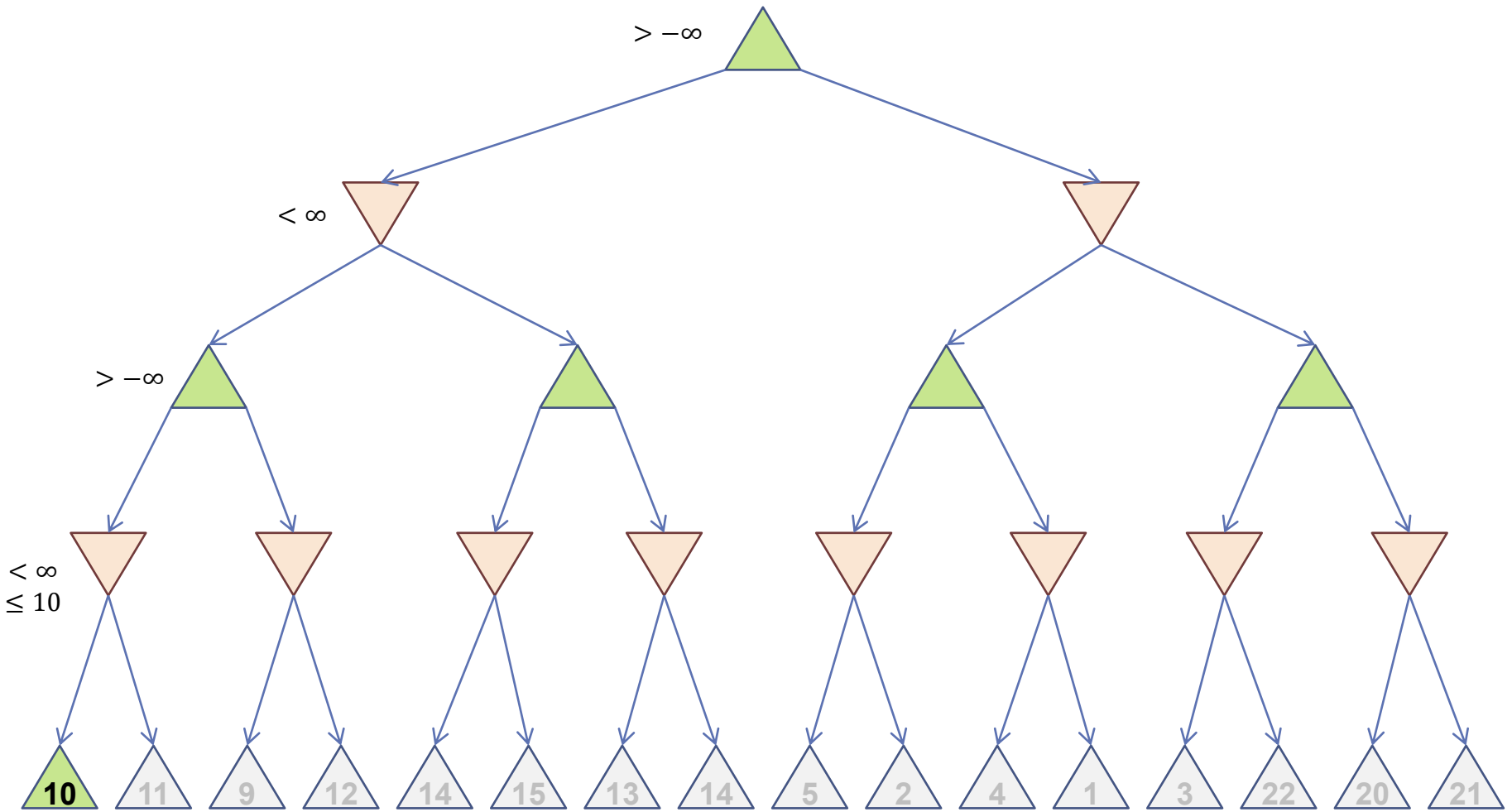



Vir: CSCI 6350 Artificial Intelligence: Minimax and Alpha-Beta Pruning Algorithms

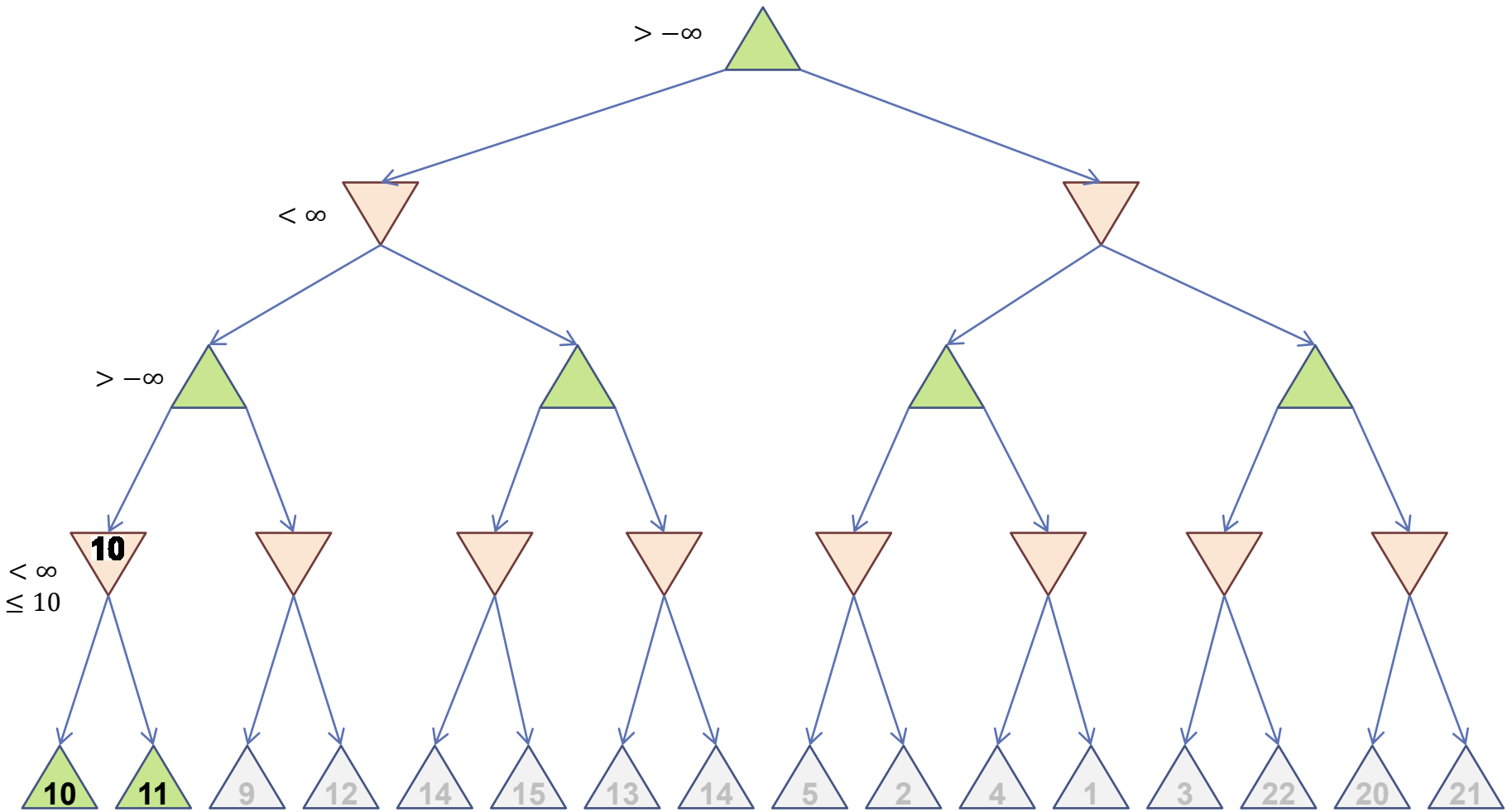
# Drugi primer



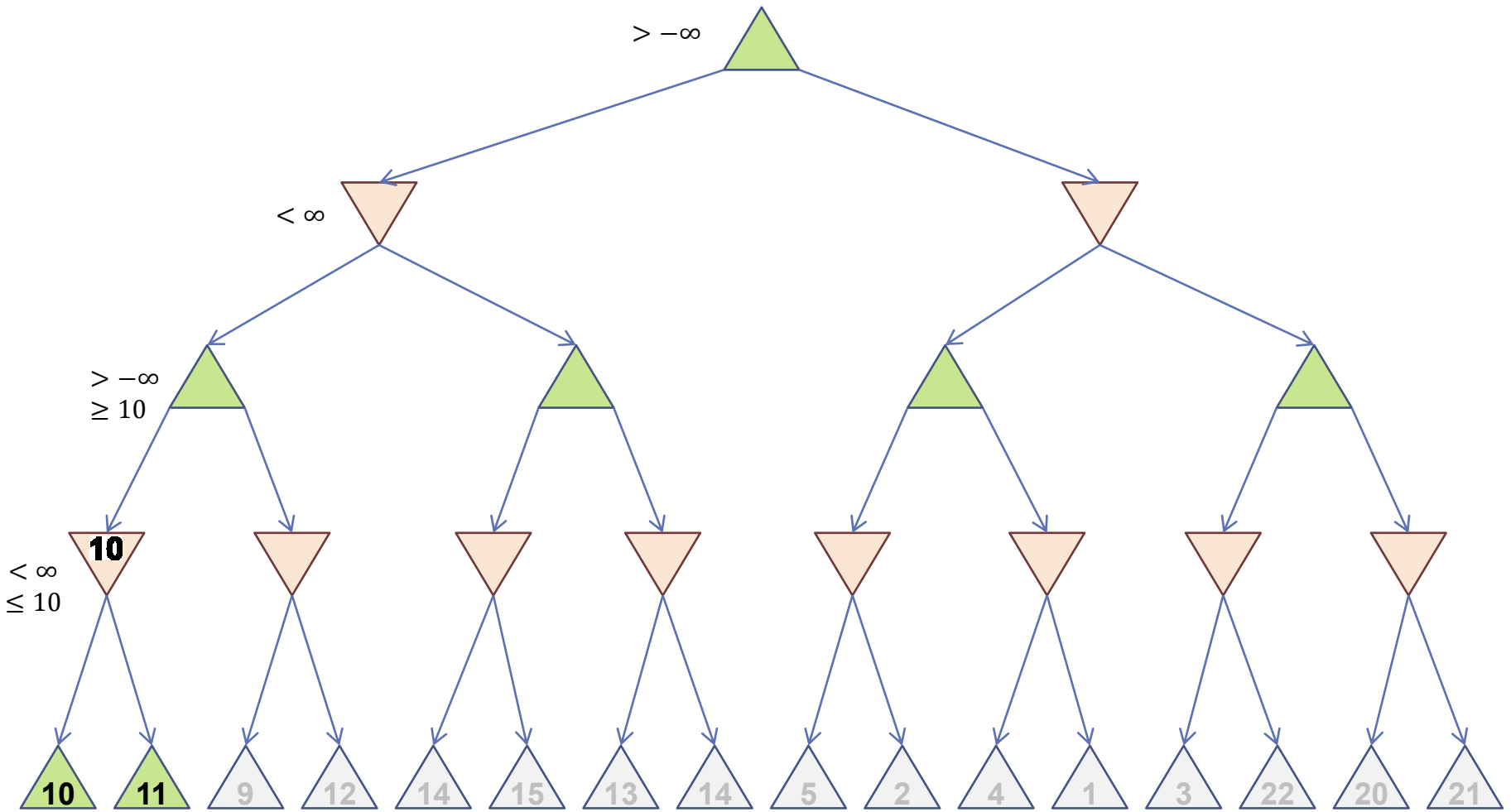
# Drugi primer



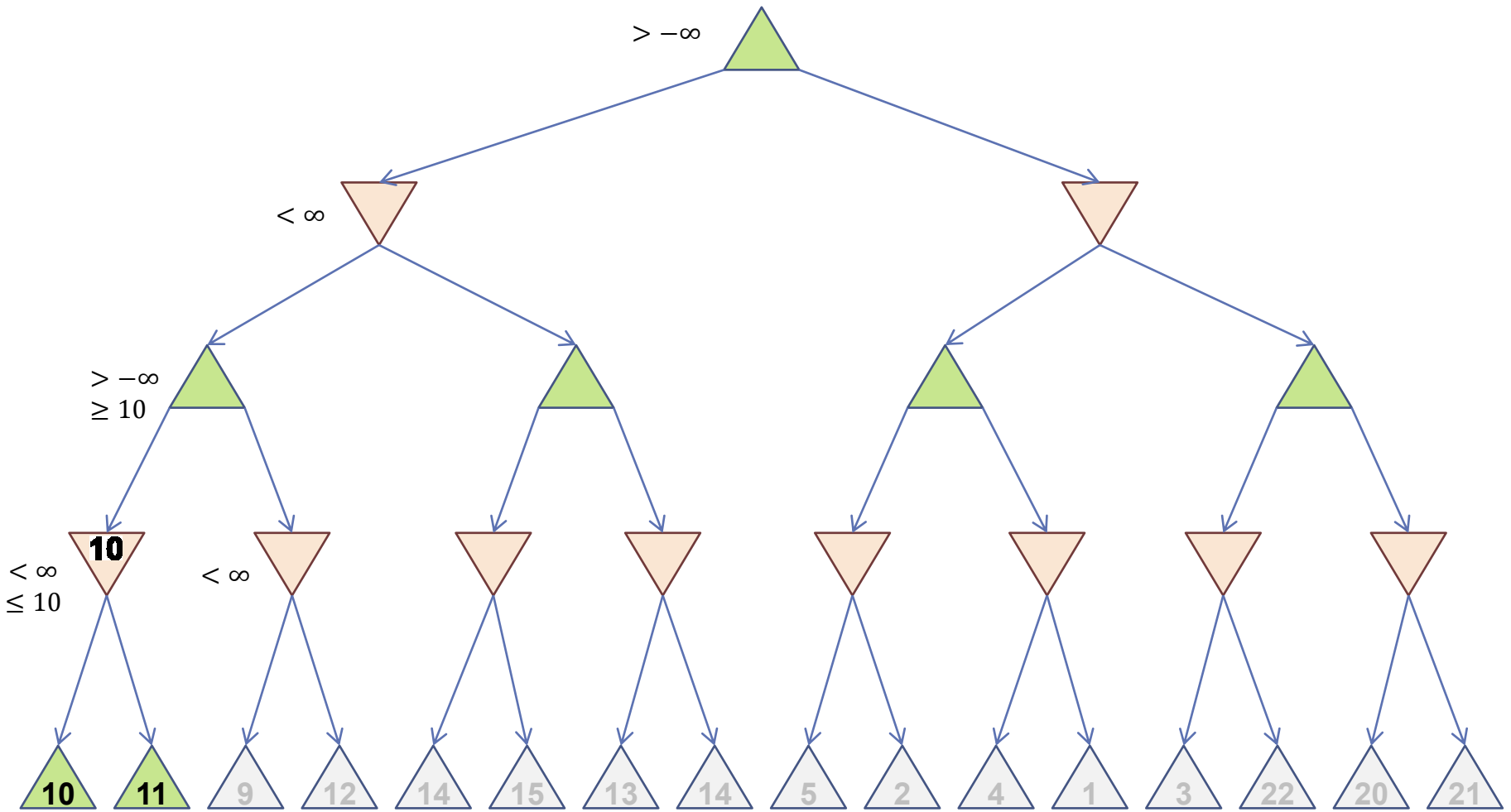
# Drugi primer



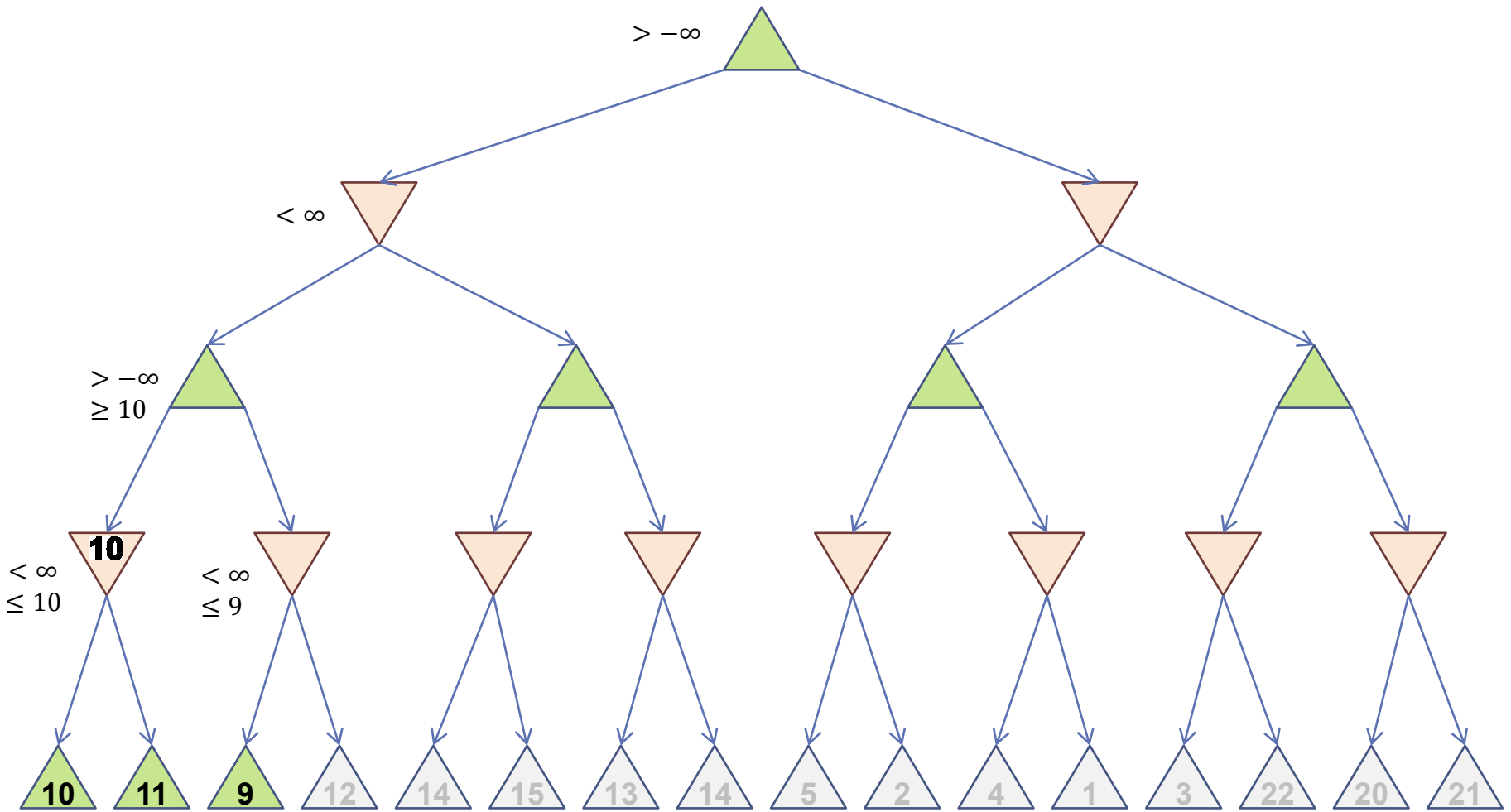
# Drugi primer



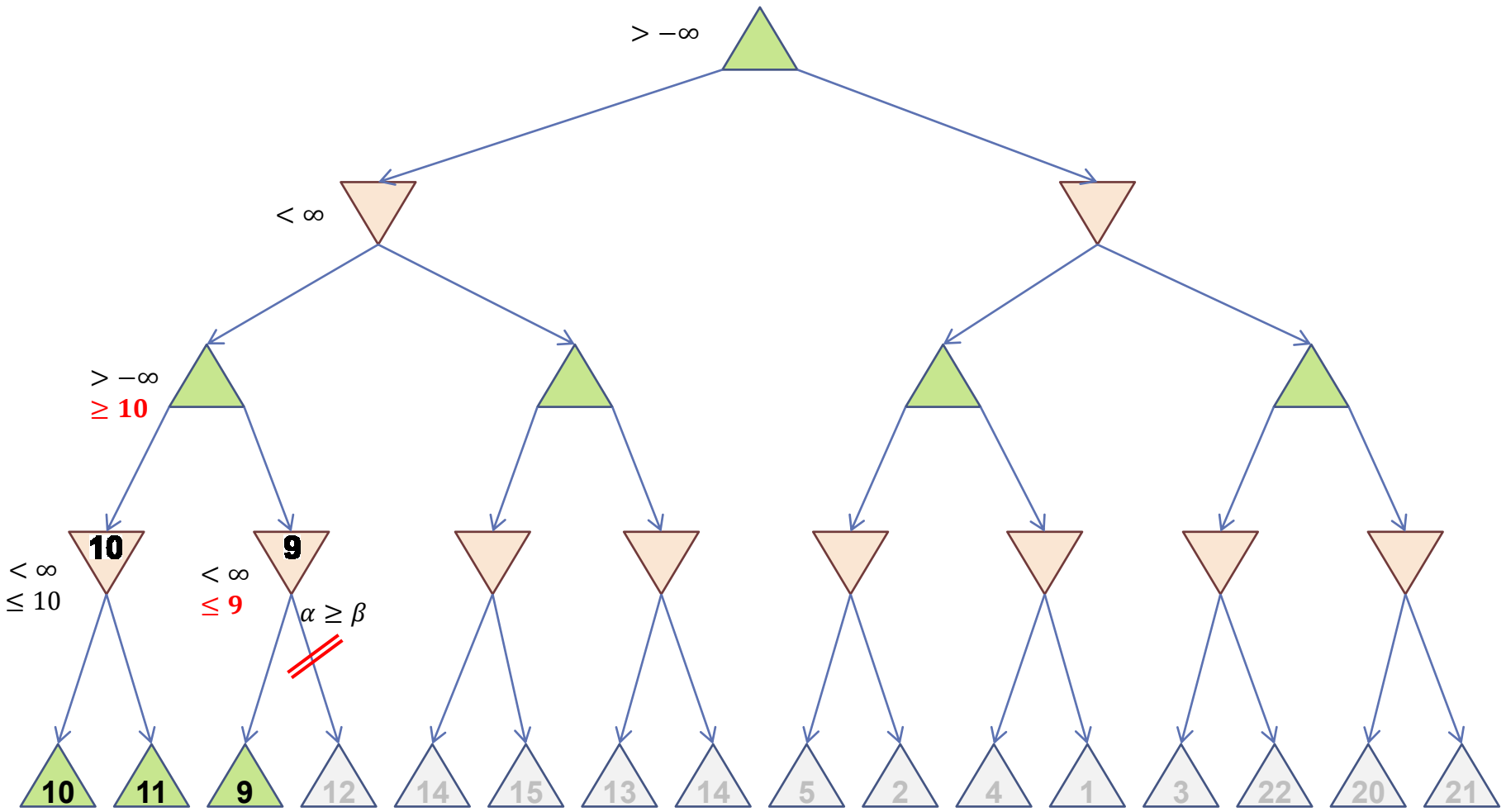
# Drugi primer



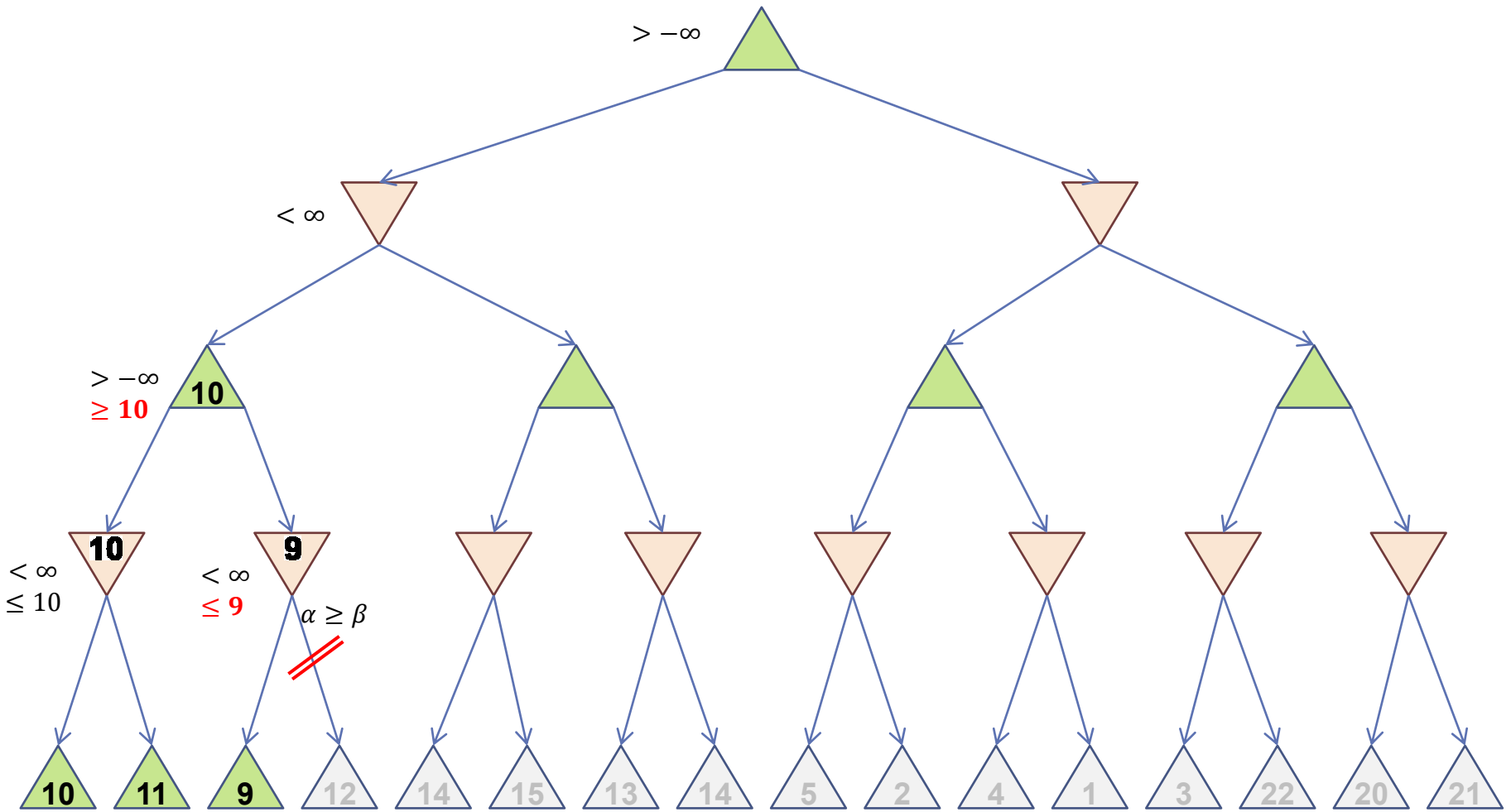
# Drugi primer



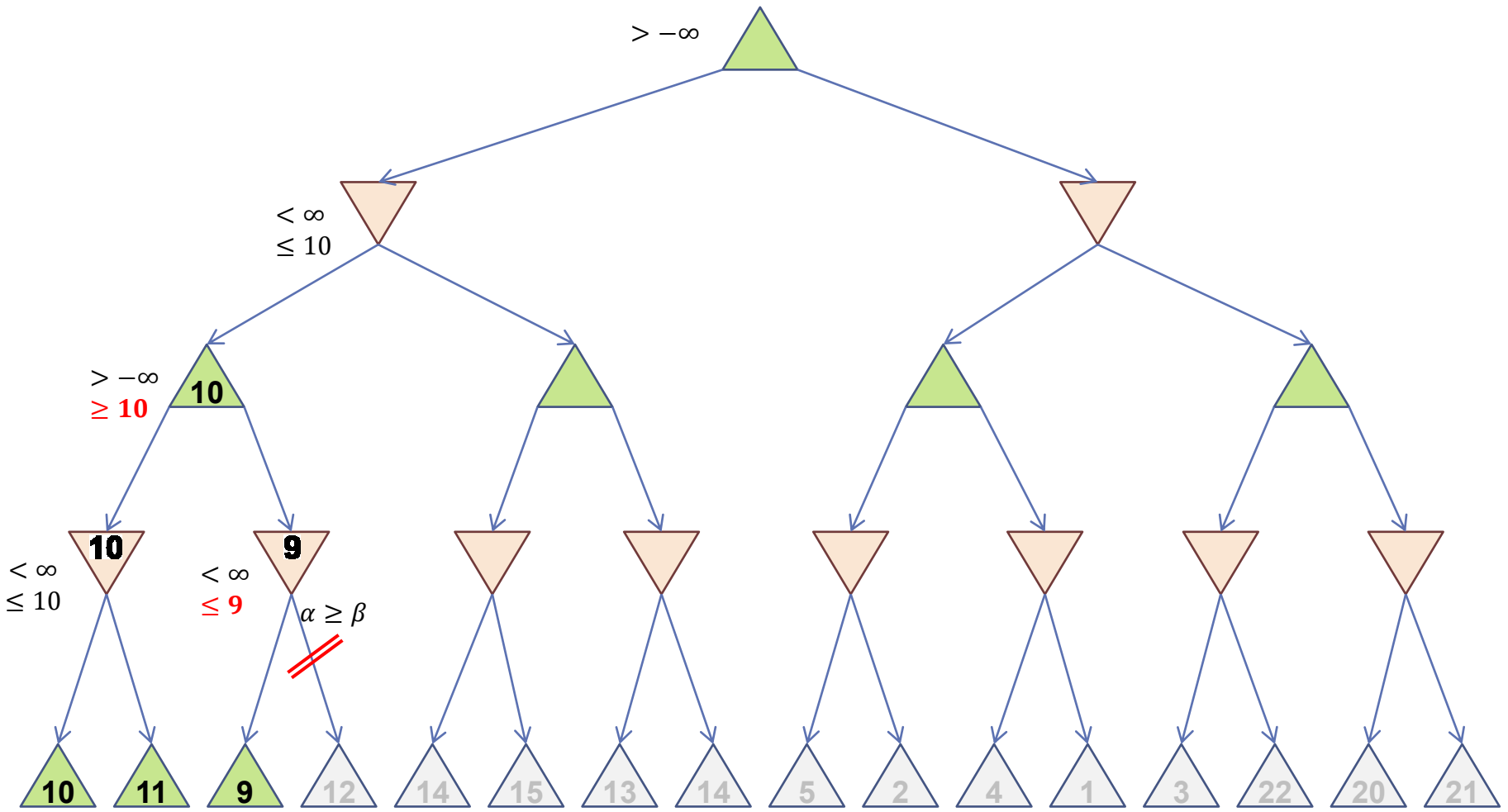
# Drugi primer



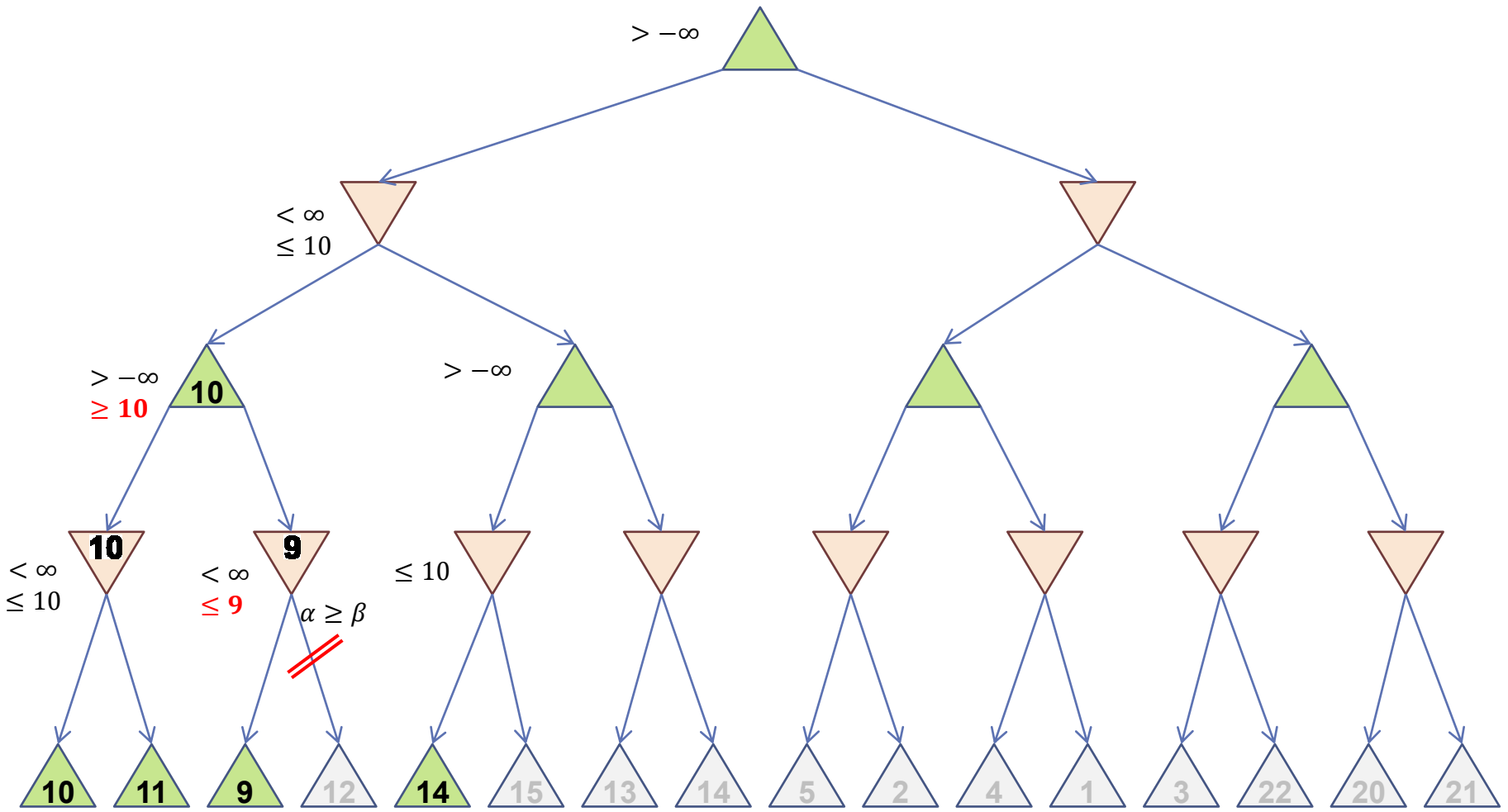
# Drugi primer



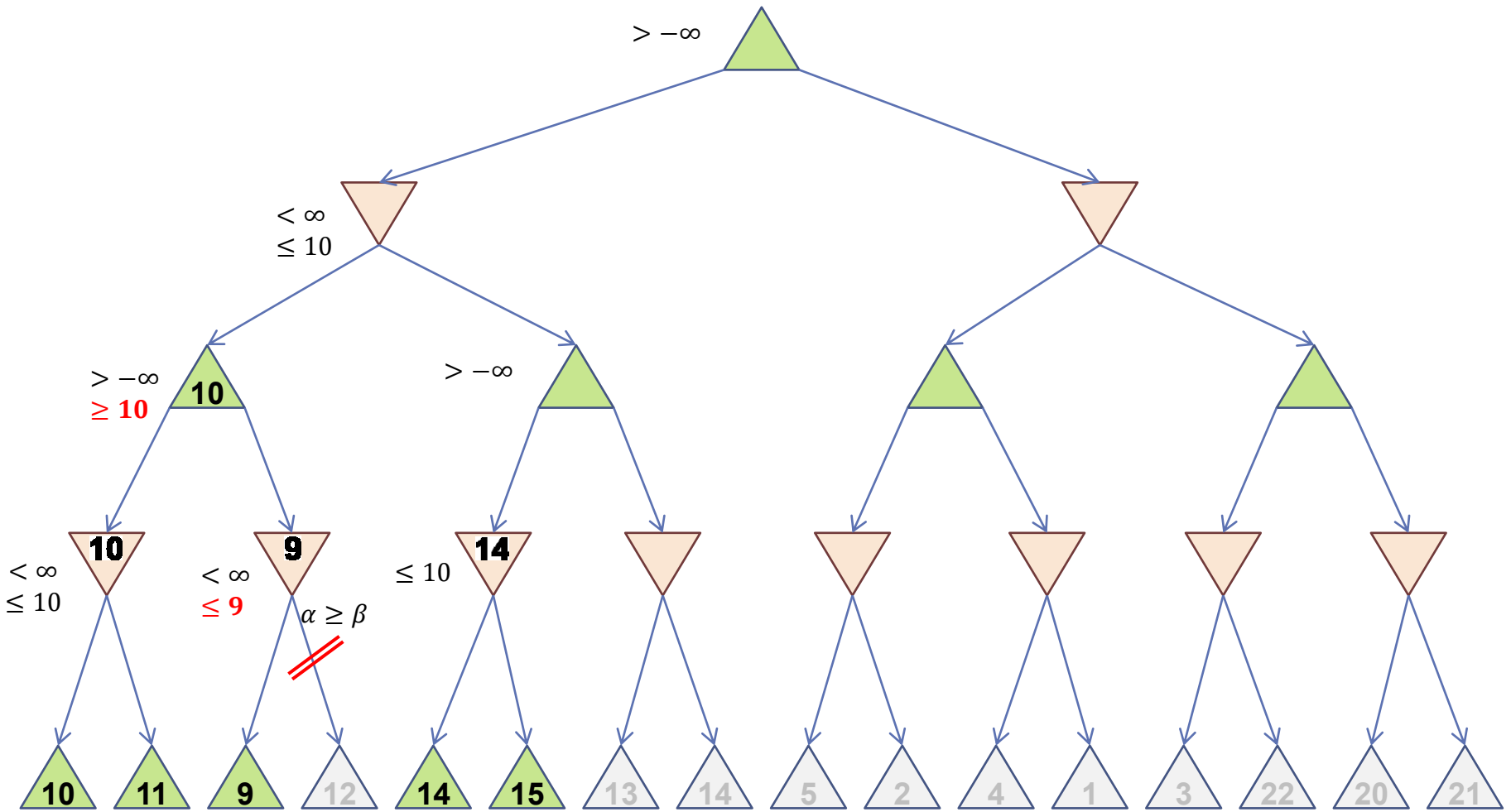
# Drugi primer



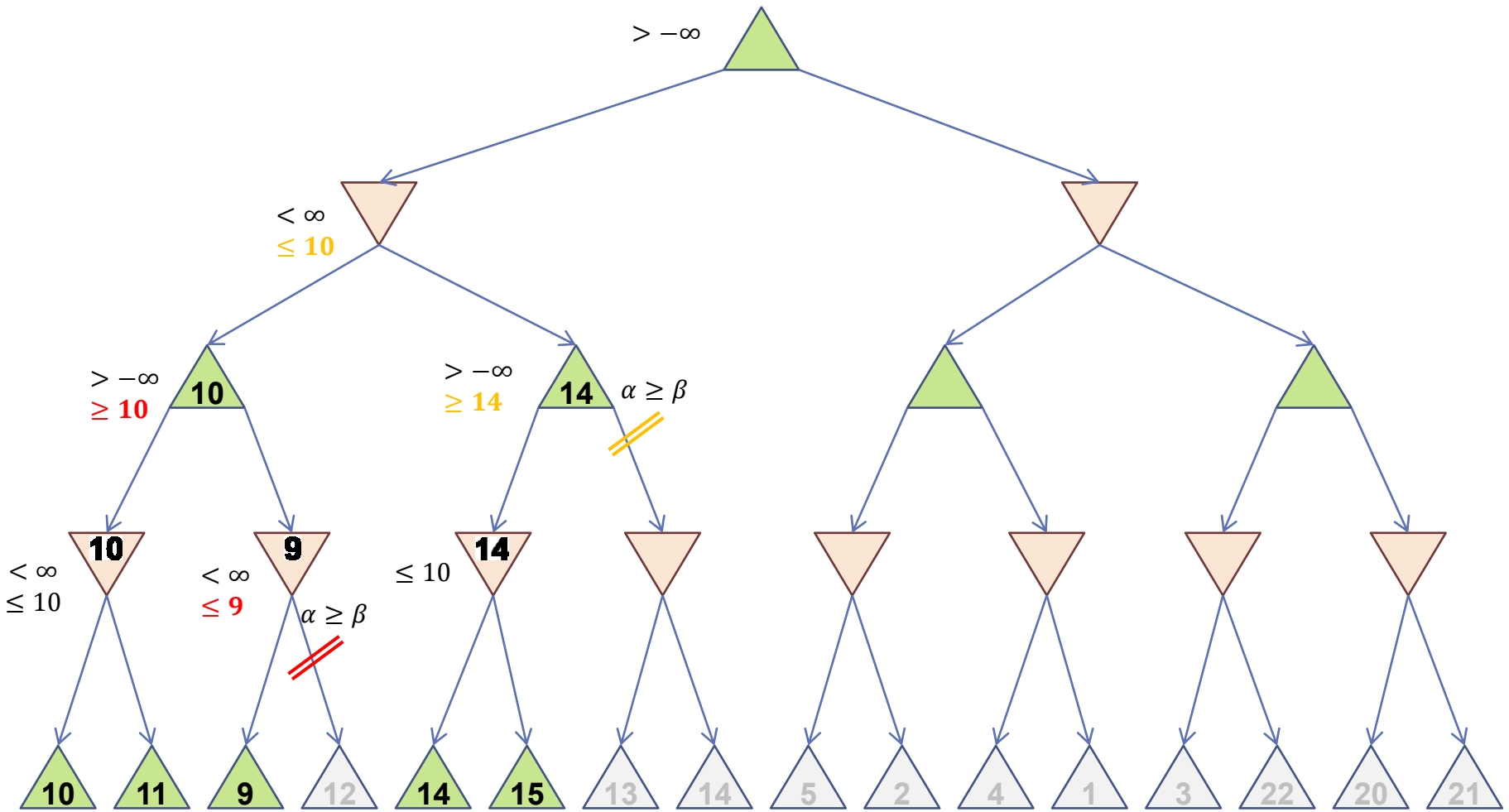
# Drugi primer



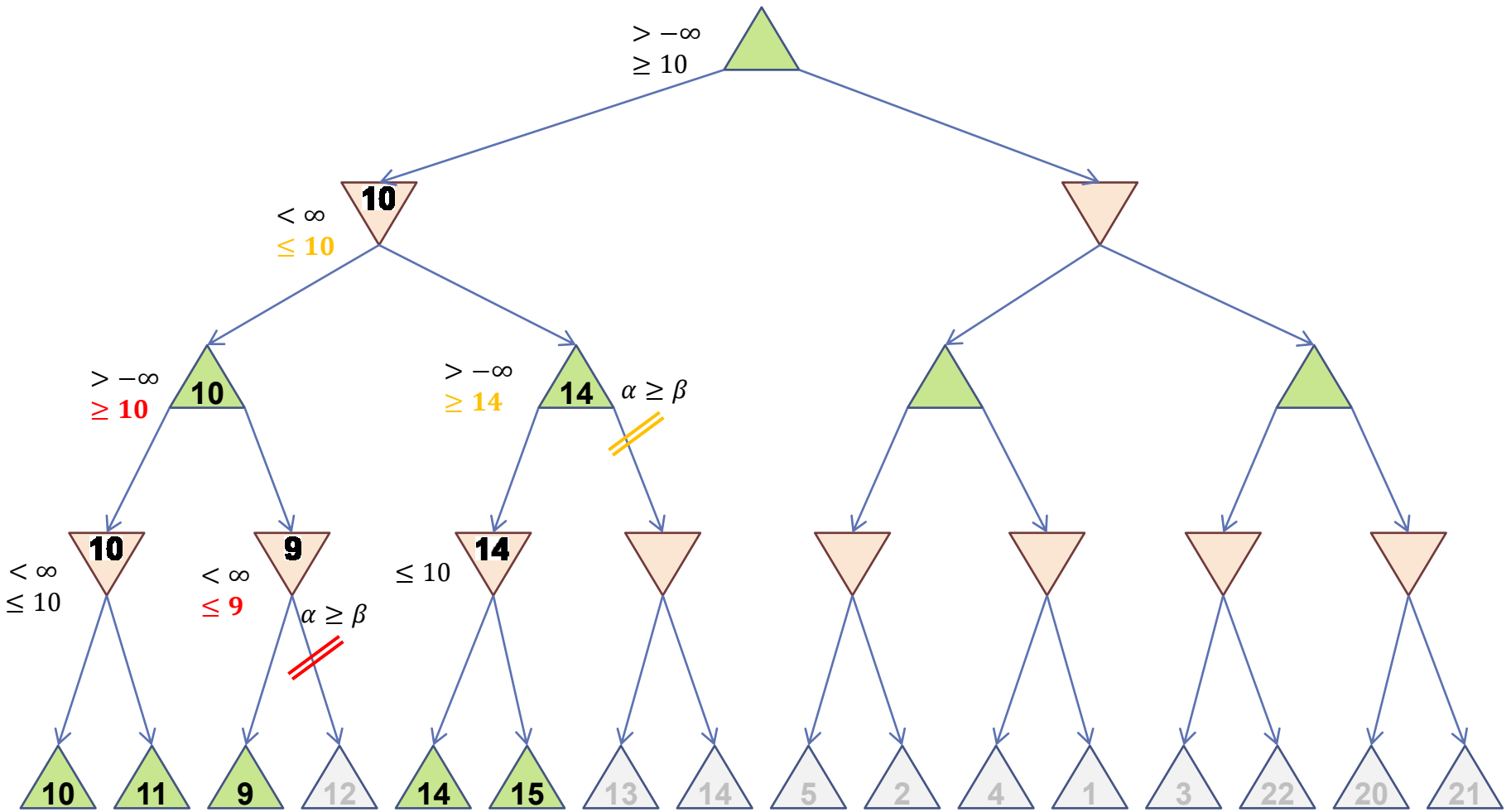
# Drugi primer



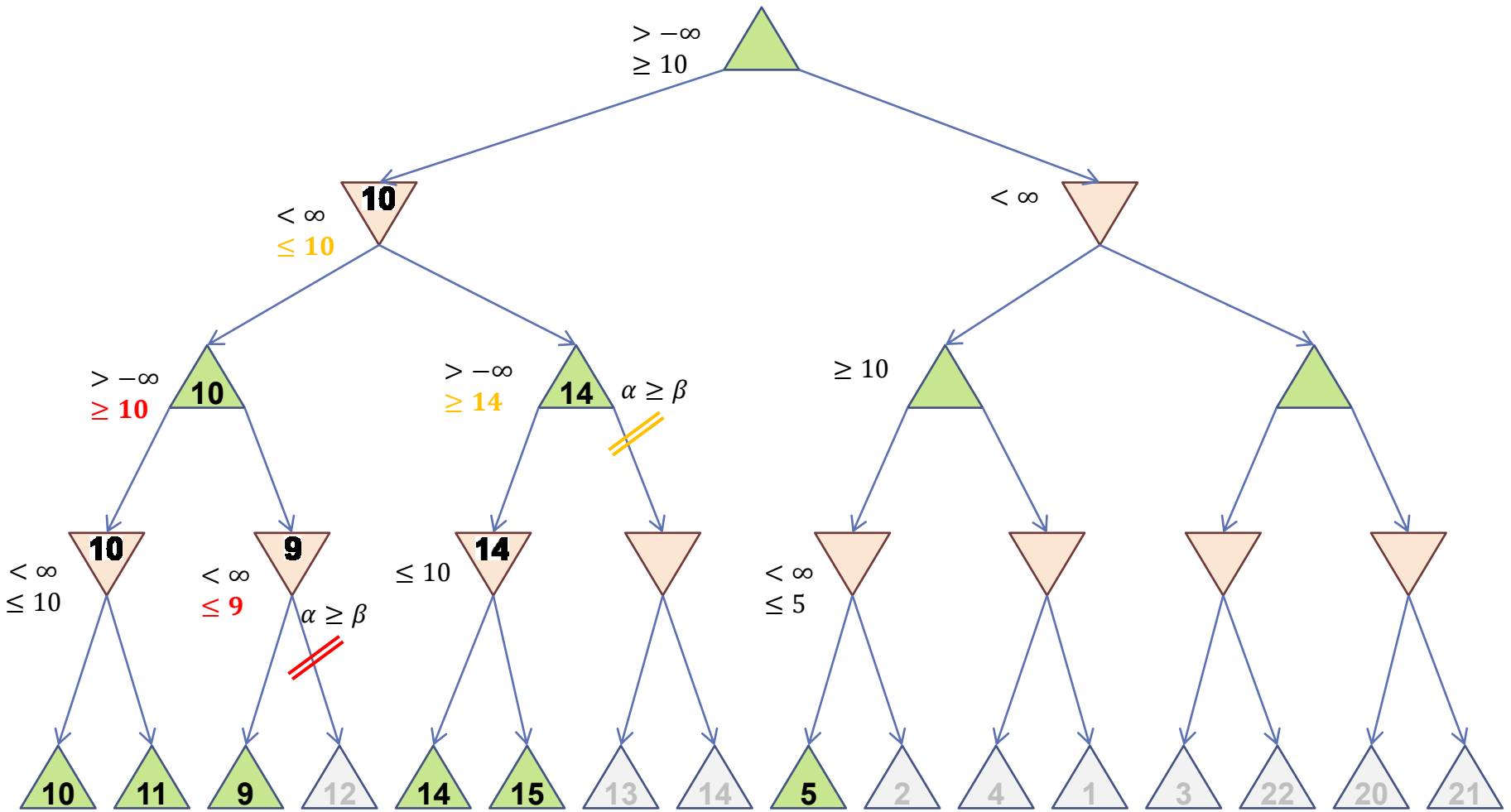
# Drugi primer



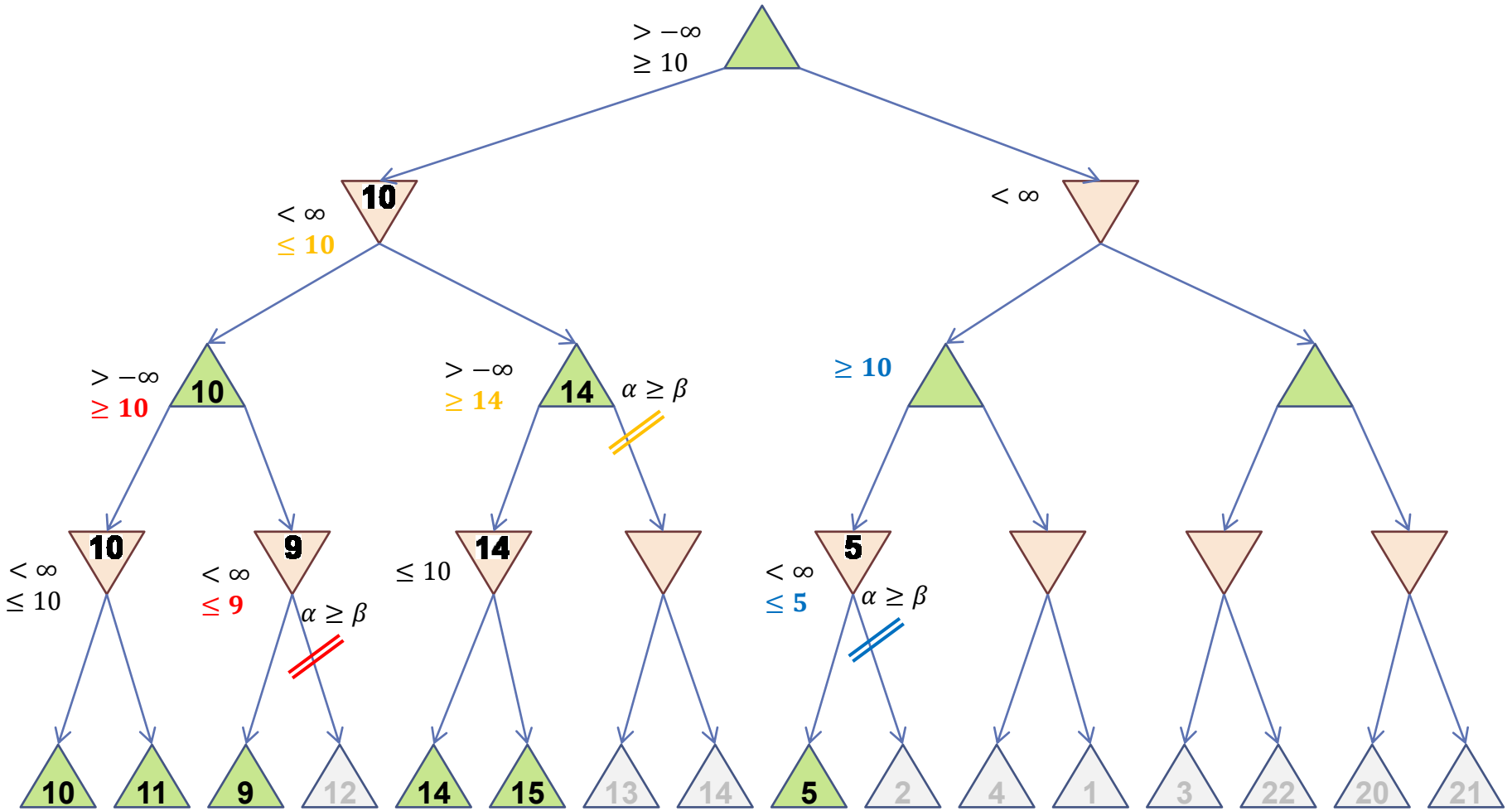
# Drugi primer



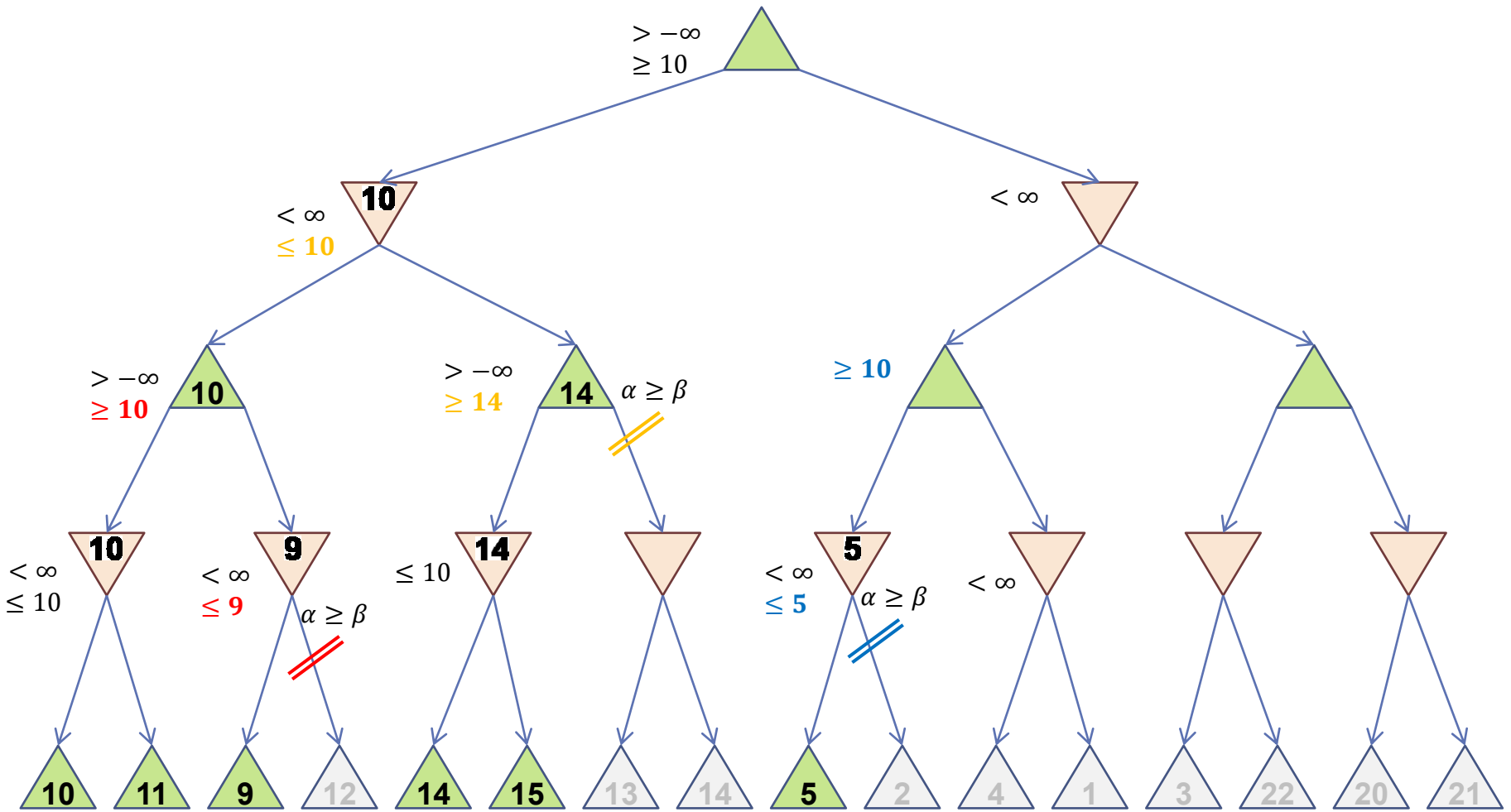
# Drugi primer



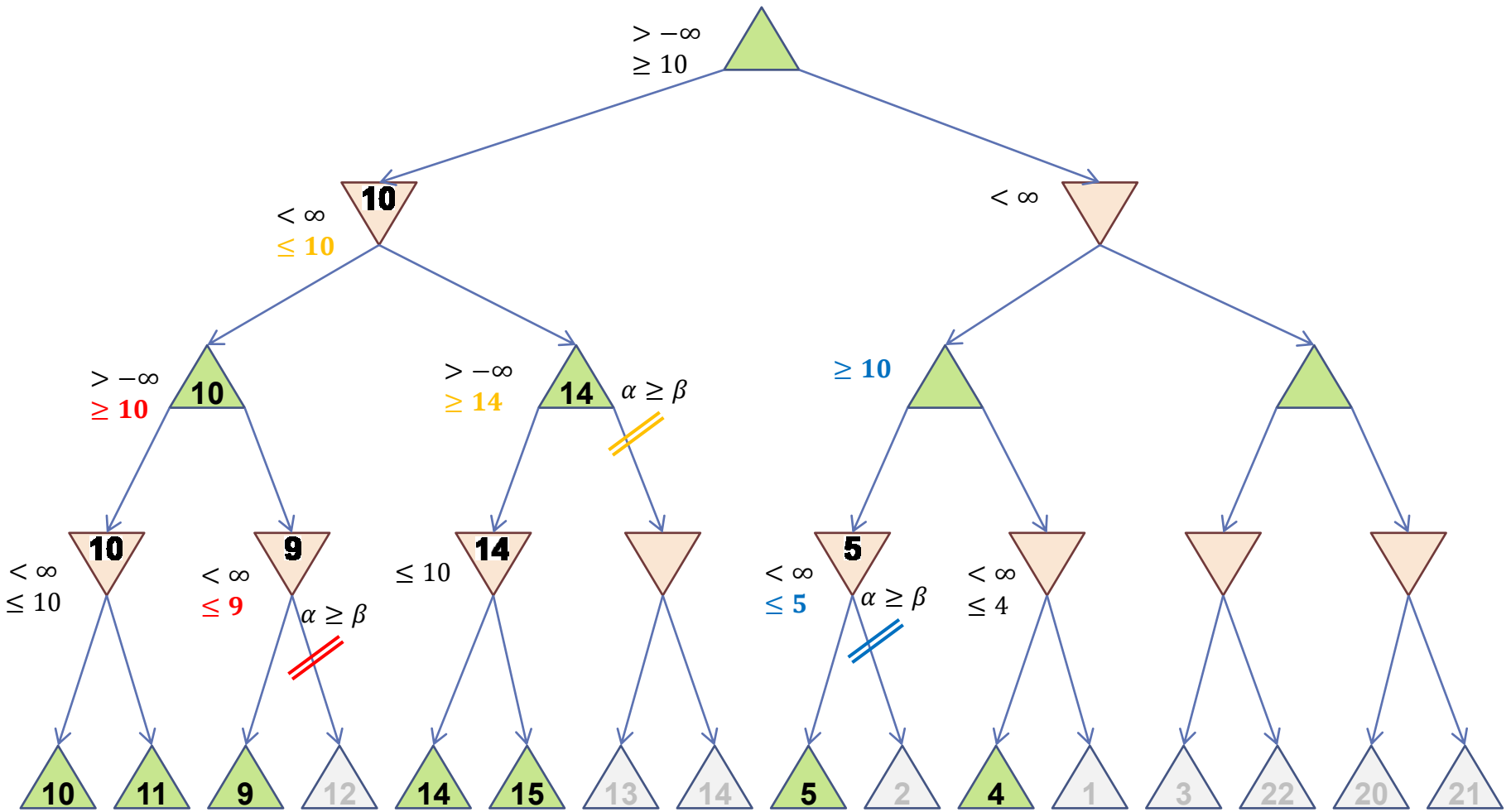
# Drugi primer



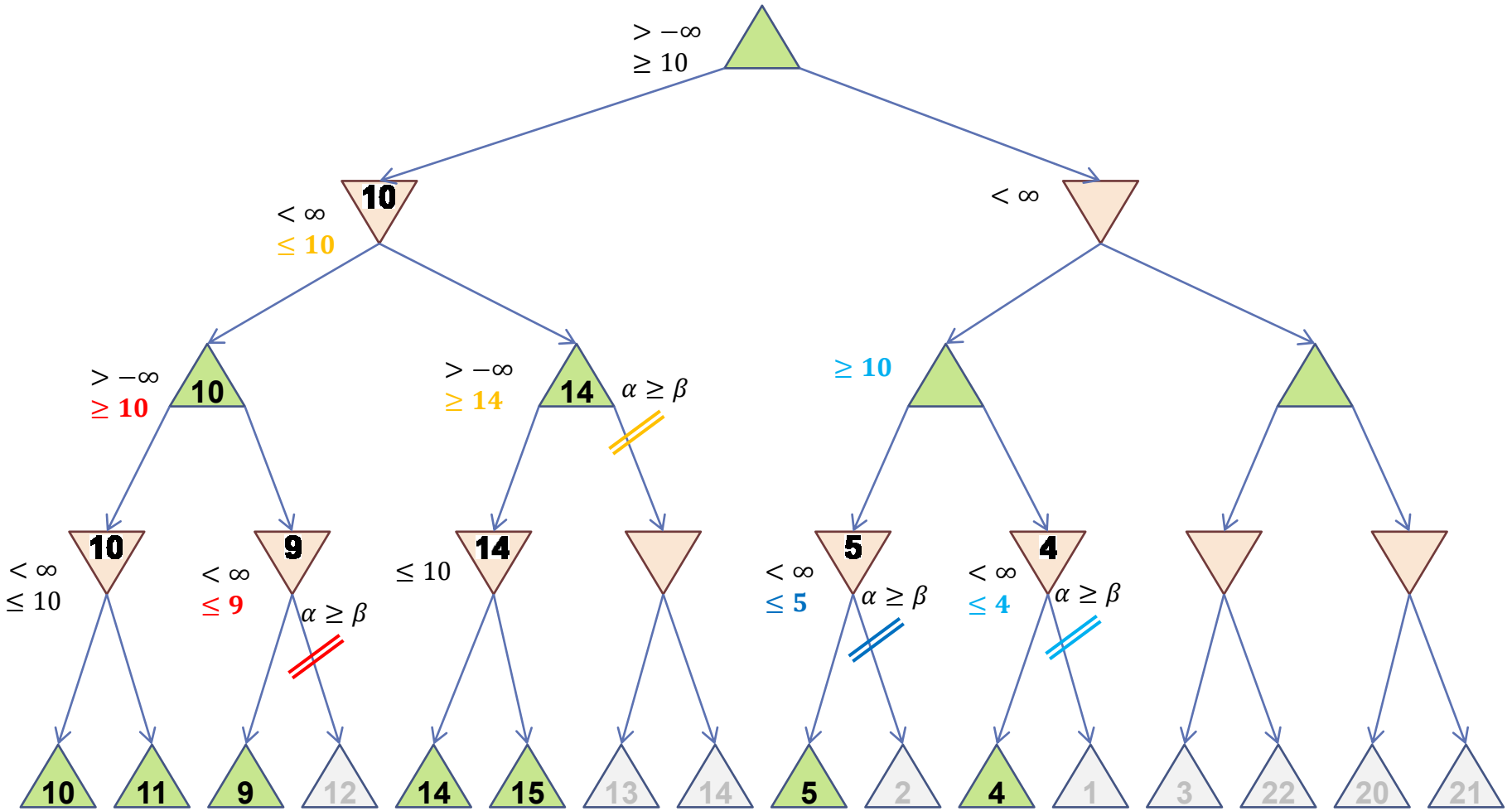
# Drugi primer



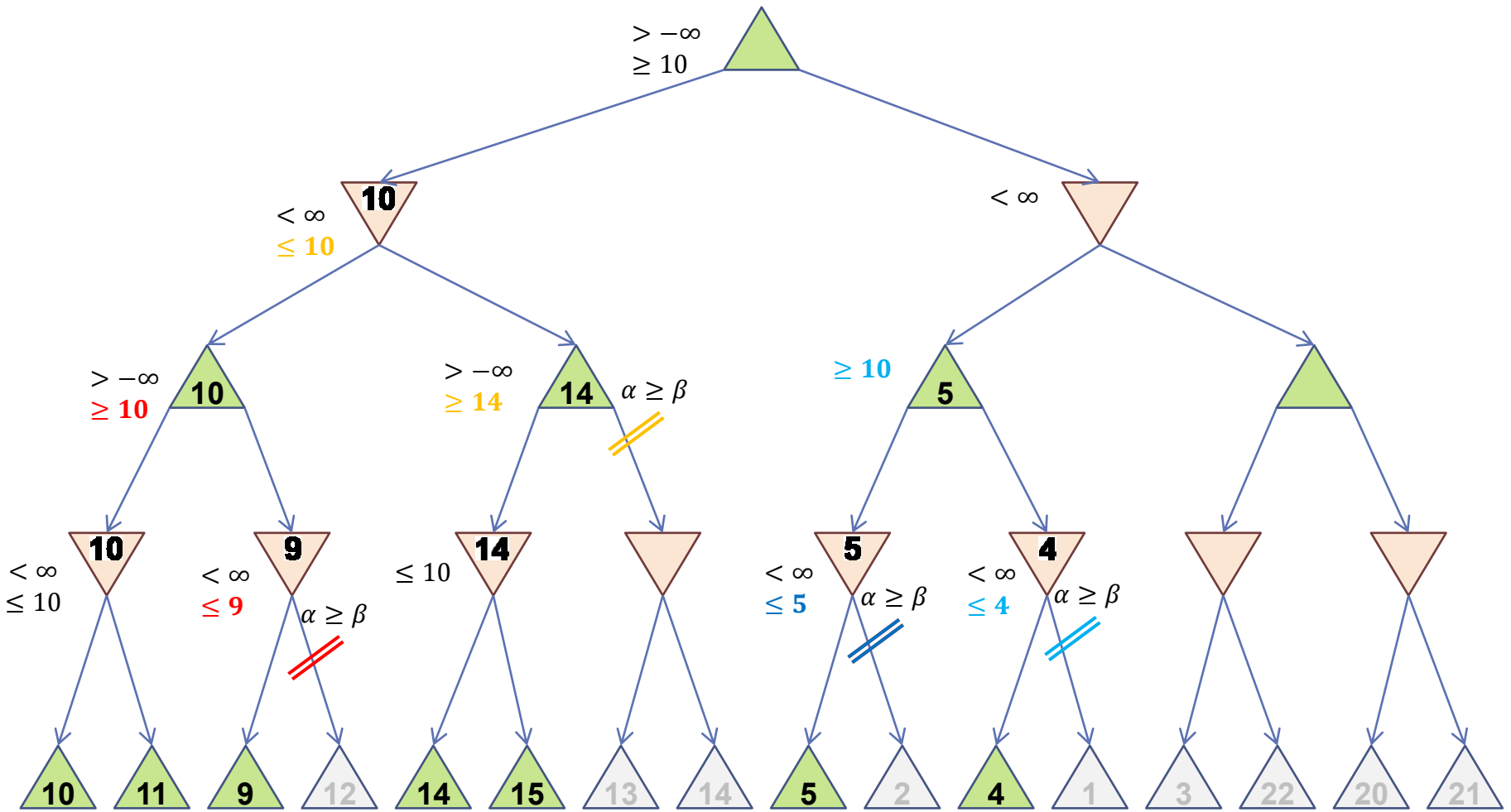
# Drugi primer



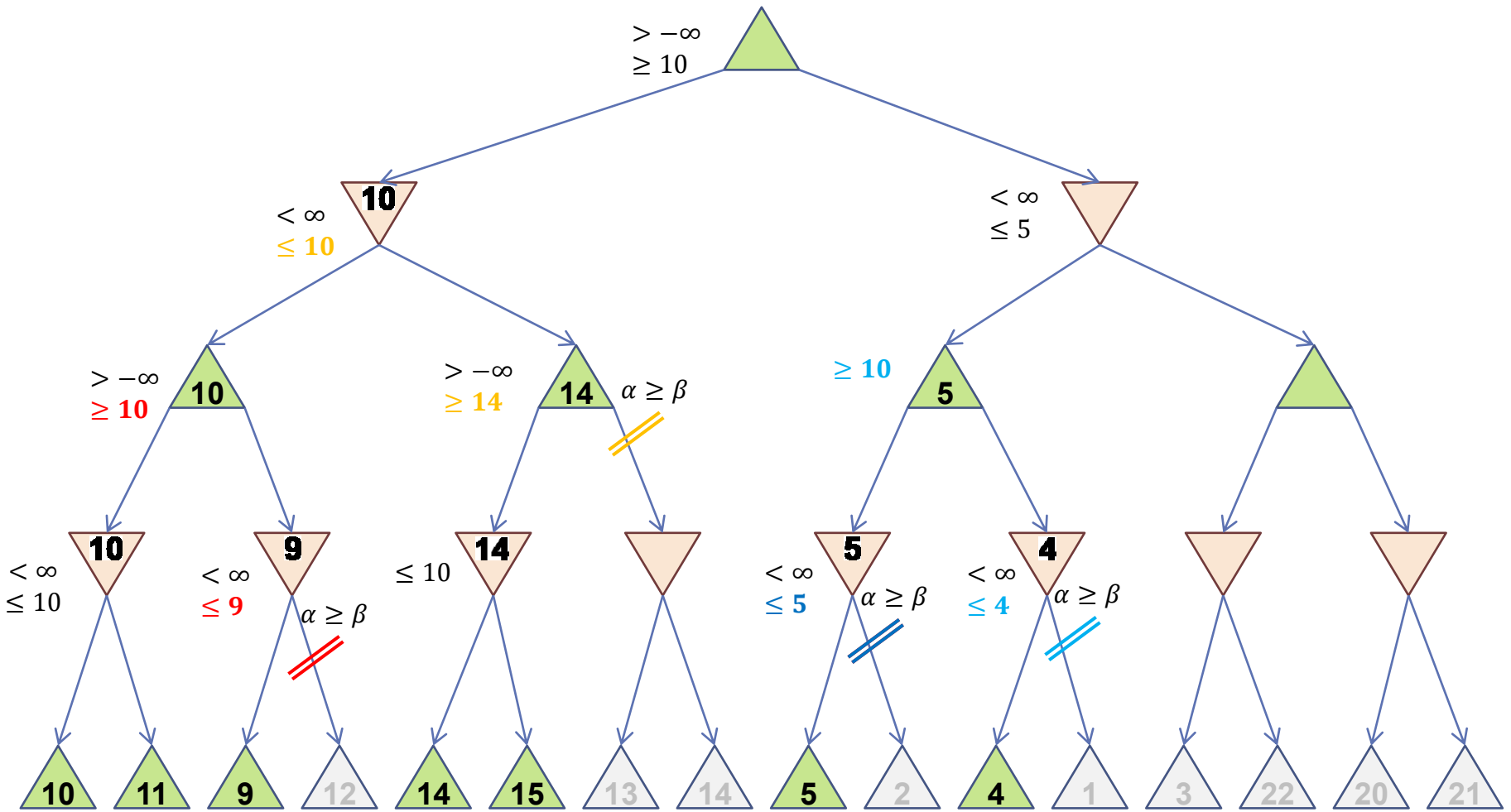
# Drugi primer



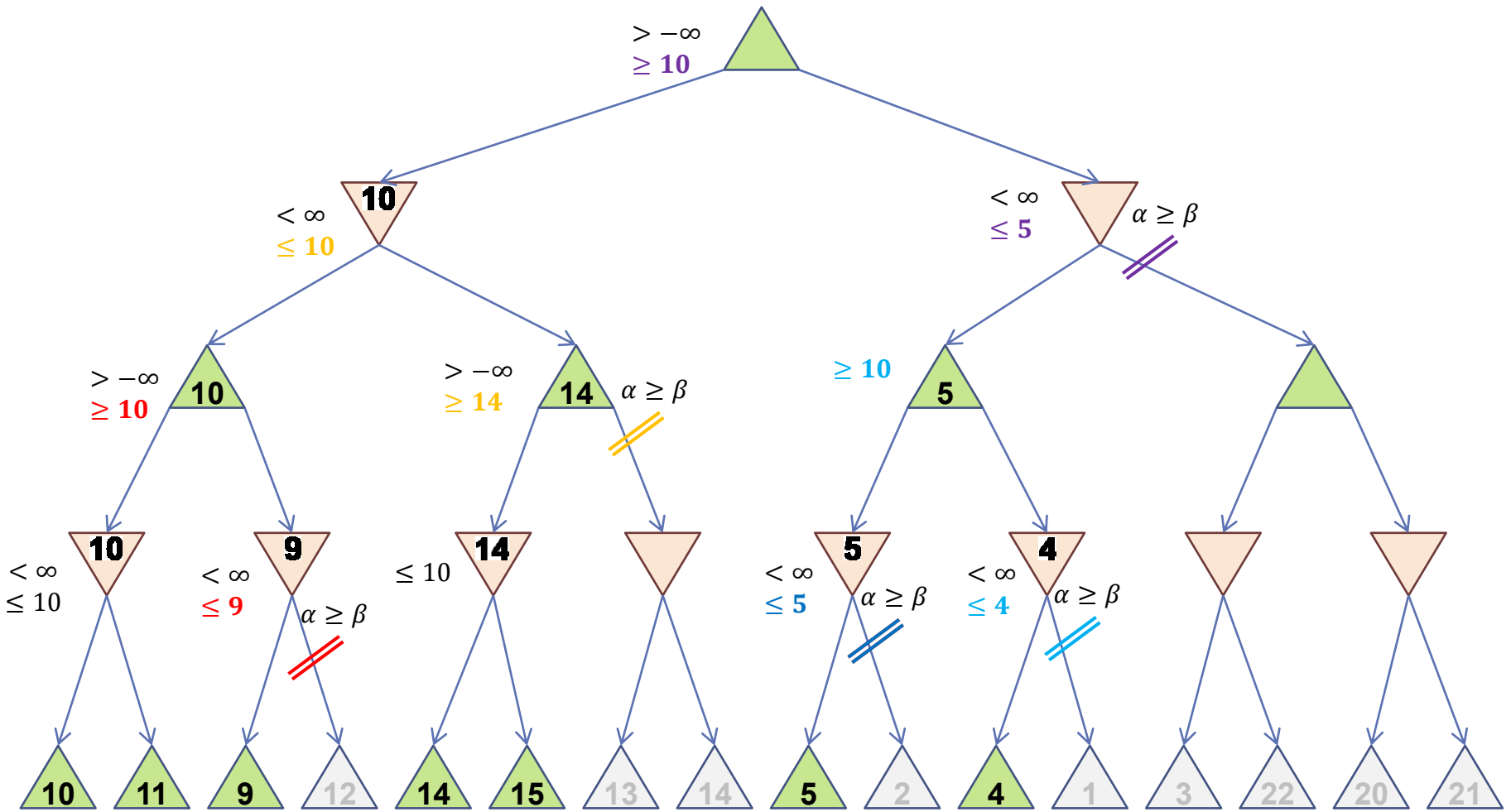
# Drugi primer



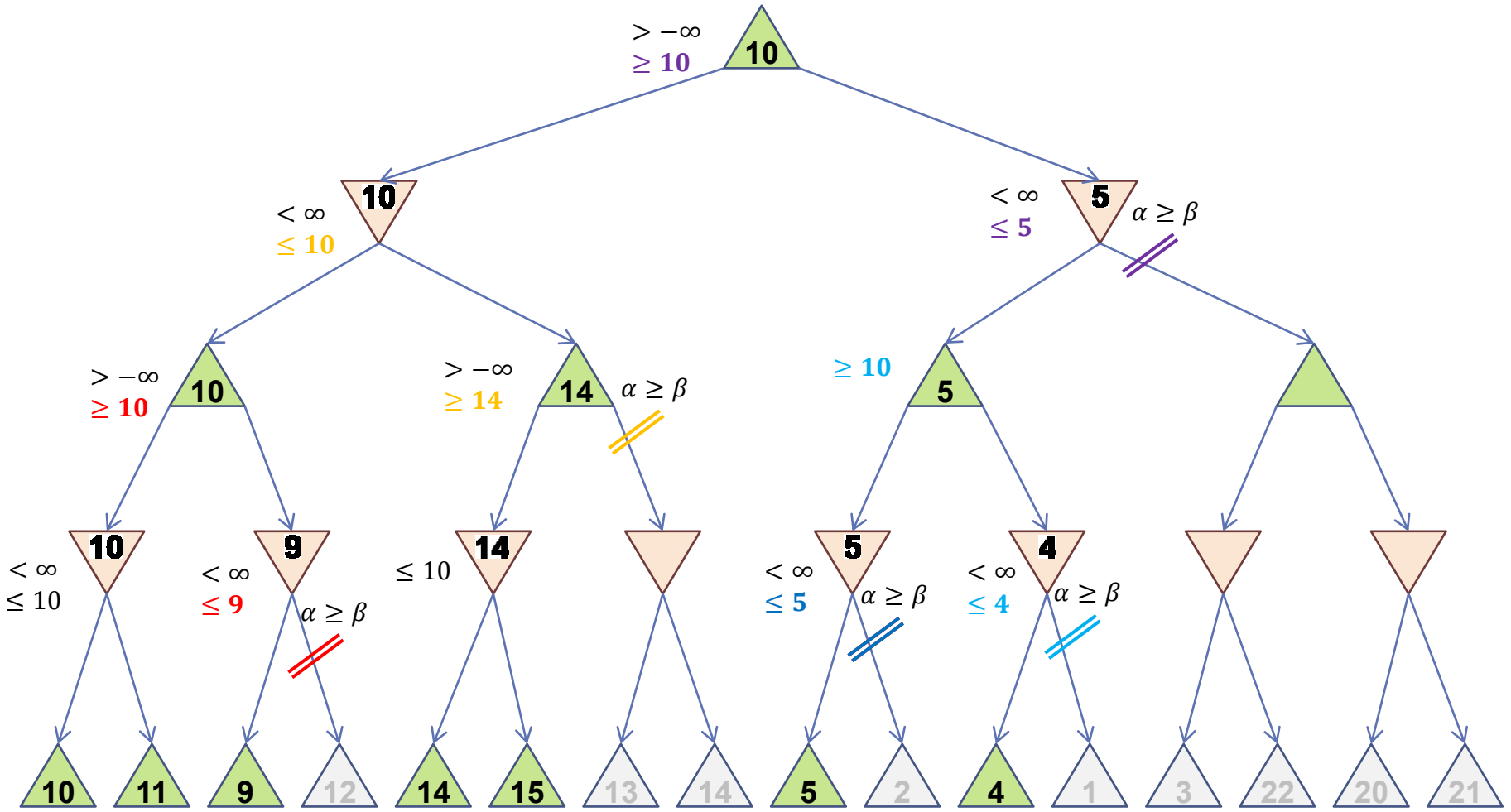
# Drugi primer



# Drugi primer



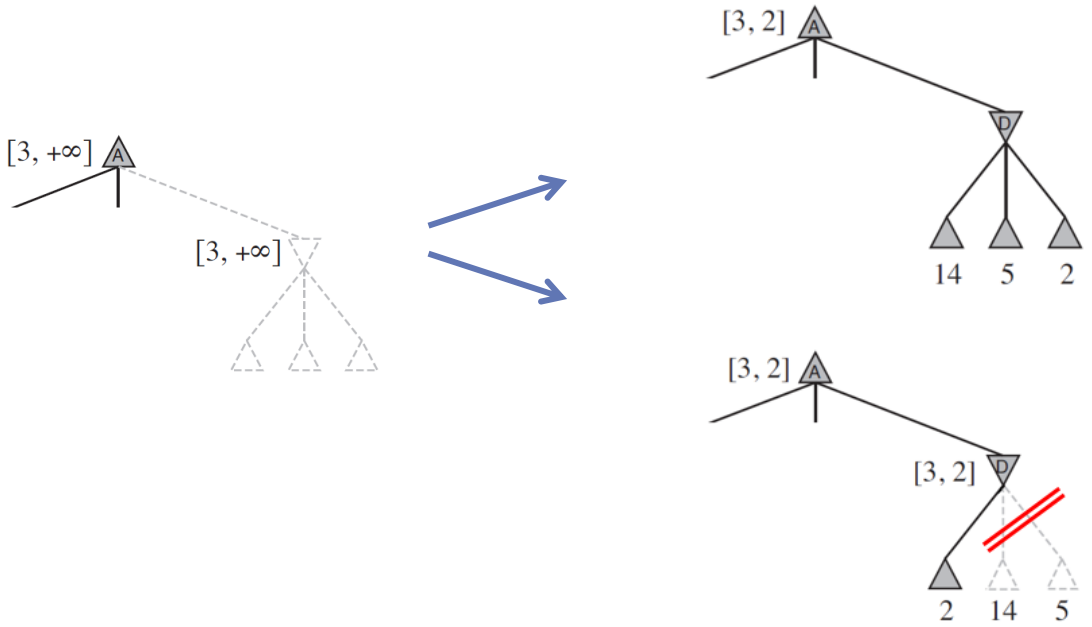
# Drugi primer




Vir: CSCI 6350 Artificial Intelligence: Minimax and Alpha-Beta Pruning Algorithms

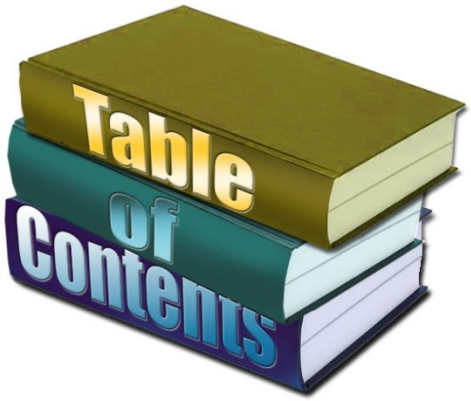
# Lastnosti rezanja alfa-beta

- potek algoritma je odvisen od vrstnega reda naslednikov




- rezanje alfa-beta zniža časovno zahtevnost algoritma MINIMAX z zahtevnosti z $O(b^m)$ na $O\left(b^{m/2}\right)$
  - šah: faktor vejanja se zniža s 35 na 6
  - v splošnem: možna globina preiskovanja se podvoji
  - uporaba strategij za določanje vrstnega reda preiskovanja naslednikov (uporabi se lahko znanje iz preteklih iger)

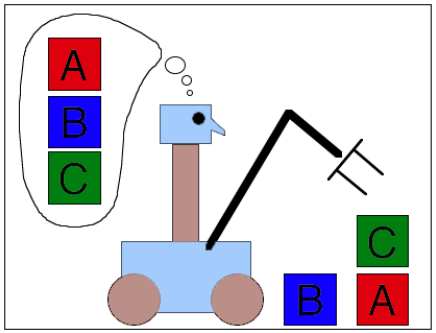
# Pregled




- igranje iger
  - predstavitev problema
  - algoritem *MINIMAX*
  - rezanje alfa-beta

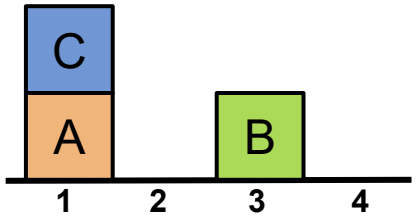
- planiranje
  - planiranje s "klasičnim" preiskovanjem prostora stanj
  - planiranje s sredstvi in cilji
  - planiranje z regresiranjem ciljev

# Planiranje




- postopek načrtovanja akcij, ki dosežejo želene cilje
- plan: zaporedje akcij, ki pripelje od začetnega do končnega stanja
- praktična uporaba:
  - logistični problemi, planiranje operacij
  - robotika
  - razporejanje opravil, urniki
- Za **formalni opis problema** planiranja potrebujemo:
  - definicijo **začetnega stanja** in ciljnih stanj
  - definicijo **akcij** (angl. *actions*) z njihovimi predpogoji (angl. *conditions*) in učinki (angl. *effects*)
  - definicijo **omejitev** (angl. *constraints*)
  - predpostavko zaprtega sveta:
    - za vsa dejstva, ki niso omenjena, predpostavimo, da niso resnična
    - akcija nima učinkov, ki niso omenjeni
- za formalizacijo problema planiranja uporabljamo predstavitev s formalnim jezikom:
  - STRIPS: Stanford Research Institute Problem Solver (1971)
  - ADL: Action Description Language (1986)
  - PDDL: Planning Domain Definition Language (1998-2005)

# STRIPS / PDDL




stanje: `[on(c,a), on(a,1), on(b,3), clear(c), clear(2), clear(b), clear(4)]`
cilj: `[on(a,b), on(b,c)]`

akcija (akcijska shema):

`move(Block, From, To)`
predpogoji: `[clear(Block), on(Block,From), clear(To)]`
učinki
add: `[on(Block,To), clear(From)]`
del: `[on(Block,From), clear(To)]`
omejitve: `[block(Block), To≠From, To≠Block, From≠Block]`

**atomi (literali, končni objekti)**

**spremenljivke (z veliko začetnico), so eksistenčno kvantificirane (∃)**

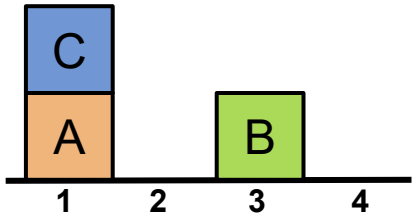
**učinki akcij in stanja so konjunkcije trditev**

Rezultat izvedbe akcije $a$ v stanju $s$ je stanje $s1$:

$$s1 = (s - del(a)) \cup add(a)$$

Kako se predpostavka zaprtega sveta odraža na zapisu stanja in akcij?

# STRIPS / PDDL




PDDL uporablja sorodno (alternativno) sintakso:

- dovoljuje negacije učinkov
- spremenljivke z malimi črkami, atomi z velikimi (ravno obratno)
- imena akcij z velikimi črkami

STRIPS:

```
move(Block, From, To)
predpogoji:      [clear(Block), on(Block,From), clear(To)]
učinki-add       [on(Block,To), clear(From)]
učinki-del:      [on(Block,From), clear(To)]
omejitve:        [block(Block), To≠From, To≠Block, From≠Block]
```

PDDL:

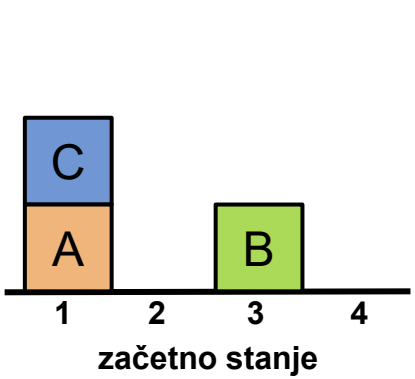
```
Action(Move(block, from, to),
     PRECOND: Clear(block) ∧ On(block, from) ∧ Clear(to)
     EFFECT: On(block,to) ∧ Clear(from) ∧ ¬On(block,from) ∧ ¬clear(to))
```

# Primer 1: Svet kock

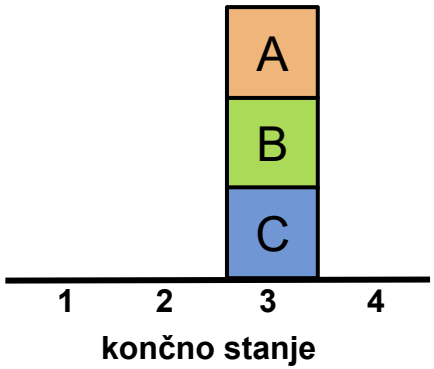
Plan: zaporedje akcij, ki pripelje od začetnega do končnega stanja

Možna rešitev:

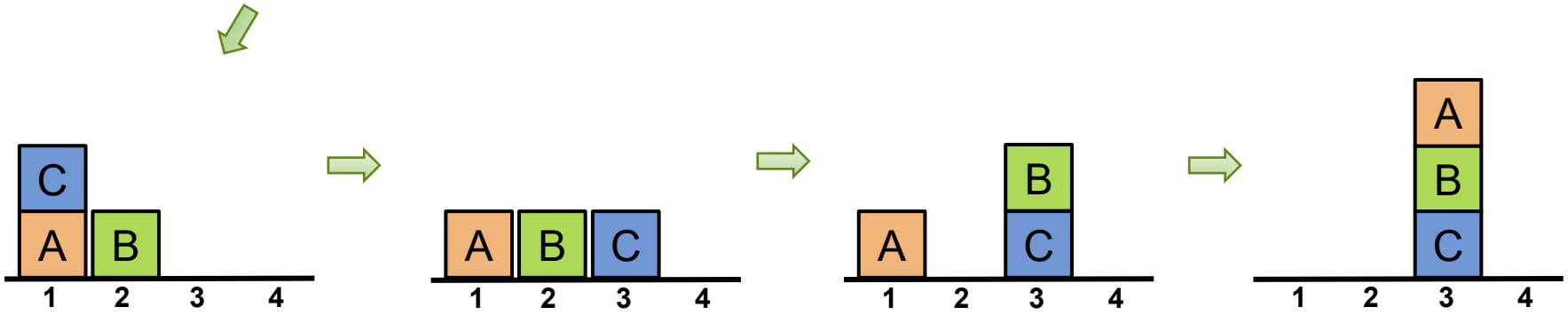
```
plan = [move(b,3,2), move(c,a,3), move(b,2,c), move(a,1,b)]
```



začetno stanje



končno stanje



# Primer 2: menjava pnevmatike (PDDL)

**Init**(Tire(Flat) ∧ Tire(Spare) ∧ At(Flat, Axle) ∧ At(Spare, Trunk))

**Goal**(At(Spare, Axle))

**Action**(Remove(obj, loc),
    PRECOND: At(obj, loc)
    EFFECT: ¬At(obj, loc) ∧ At(obj, Ground))

**Action**(PutOn(t, Axle),
    PRECOND: Tire(t) ∧ At(t, Ground) ∧ ¬At(Flat, Axle) ∧ ¬At(Spare, Axle)
    EFFECT: ¬At(t, Ground) ∧ At(t, Axle))

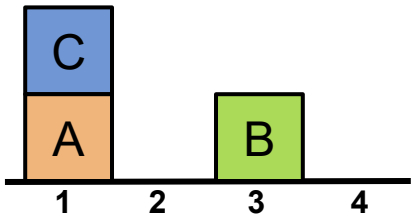
**Action**(LeaveOvernight,
    PRECOND:
    EFFECT: ¬At(Spare, Ground) ∧ ¬At(Spare, Axle) ∧ ¬At(Spare,Trunk)
    ∧ ¬At(Flat, Ground) ∧ ¬At(Flat, Axle) ∧ ¬At(Flat, Trunk))

- Možna rešitev?
- Kako se akcije odražajo na zaporednih spremembah stanja?

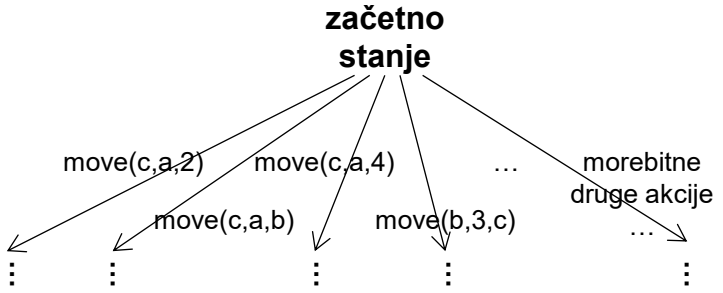
# Preiskovanje prostora stanj

- uporaba neinformiranih, informiranih ali lokalnih preiskovalnih algoritmov
- iskanje v smeri od začetnega k ciljnemu stanju lahko razvija vozlišča z uporabo akcij, ki niso relevantne
- kombinatorična eksplozija prostora stanj

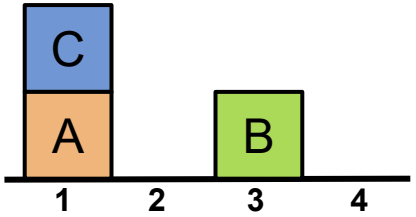
- primeri:
  - možni premiki kock iz začetnega stanja
  - akcija: `buy(Isbn)`
    začetno stanje: `[]`,
    predpogoji: `[]`,
    učinki: `add [own(Isbn)]`,
    cilj: `[own(1234567890)]`




- rešitve:
  - dobra hevristična ocena
  - drugačen pristop k preiskovanju

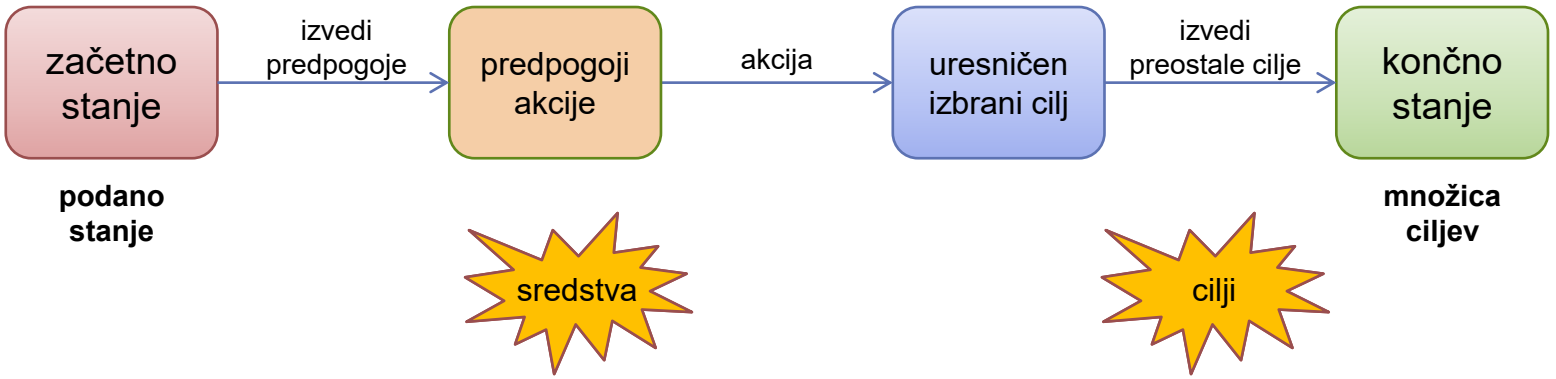


# Planiranje s sredstvi in cilji

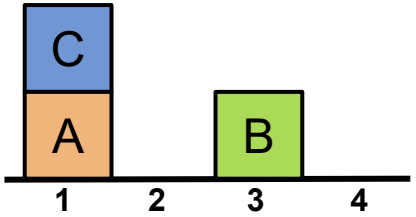



- primer iz sveta kock
  stanje: `[on(c,a), on(a,1), on(b,3), clear(c), clear(2), clear(b), clear(4)]`
  cilj: `[on(a,b),on(b,c)]`

- način reševanja:
  1. izberi **nerešen cilj**
  2. izberi **akcijo**, ki lahko vzpostavi ta cilj
  3. ker ima akcija **predpogoje**, omogoči akcijo z izvedbo predpogojev
  4. **izvedi akcijo**
  5. vrni se v korak 1



# Planiranje s sredstvi in cilji




- primer iz sveta kock

  stanje: `[on(c,a), on(a,1), on(b,3), clear(c), clear(2), clear(b), clear(4)]`

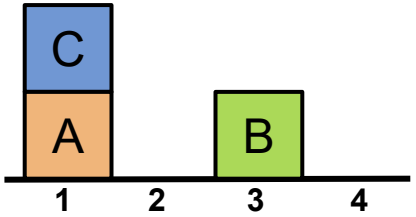
  cilj: `[on(a,b),on(b,c)]`

- rešitev plana pri preiskovanju v globino:

```
move(c,a,2)   % izpolnjujemo on(a,b), ki potrebuje clear(a)
move(a,1,b)   % izpolnimo cilj on(a,b)
move(a,b,1)   % izpolnjujemo on(b,c), ki potrebuje clear(b), pokvarimo on(a,b)
move(b,3,c)   % izpolnimo cilj on(b,c)
move(a,1,b)   % ponovno izpolnimo cilj on(a,b)
<plan zaključen, vsi cilji izpolnjeni>
```

- pomembne podrobnosti:
  - strategija preiskovanja (v globino, širino, iterativno poglabljanje)
    - ali bi iterativno poglabljanje in iskanje v širino našla najkrajši plan?
  - princip varovanja (ščitenja) ciljev (angl. *goal protection*): pri preiskovanju lahko dodatno varujemo, da ne podremo že doseženih ciljev

# Planiranje s sredstvi in cilji




- primer iz sveta kock
  stanje: `[on(c,a), on(a,1), on(b,3), clear(c), clear(2), clear(b), clear(4)]`
  cilj: `[on(a,b),on(b,c)]`

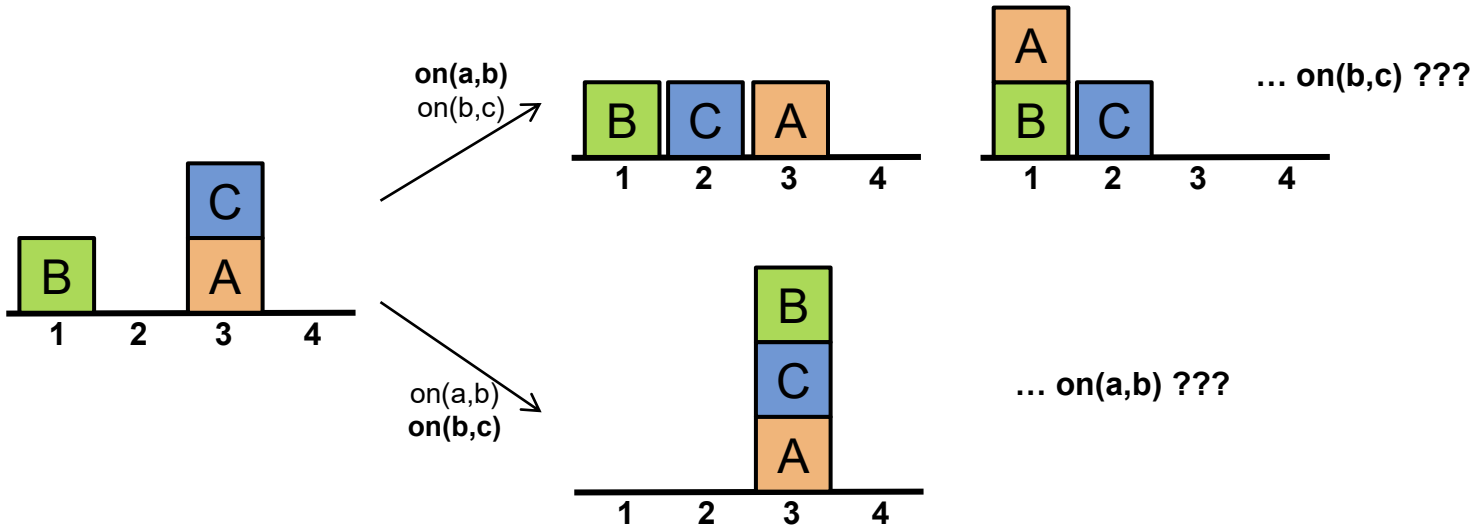
- pri iskanju v širino / iterativnem poglabljanju dobimo naslednjo rešitev:

```
move(c,a,2)   % doseže clear(a) za move(b,3,a)
move(b,3,a)   % doseže on(b,a) za move(b,a,c)      idealno bi bilo move(b,3,c) ?
move(b,a,c)   % doseže clear(a) za move(a,1,b) / doseže on(b,c)
move(a,1,b)   % doseže on(a,b), vsi cilji izpolnjeni
```

- najkrajši plan ni (vsebinsko) optimalen?
- zakaj?

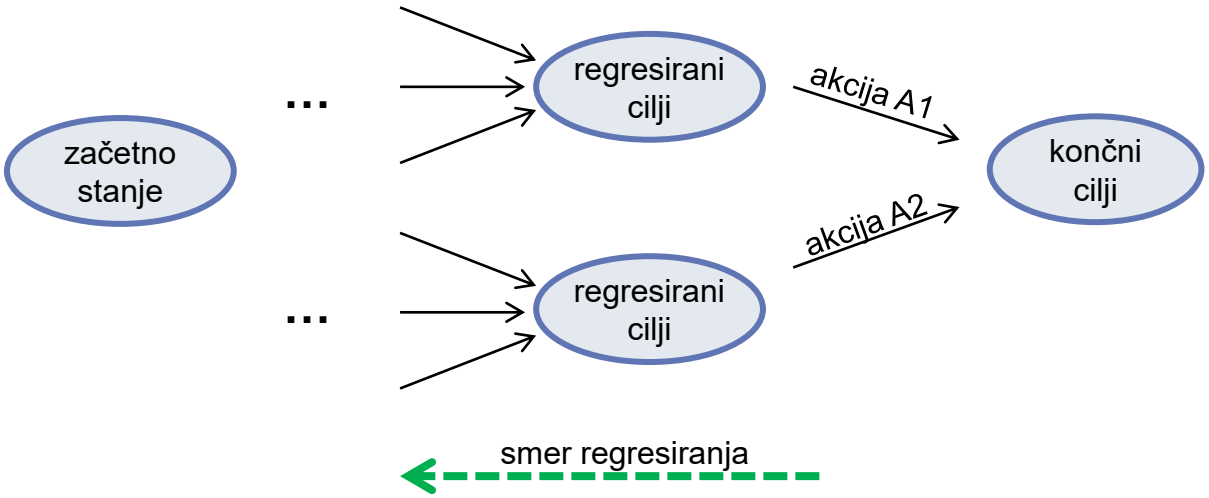
# Sussmanova anomalija

- Sussman anomaly (Gerald Sussman, 1975)
- je problem interakcije med cilji
  - algoritem za planiranje (STRIPS) obravnava cilje "lokalno" (enega po enega, ne ozirajoč se na drugega med reševanjem prvega)
  - z doseganjem enega cilja algoritem razveljavi že dosežene cilje ali predpogoje za njihovo doseganje
  - planiranje poteka linearno (najprej prvi cilj, šele nato naslednji, …)
  - vrstni red obravnavanja ciljev vpliva tudi na nepotrebne korake pri planiranju
- rešitve
  - drugačen algoritem za planiranje (regresiranje ciljev)
  - ne vztrajaj na urejenosti ciljev, če to ni nujno potrebno (nelinearno planiranje)



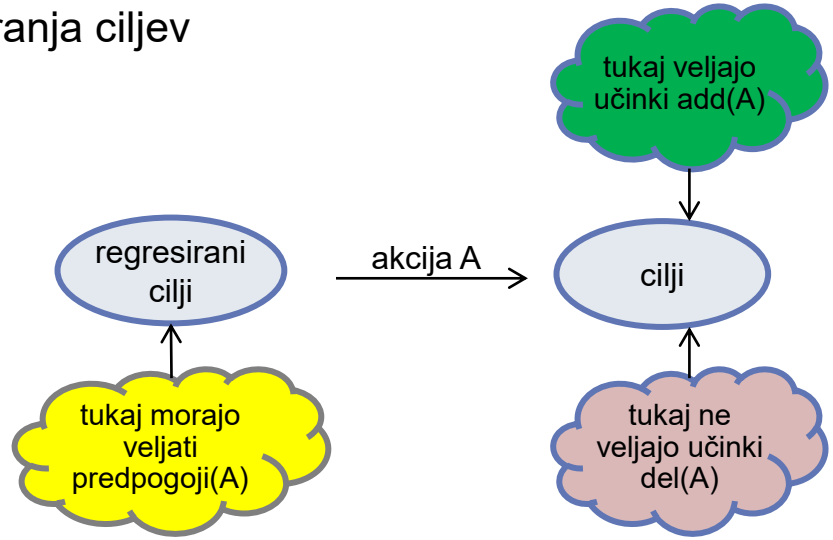
# Planiranje z regresiranjem ciljev

- rešitev za Sussmanovo anomalijo
- vzvratno preiskovanje od cilja proti začetnemu stanju (angl. *goal regression through action*)
- drugačna filozofija:
  - *globalno* planiranje, ker algoritem za planiranje obravnava vse cilje hkrati
  - ne obravnavamo samo akcij, ki so *možne*, temveč *najbolj smiselne*
- postopek:
  - izberemo akcijo, ki doseže izbrani cilj (in čim več preostalih ciljev)
  - izračunamo "predhodne" cilje ob uporabi te akcije (= regresiranje ciljev skozi akcijo)
  - nadaljujemo z regresiranjem, dokler ne pridemo do ciljev, ki so izpolnjeni v začetnem stanju



# Planiranje z regresiranjem ciljev

- postopek regresiranja ciljev

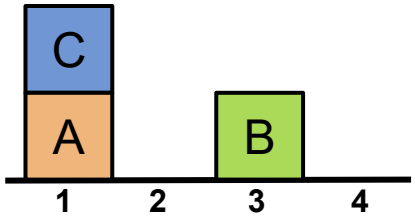



- $\text{regresirani cilji} = \text{cilji} \cup \text{predpogoji}(A) - \text{add}(A)$
- veljati mora $\text{cilji} \cap \text{del}(A) = \emptyset$

- "stanja" pri preiskovanju so **množice ciljev**
- **ciljni pogoj:** $\underline{\text{regresirani cilji} \subseteq \text{cilji}}$ v začetnem stanju
- uporabimo znane preiskovalne algoritme (neinformirani / informirani algoritmi; A*, hevristika?)

# Planiranje z regresiranjem ciljev

- v primeru iz sveta kock velja:
  stanje: `[on(c,a), on(a,1), on(b,3), clear(c), clear(2), clear(b), clear(4)]`
  cilj: `[on(a,b),on(b,c)]`




- rešitev z regresiranjem ciljev najde optimalno rešitev (rešitev na vajah)

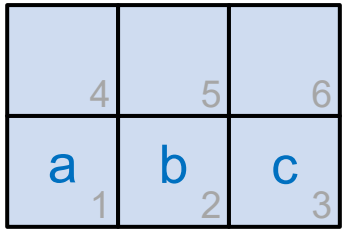
```
move(c,a,2)
move(b,3,c)
move(a,1,b)   %plan zaključen, vsi cilji izpolnjeni
```

- vaja regresiranja:
  Regresiraj cilja $C = \{on(a,b), on(b,c)\}$ skozi akcijo `move(a,2,b).`

$$Regresirani\ cilji\ = C \cup predpogoji(A) - add(A)$$
$$= \{on(a,b), on(b,c)\} \cup \{clear(a), clear(b), on(a,2)\} - \{on(a,b), clear(2)\} =$$
$$= \{on(b,c), clear(a), clear(b), on(a,2)\}$$

Pogoj: $\{on(a,b), on(b,c)\} \cap \{on(a,2), clear(b)\} = \emptyset$

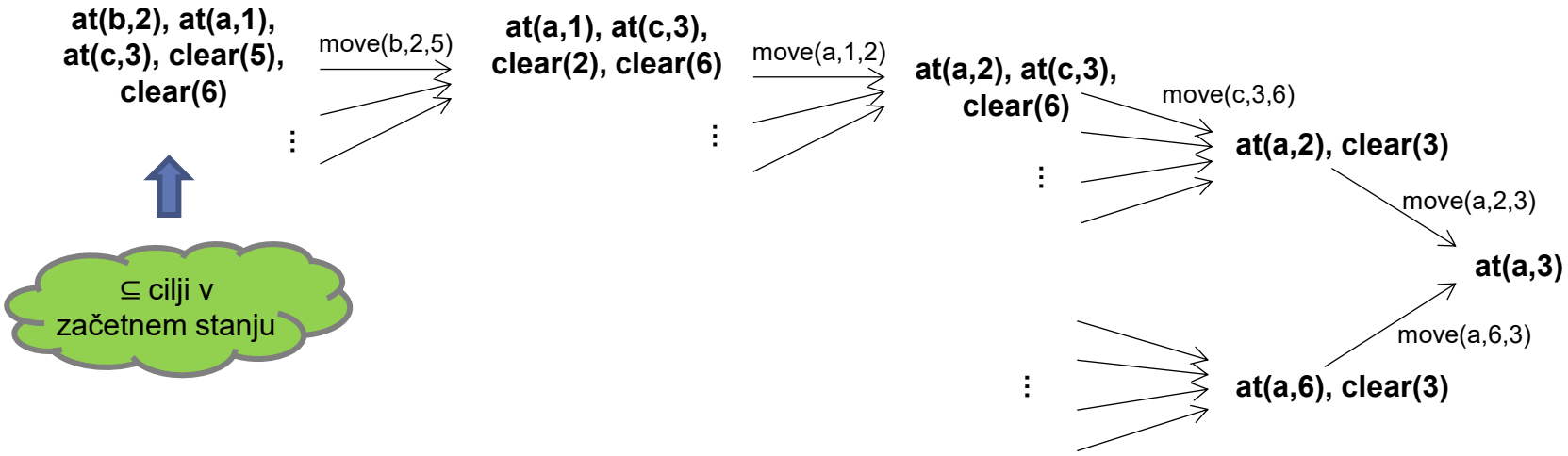
# Planiranje z regresiranjem ciljev




- primer: roboti na pravokotni mreži
- začetno stanje: `[at(a,1), at(b,2), at(c,3), clear(4), clear(5), clear(6)]`
- ciljno stanje: `[at(a,3)]`
- akcija:

| | `move(Robot,From,To)` |
|---|---|
| predpogoj: | `[at(Robot,From), clear(To)]` |
| implicitne omejitve: | `[robot(Robot), adjacent(From,To)]` |
| add: | `[at(R,To), clear(From)]` |
| del: | `[at(Robot,From),clear(To)]` |

---




- plan: `move(b,2,5), move(a,1,2), move(c,3,6), move(a,2,3)`


Razporejanje opravil