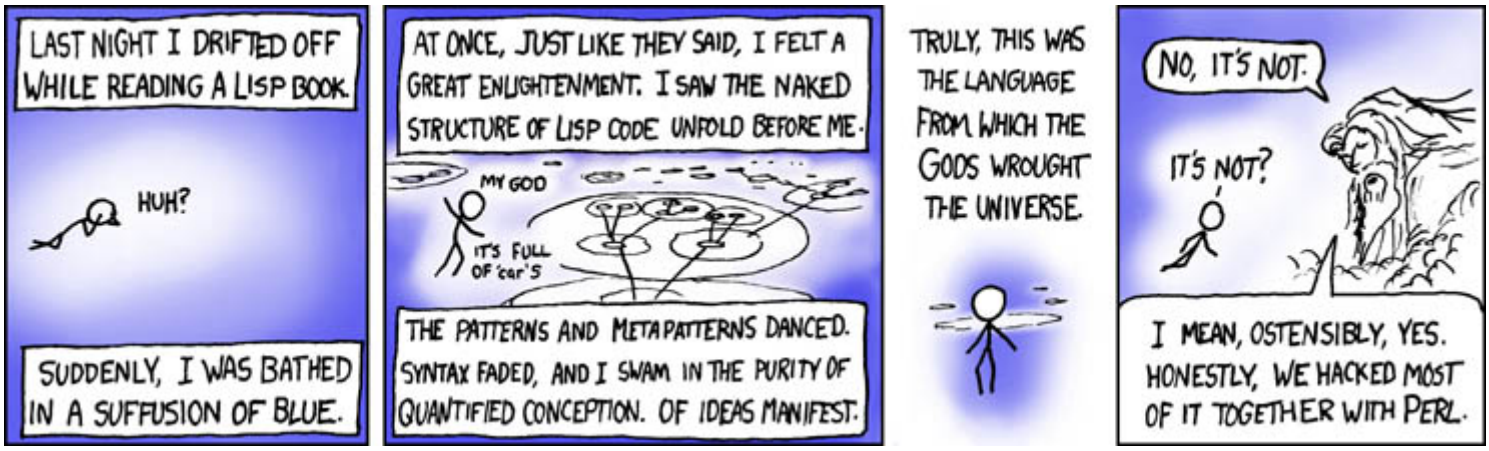
Some Advanced LISP Topics

Now that you've done HW1, are all experts in LISP, and can see its beauty firsthand, you'll appreciate the following comic:



(credit to XKCD, of course)

(https://xkcd.com/224/)
Since you're enlightened by the LISP way, let us dive further into its beauty and see some convenience functions as well as some nice functional programming paradigms.

(member item list)

⚙ The **member** function is used to determine if parameter item is located anywhere in parameter list (analogous to set-membership).

There are a couple of peculiarities with this function:

- If parameter item is indeed located within parameter list, then the *tail* of the list starting with matched element item is returned.
- Asking if item NIL is located in the list is not the set-theoretical "is the empty set a member of every set" but is instead asking if NIL is an element of list!

Example
☑ What will the following statements return?

```
;; #1
(member 1 '(1 2 3))

;; #2
(member 2 '(1 2 3))

;; #3
(member NIL '(1 2 3))

;; #4
(member NIL '(1 NIL 3))
```

Some functions offer optional named parameters.

ℹ A **named parameter** is a parameter that is passed not by position in the argument list, but by name using the :name syntax.

Furthermore, we can pass function names as arguments using the special syntax: `#'fun-name`

ℹ A **function reference** is a way to access the function definition associated with a symbol, using the syntax: `#'fun-name`

So, say we wanted our member function to be able to find lists instead of just atoms within the input list, observe:

```
;; #1
(member '(1 2) '((3 4) (1 2 3) (1 2) (4 5) ()))

;; #2
(member '(1 2) '((3 4) (1 2 3) (1 2) (4 5) ()) :test #'equal)
```

The first call to member defaults by treating the input item '(1 2) as an atom, and so we actually don't find a match!

So, instead of using member's default test, we can instead define our own test using the equal function that we know and love!

The second statement says, "See if our item is a member of the given list *using the test comparator of the equal function*"

ℹ HINT! You can use the :test parameter in your unit tests to compare membership of frames (using #'equal-sf) and membership of lexicon entries (using #'equal).

So, what's the main difference between a symbol and a function reference?

We'll see how to evaluate these function references using different forms later.

(subsetp list1 list2)

The **subsetp** predicate returns a boolean value for whether or not list1 is a subset of list2.

So, we can determine if all elements of list1 appear, somewhere in sequence, in list2

Example
What will the following statements return?

```
;; #1
(subsetp '(1) '(3 2 1 2 3))

;; #2
(subsetp '(1 2) '(3 2 1 2 3))

;; #3
(subsetp '(2 2) '(3 2 1 2 3))

;; #4
(subsetp '() '(3 2 1 2 3))
```

Other Set Operations

We can perform the other standard set operations, treating lists as sets, including: **intersection, union, and set-difference**

Example
What will the following statements return?

```
;; #1
(union '(1 2 3) '(2 3 4))

;; #2
(intersection '(Veteran Wilshire) '(Wilshire Sepulveda))

;; #3
;; Returns (list1 - list2)
(set-difference '(1 2 3) '(2 3 4))
```

(subseq sequence start &optional end)

☼ The **subseq**(uence) function returns a list consisting of parameter sequence starting with element index start and (if provided) ending with element index end.

Example
☑ What will the following statements return?

```
;; #1
(subseq '(DONT REMOVE ME PLEASE too bad) 4)

;; #2
(subseq '(DONT REMOVE ME PLEASE too bad) 1 3)
```

(if (cond) true-exp false-exp)

Want to have a conditional with only two cases? Why bother using cond when you can use if!

☼ The **if** form evaluates a single condition and then executes a single true expression if the conditional evalutes to true, or the false expression otherwise.

Example
☑ What will the following statements return?

```
;; #1
(setq L2 'L1)
(if (equal 'L1 L2) 'WOW-TRUE 'AWW-FALSE)

;; #2
(if (equal 'L1 'L2) 'WOW-TRUE 'AWW-FALSE)
```

Now, what if we wanted to execute multiple expressions in the true or false slot?

(progn exp1 exp2 ...)

Want to execute multiple expressions where a single one is expected?

⚙ The **progn** form evaluates each of its expressions as though they were top-level expressions and returns the last one evaluated.

Example
☑ What will the following statements return?

```
;; #1
(setq L2 'L1)
(if (equal 'L1 L2) (progn (setq A 3) (* A 3)) (progn (setq A 2) (- A 3)))

;; #2
(if (equal 'L1 'L2) (progn (setq A 3) (* 2 3)) (progn (setq A 2) (- 2 3)))
```

(loop for item in list do (expr1) (expr2))

Thirsty for the old means of iteration? Well, LISP offers that too (reluctantly).

⚙ The **loop** form repeatedly sets variable item to the next element in list starting with the first, and executes everything to the right of do with item = (first list).

There are some peculiarities with looping:

- Loops will return nil for the entire loop by default
- If you want to return something from a loop, you must explicitly use the (return expr) function, which terminates the loop

Example
☑ What will the following statements print / return?

```
;; #1
(loop for v in '(1 2 3 4) do (if (evenp v) (print v) (print 'ODD)))

;; #2
(loop for v in '(1 2 3 4) do (if (evenp v) (return v) (print 'ODD)))
```

There are a *bunch* of variants on the loop syntax that each do slightly different things, three of which are listed below:

- **(loop for v on list do)**
  Same as with using "in" list loop format except each iteration sets v = (rest list)
- **(loop for v while (condition) do (expr))**
  Keep iterating with v = 0, 1, 2, ... until condition returns nil.
- **(loop for v from lower to upper [by step] do (expr))**
  Iterate through the range from v = lower up to v = upper with optional step size (default 1)

Example
☑ What will the following statements print / return?

```
;; #1
(loop for v on '(1 2 3 4) do (print v))

;; #2
(loop for v while (not (= 5 v)) do (print v))

;; #3
(loop for v from 2 to 8 by 2 do (print v))
; Who do we appreciate?
```

We can even collect the results of our individual loop iterations and compile them into a list!

The following two loop syntaxes are handy for this, where we use any form listed above but simply replace the "do" keyword with:

- **(loop for v ... collect (fun-name v))**
  Returns a list of returned values from each iteration with fun-name called on each v.
- **(loop for v ... append (fun-name v))**
  Similar to the above, but applies the append function to each element of the collection at the end.

Example
What will the following statements print / return?

```
;; #1
(loop for v in '(1 2 3 4) collect (evenp v))

;; #2
(loop for v in
    '((some sublists) (ARE TOO LONG) (TO MAKE THE CUT) (yay))
    append (if (< (length v) 3) v nil))
```

There are also a variety of shorthand loops that employ our function references again.

One of the more useful ones is "every"

(every #'fun-name '(arg1 arg2 ...))

The **every** form applies the function represented by the function reference to every element on its right, and returns true if all applications of #'fun-name also return true (nil otherwise).

Example
What will the following statements print / return?

```
;; #1
(every #'evenp '(2 4 6 8 22))

;; #2
(every #'print '(4 5 2))
```

Function Evaluation

We've already seen eval used to evaluate symbols and use data that can be treated as a function.

As it turns out, there are a couple of other function calling forms we can use depending on our syntactic needs, and what we know at runtime.

ℹ The **apply** form allows us to split our function call into two parts: the function symbol name, and then a list containing the arguments.

ℹ The **funcall** form allows us to split our function call into two parts: the function reference, and then the arguments, one after another.

The difference is that apply takes a function symbol name and the arguments in a list whereas funcall takes a function reference and then the arguments one after another.

So, we can represent our three methods for calling functions as follows:

- **(eval list)** where list is a function and its arguments.
- **(apply 'fun-symbol '(arg1 arg2 ...))** where fun-symbol is the symbol for a function definition.
- **(funcall #'fun-ref arg1 arg2 ...)** where #'fun-ref is the function reference and the arguments are listed right after, not necessarily in list form.

Example
☑ What will the following statements return?

```
(defun test (p1 p2 p3) (+ (* p1 p2) p3))

;; #1
(setq data-eval '(test 2 3 4))
(eval data-eval)

;; #2
(apply 'test '(2 3 4))

;; #3
(funcall #'test 2 3 4)
```

They all do the same thing!

So why use one over another? Well, sometimes our function names and arguments are scattered and these provide us with different ways of calling functions, e.g.:

- **Use eval** when our function name and arguments are all in one list ready to be executed.
- **Use apply** when our function name might need to be run on some list of arguments located elsewhere.
- **Use funcall** when we have a function reference and don't have a single list containing our arguments.

Use whichever one most naturally fits your task!

(set var val) vs (setq sym val)

So, what's the difference between set and setq?

⚙ The **set** form sets the symbol *represented by* its first argument to the value val.

⚙ The **setq** form sets the symbol *that IS the first argument* to the value val.

Example
☑ What will the following statements return?

```
;; #1
(setq test 'stuff)
test

;; #2
(setq stuff 'yo)
stuff

;; #3
(set stuff 'test)
stuff
yo

;; #4
(set yo 5)
yo
test
```

This is a hint for how to do some of the "demons" in HW2's problems!

Optional Parameters

We saw an optional parameter with subseq already, but let's define it a bit further.

⚙ An **optional parameter** to a function is one that can be omitted entirely in a function call and still function properly.

We use the `&optional` syntax to indicate, in a parameter list, that every parameter to the right is optional.

Example
☑ What will the following statements return?

```
(defun test (p1 p2 &optional p3)
    (let* ((result (* p1 p2)))
        (if p3 (+ result p3) result)
    )
)

;; #1
(test 2 3)

;; #2
(test 2 3 1)
```

ℹ You may also provide default values for optional parameters via the syntax: `(defun (p1 &optional (def-param 1)) ...)` (where def-param is a default parameters that has value 1 if it is not provided, or else the value of def-param as passed in.

ℹ NB. Feel free to augment any of the HW2 function signatures with optional parameters SO LONG AS THE ORIGINAL PARAMETERS ARE STILL PRESENT AND IN THE SAME ORDER.

For example, for FIND-CON:

```
;; Original:
(FIND-CON mycon dir class)

;; The following is fine:
(FIND-CON mycon dir class &optional opparam)
```

Your recursive calls might be cleaner given some optional parameters, but again, it's just a tool that you can use where you find it convenient!

Homework 2 In a Nutshell

So homework 2... what would you say... ya do here?

Homework 2 is attempting to perform the task of taking a list of words representing a sentence, and turn them into a giant frame!

To do this, we use a few *cringe* global variables.

> ℹ A **lexicon** is a database of frames and "semantic expectations" indexed by words and phrases.

We'll use our lexicon to build up the relevant definitions that can be found from words and phrases in our sentences.

Within a sentence, however, we need a way of resolving references. For example, using one of Dyer's examples:

"John bludgeoned Mark with the sledgehammer of doom. He felt no regret."

Who is the "he" mentioned in the second sentence? John, presumably, but our system will need a means of disambiguating.

> ℹ The **working memory** is a list of frames (their representative atoms) that are used in order to resolve linguistic references and expectations.

Furthermore, we need a means of empowering our reasoning system with some knowledge of classifications.

It may be useful, for example, to know that a DOG is a MAMMAL, a HUMAN is a MAMMAL, but no HUMAN is a DOG.

> ℹ A **class hierarchy** provides some classifications of subtypes and supertypes, and are defined recursively in ISA relationships.

So, let's take a look at the various functions now and discuss any questions thusfar (consult your actual homework pdf if you're reading from home).

Function-specific Hints

Function	Hints
ADD-TO-LM	Use a helper to check for duplicate phrases. Just make sure you modify the global variable with the proper order of inputs.
LOOKUP-PHRASE	Non-trivial, you might consider the following: • Consider making a helper function that determines the length of a match between the front of the input sentence and an entry in the input lexicon. • You'll only ever have to compare two matches at a time: the current best that you've found and then a contender. Replace the current best if the length is greater in the contender.

Function	Hints
NEWGAPS	Big hint: the format of this one will look very similar to that of HW1's EXPAND.
FIND-CON	One of the least trivial; here are some hints: • Consider making a helper function that breaks the search space down into only the valid locations for a match to be within. For example, if your dir = AFT, only consider the sub-list from the matched starting atom to the end of the atmlst. • Using a particular list manipulator, consider how we might turn direction BEF into its AFT counterpart, for simplicity. • You might consider using an optional parameter to help you solve this problem!
IS-SUBCLASS	Remember to check for class hierarchies recursively! If A is a member of B and B is a member of C, then A is a member of C!
GEN-DEMS	Remember to modify the *DM global with completed demon instances!
DEM-EXEC	Not particularly long but the pieces will have to fit together properly using some of the new tricks about calling functions you found today: • Consider the proper function call form that you'll use to execute each demons in the *DM. • This function might be another candidate for an optional parameter! • You may find an iteration technique useful for executing each demon instance.
PARSE-SENT	Once all of the other functions are completed, PARSE-SENT will simply tie everything together. It probably won't be long, but you might find some loop mechanics useful...

Adversarial Search

Adversarial Search is a game-theoretical search strategy where we have one agent competing against another in order to maximize utility.

What a perfectly academic and boring definition.

Let's start with some definitions...

> ℹ An **agent** can be a human, a computer, or some sort of interacting entity within our search problem.

When we're talking about adversarial search, usually we have two or more agents that are competitors on opposing sides of a game or conflict.

So, for example, if a human was playing tic-tac-toe against a computer, we would have two agents: the human player and the computer player, each trying to win.

In the case of a game like tic-tac-toe, each agent is attempting to "maximize" their utility, meaning they have the goal of winning and want to make moves that increase their odds of doing so.

> ℹ **Utility** is therefore the score of a particular game outcome / conclusion state within our adversarial search space.

Hence, each agent will want to maximize its utility at the game's conclusion and make moves that take them to a state of maximal utility.

So, let's use tic-tac-toe as our motivating example for this section.

The game consists of the following attributes:

- Two players (agents) each attempting to win the game
- A board with 9 tiles
- A victory condition of one player connecting 3 of their symbols in a row horizontally, vertically, or diagonally.
- A tie condition of all 9 tiles being filled before either player reached a victory condition
- A turn where each player places one of their symbols on a vacant board slot

So, if one of our players is an intelligent agent that we're programming, then we can model the game (i.e., attempting to win it) as an adversarial search problem!

> ❓ What will a state look like in our tic-tac-toe search problem?

> ❓ A terminal state is one in which the game ends. What will terminal states look like in our problem?

Will a terminal state always have all squares filled in?

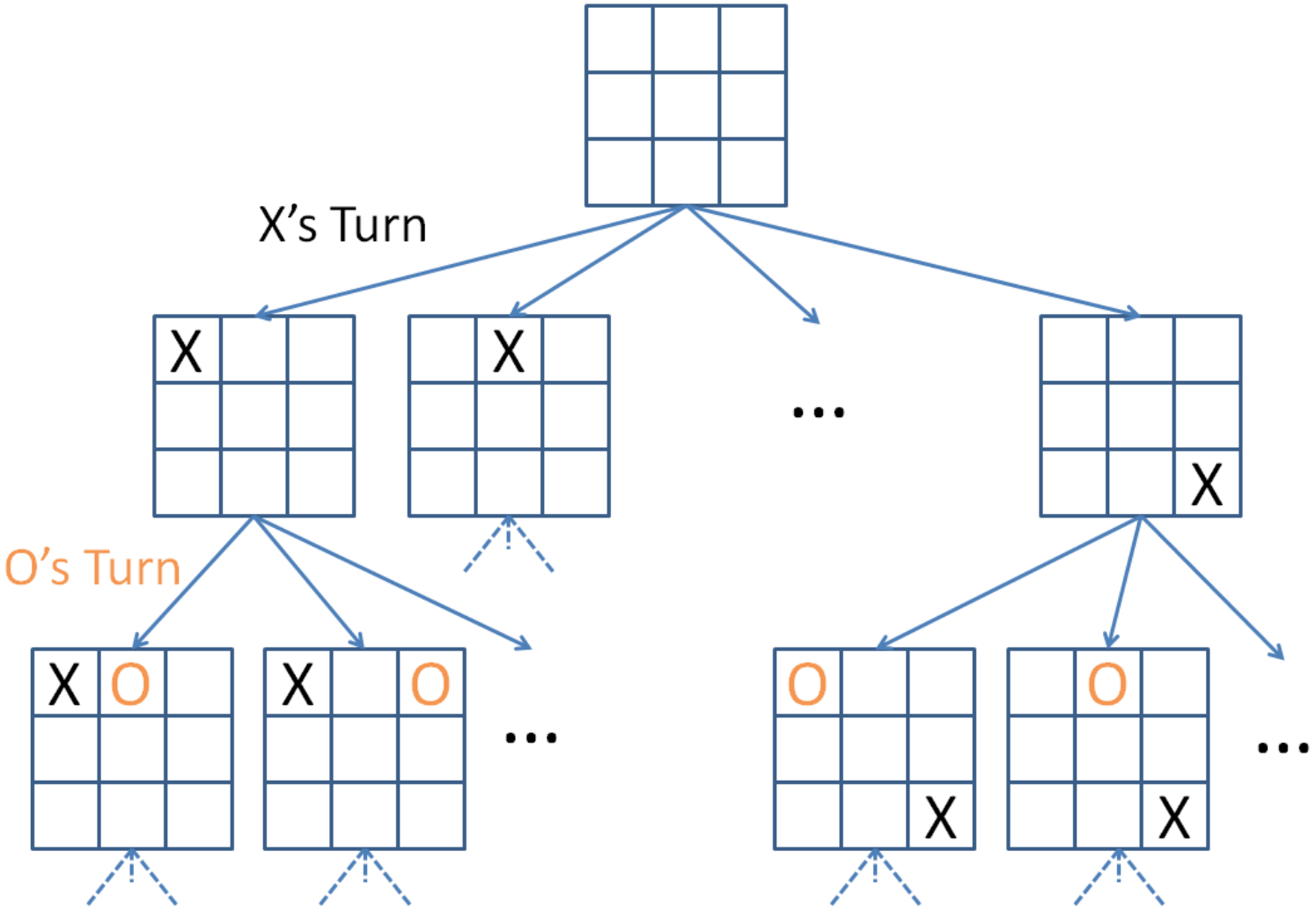
What will an action look like in our search?

Now that we have the components of our search defined, the real difference between search and adversarial search will be in the format of the search tree.

The **game tree** is the theoretical complete instantiation of all states and the actions that were used to move between them, starting with the initial state at the root.

The game tree for tic-tac-toe, then, consists of all of the possible states starting with the initial, blank board and leading to all terminal states at the leaves.

So, part of our game tree might look like the following:



Now, the search tree that we'll ultimately use to explore possible moves will be some subset of nodes from the game tree because as the games progresses, we will end up trimming branches based upon the move that our opponent makes.

Like we said, the objective of our intelligent agent will be to maximize its utility; we can therefore evaluate the utility of our terminal states using a utility function, U(s, p).

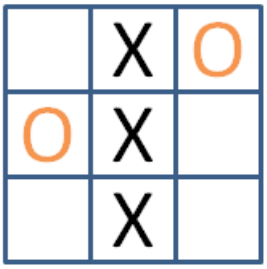
> ℹ A **utility function** U(s, p) is a function that scores a terminal state s based on how favorable it is to player p.

So, for example, in tic-tac-toe, we might say that U(s, p) = 1 when player p wins in state s, 1/2 when state s is a tie, and 0 when state s is a loss for player p.

This gives us some example terminal states:

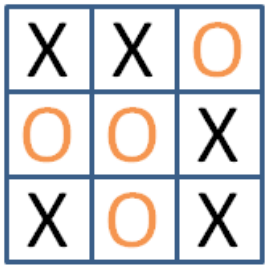
U(S1, X) = 1
U(S1, O) = 0

S1



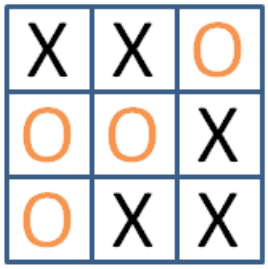
U(S2, X) = 1/2
U(S2, O) = 1/2

S2



U(S3, X) = 0
U(S3, O) = 1

S3



Thus, if we're X, then we want our agent to choose a path (given O's moves) that maximizes its chances of winning the game, or at the very least, tying it.

Our agent would then need some sort of strategy to search for the best chance of winning, which might mean using an evaluation function at every node along its frontier.

❓ If we had the full game tree at our disposal, what would be a good evaluation function for a node to maximize the agent's chances of winning?

Thus, if a node had a high evaluation function, that meant that there were many states along that path that could lead to a victory, and so, we would choose the move with the highest score.

❓ What are two problems with this strategy and evaluation function?

We'll address issue #1 first...

Minimax Search

Like we said, one of the issues with our traditional search is that it operates under the idea that the best solution can be found without the impedence of another agent.

With adversarial search, we have two or more agents fighting against one another.

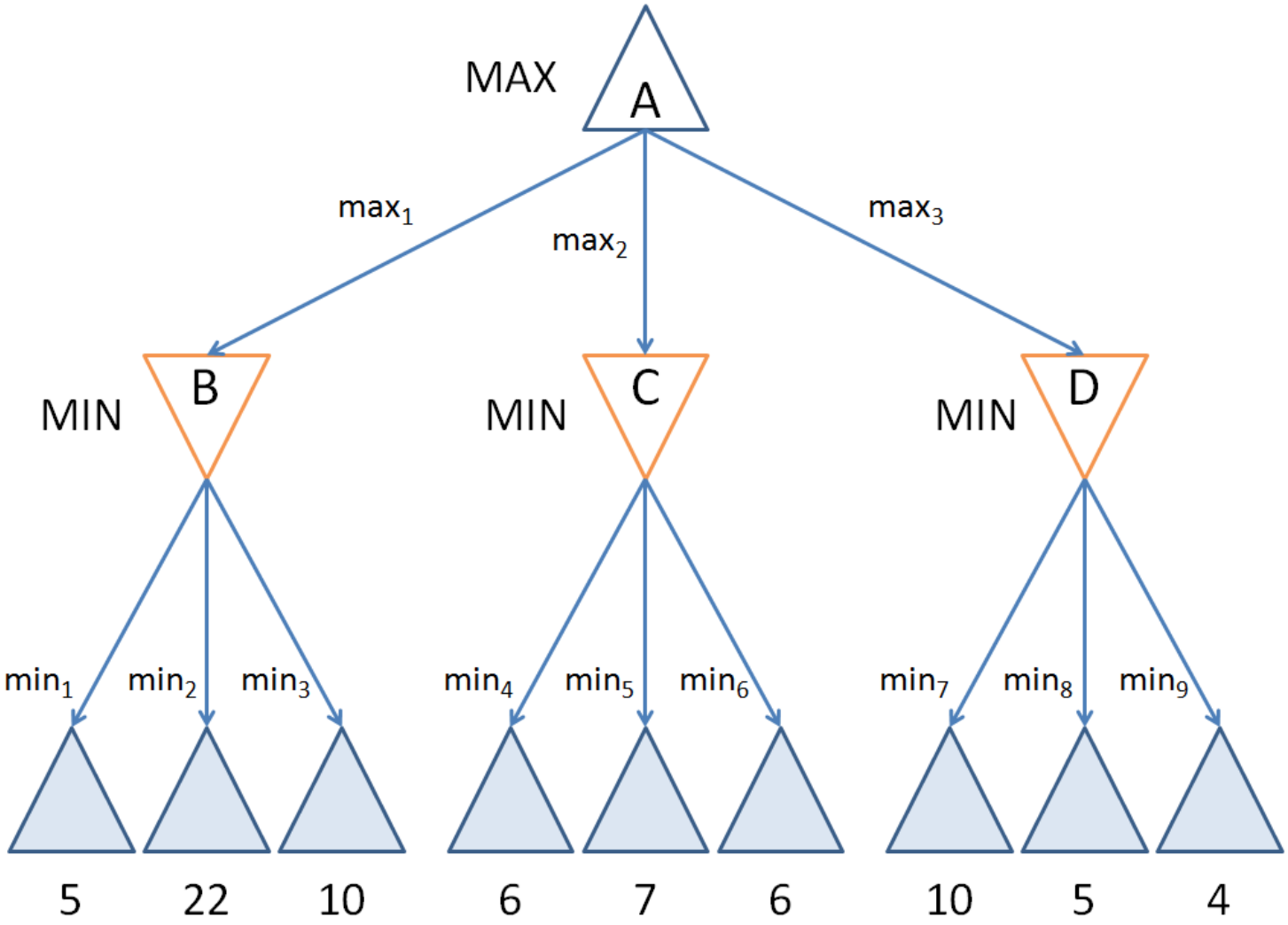
For the purposes of the search strategy we're about to discuss, let's consider two agents named MAX and MIN, and the following abstract problem specifications:

- At every move, either MAX or MIN can make one of three choices, starting with MAX and alternating.
- Terminal nodes have a utility function that returns a positive integer score.

So, in summary, for our arbitrary game, at each turn, MAX tries to maximize the utility it could possibly reach by pursuing a branch, and MIN tries to minimize the utility MAX could get by pursuing a branch.

We also make one key assumption about our agents: They are infallible, meaning that if a best move exists, they will make it (so we always prepare for the worst)

This means we can model our problem with a game tree that looks like the following (modified from book example):



❓ Assuming MAX started by making move max1, what move would MIN make from B?

❓ Assuming MAX started by making move max2, what move would MIN make from C?

❓ Assuming MAX started by making move max3, what move would MIN make from D?

❓ What, then, is MAX's best starting move?

So nothing particularly astounding about minimax search! Some stats about it:

- We perform depth-first search on the minimax tree, giving us both of the time and space complexities associated with depth-first with some max depth m [O(b^m) and O(b*m), respectively]
- Optimal assuming we have the game tree.
- Complete unless the problem specification allows for a potentially infinite loop.

⚠ However, naive minimax search may explore branches of the game tree that would never be considered in a realistic game against an optimally acting opponent.

Such wasteful explorations can hinder the efficiency of minimax, so next we'll see how we can cut down on those wasteful explorations.

Alpha-Beta Pruning (α-β Pruning)

ℹ **Alpha-Beta** pruning is a simple tactic used to reduce the number of nodes expanded by minimax search, and says: "I won't even bother exploring a subtree that I know will be worse than what I've already explored."

This is accomplished by placing bounds of what the min and max value could be at a given node based on what we've already explored.

If a new node we then later explore is less than the established min or greater than the established max of its parent node's range, then we don't bother exploring it.

We'll accomplish this by record-keeping values at each node:

ℹ The value of α at each node is the max score that the max player is guaranteed so far in the search; α is updated on MAX nodes only.

ℹ The value of β at each node is the min score that the min player is guaranteed so far in the search; β is updated on MIN nodes only.

ℹ In a nutshell, we explore the search tree using DFS and bubble up scores from leaves to their ancestors, updating α and β at max and min nodes respectively, and then cease to explore a subtree whenever β ≤ α.

❓ Based on the above definitions, why do we stop exploring a subtree whenever β ≤ α?

Wikipedia lists a nice pseudocode of the alpha-beta pruning algorithm, repeated here for your consumption. In particular, note the meaning of these variables:

```
function alphabeta(node, depth, α, β, maximizingPlayer)
  if depth = 0 or node is a terminal node
    return the heuristic score of node
  if maximizingPlayer
    v := -∞
    for each child of node
      v := max(v, alphabeta(child, depth - 1, α, β, FALSE))
      α := max(α, v)
      if β ≤ α
        break;
    return v
  else
    v := ∞
    for each child of node
      v := min(v, alphabeta(child, depth - 1, α, β, TRUE))
      β := min(β, v)
      if β ≤ α
        break;
    return v
```

Additionally, assuming we're starting with the maximizing player's turn, we would start the ball rolling with the call:

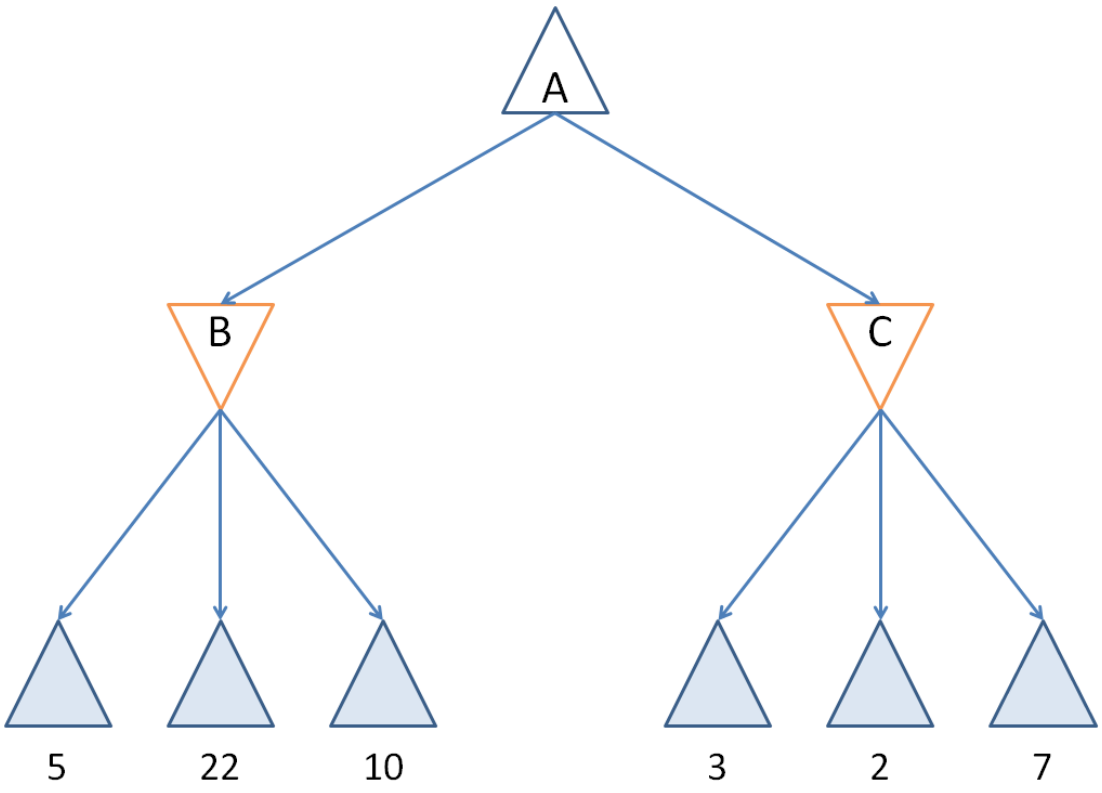
```
alphabeta(root, depth, -∞, ∞, TRUE)
```

So, we see the algorithm above, but what if we have to do this by hand on... say... an exam?

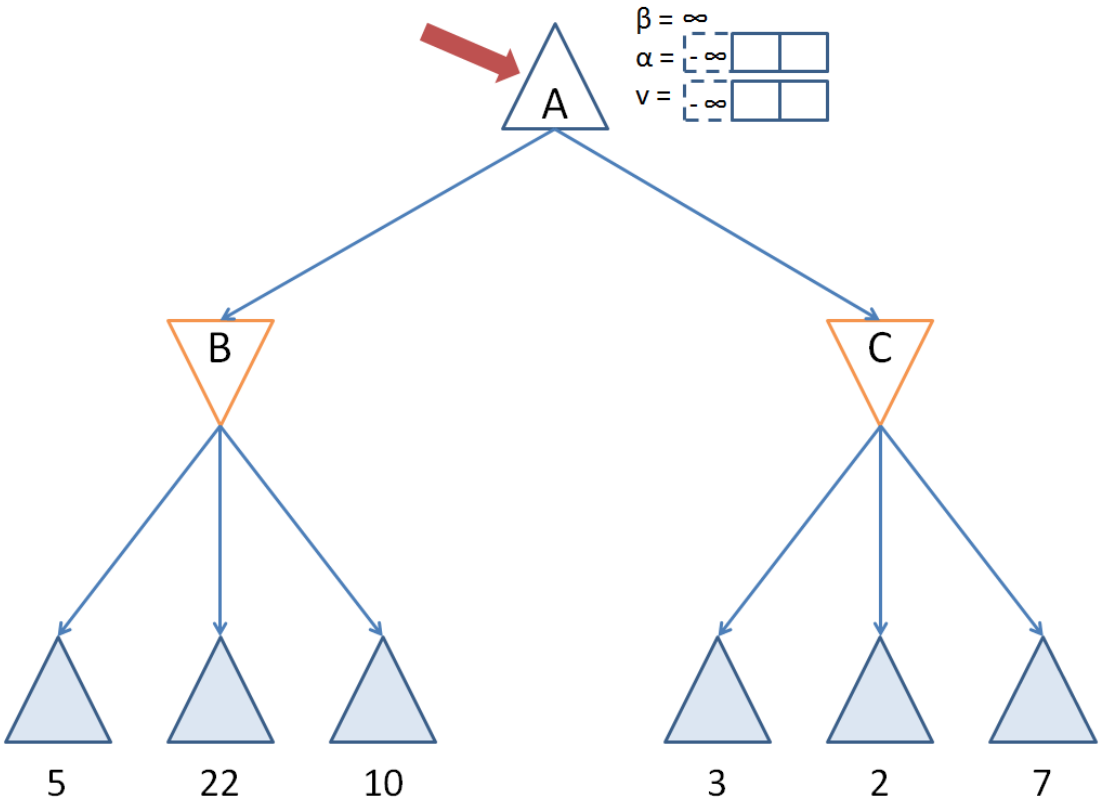
Fear not! There's a simple way to keep track of all of your stack frames.

1. Start at the root, and continue down the tree in a DFS order. Assume that at the root, $\alpha = -\infty$ and $\beta = \infty$
2. For each node we encounter, we inherit the current α and β values of its parent, and will record-keep several items:
    - If it's a MAX node, we know that β will not change, but α might, so we'll include a small list tracking any changes to α, and vice versa for MIN nodes.
    - We will also record-keep the value of V for each child of the current node, and update α (if it's a MAX node) or β (if it's a MIN node) for each child encountered.
    - If there are NO children (i.e., we've hit a leaf node), we simply pass our heuristic value back up to our parent.
3. If at any point we have $\beta \leq \alpha$, we stop exploring that node's children.
4. Continue DFS exploration until we've terminated all explorations by the above criteria, or have explored the entire tree.

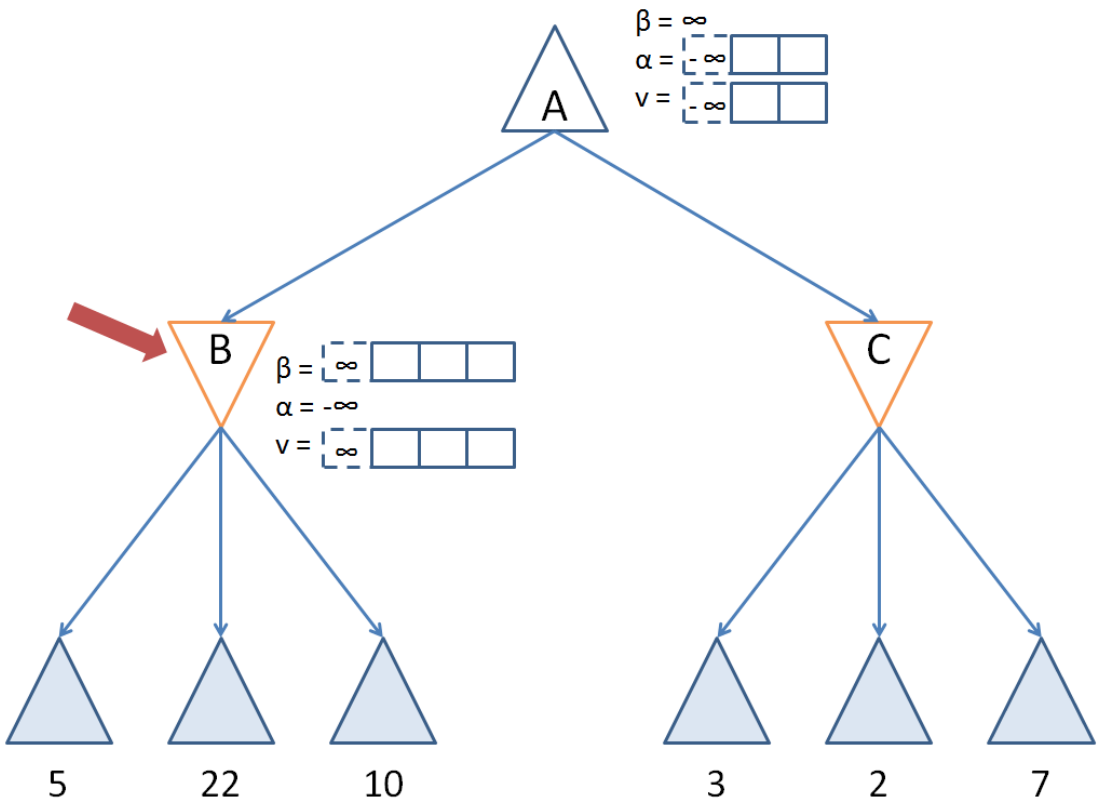
Let's see this in action on the following example:



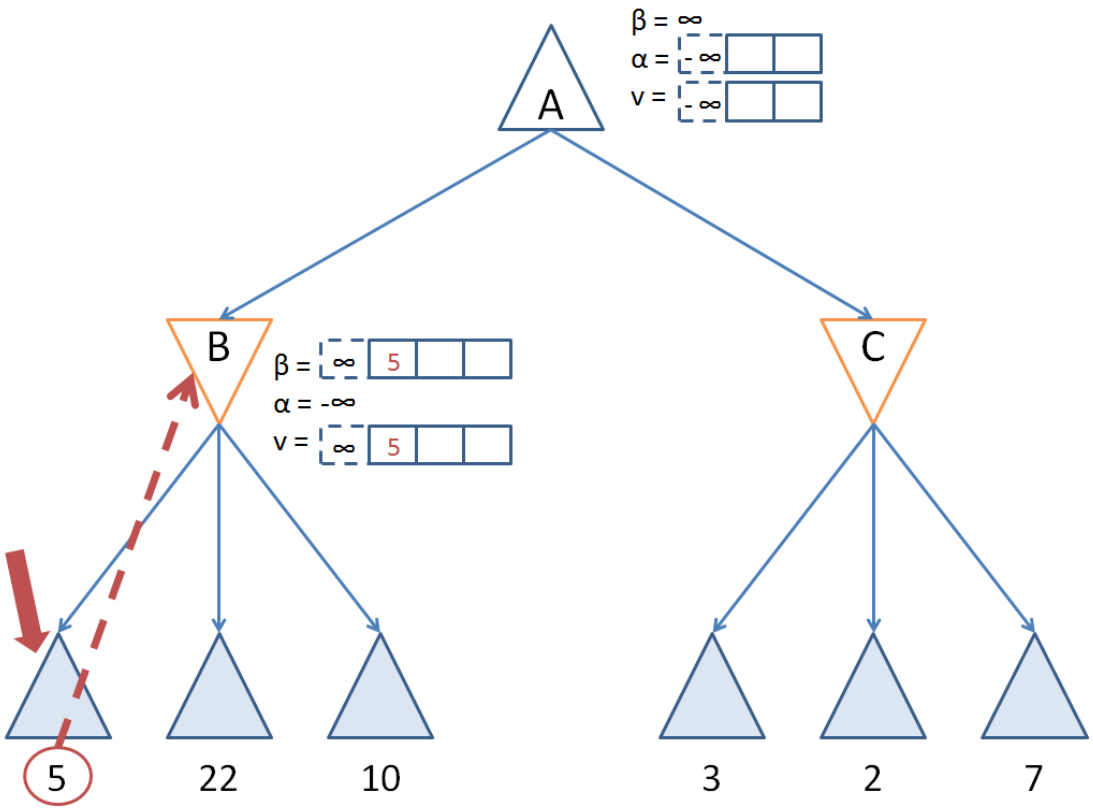
```
Starting at the root: We're at a MAX node, so:
  -> Record-keep changes to α (knowing that β won't change for this node),
     including room for change at each child of the node; we know that
     the algorithm starts α = -∞
  -> Record-keep changes to V at each child; since it's a MAX node,
     we know V starts out at -∞
```



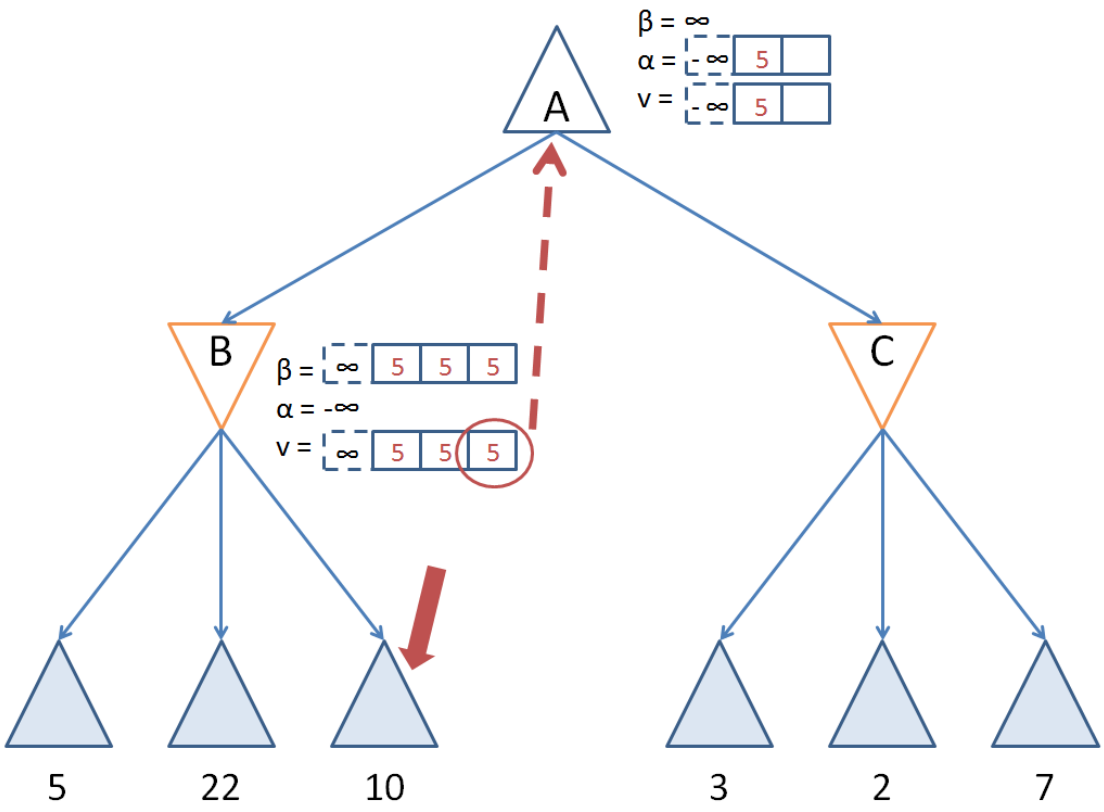

```
Visiting the root's first child, MIN node B:
  -> Since this is a MIN node, its value for α will be inherited from
     its parent, but β has the potential to change
  -> V starts out at ∞
```



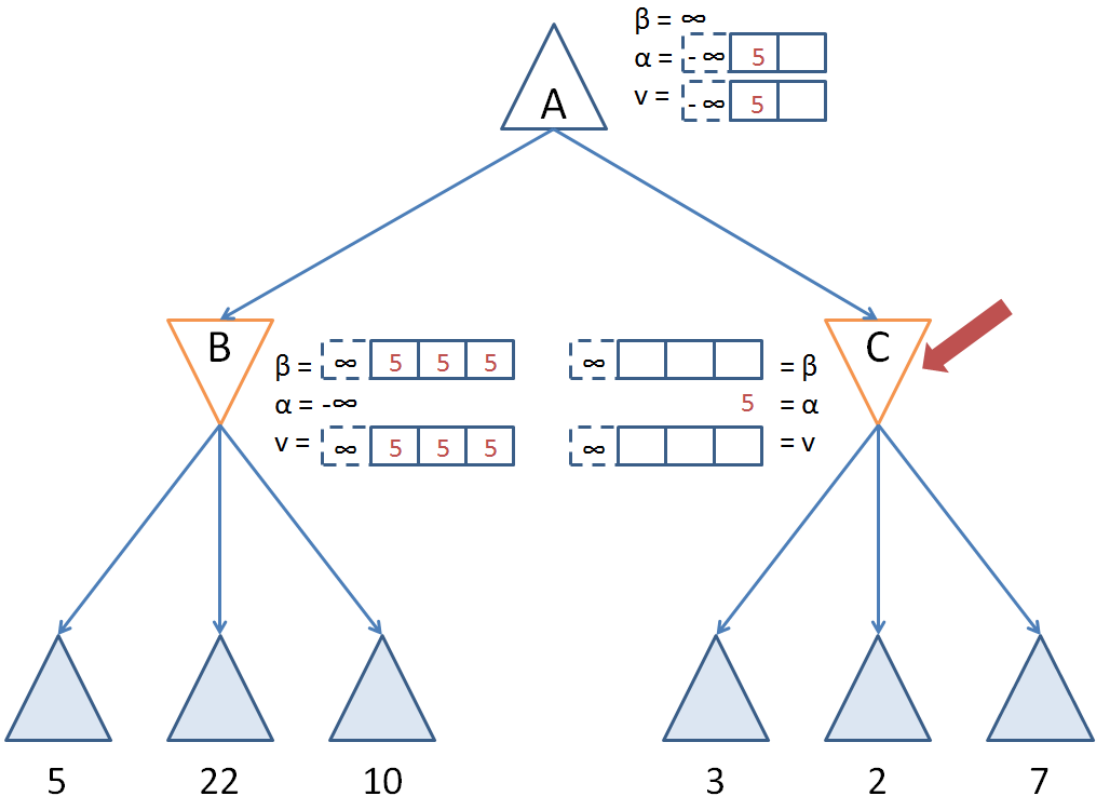

```
Visiting the first child of node B:
  -> This is a leaf, so we would return its heuristic value (5)
  -> Update the parent's current value for V to be:
       V = min(v, [child's value])
         = min(∞, 5)
         = 5
  -> Update the parent's β to be:
       β = min(β, v)
         = min(∞, 5)
         = 5
```



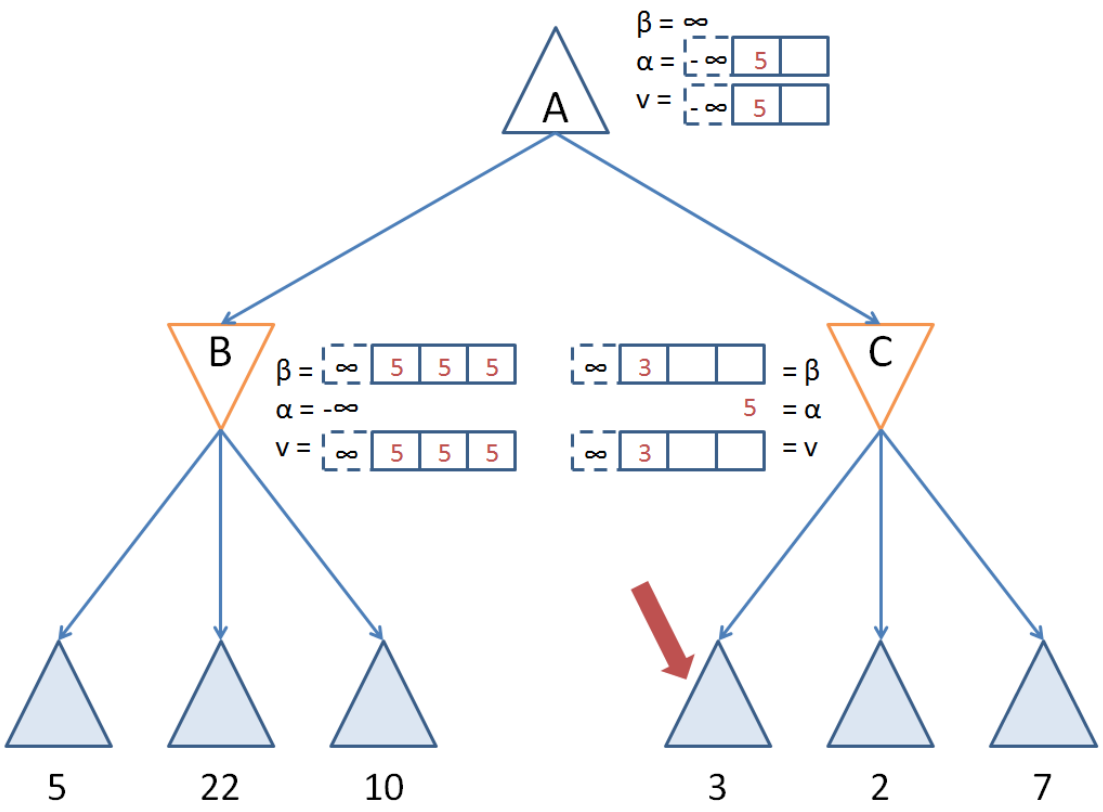

```
Continue visiting the children of node B:
  -> Notice that at all stages, α < β, so we do not terminate
  -> No child's value is less than the first child's, which was 5, so we
     end up returning V = 5 to the parent of B
  -> This means we update the values of V and α at node A accordingly
```




```
Visiting node C:
  -> Notice that the new value for α in node A gets sent down
     to this child
  -> Again, with this as a MIN node, we seed its V = ∞
```




```
Visiting first child of node C:
  -> Update the values of V and β = 3
  -> NOTE: Now we've reached a state where:
       β ≤ α
       3 ≤ 5
```




```
Terminate exploration of C's children
  -> Now we've exhausted the exploration and terminated,
     with no more nodes to explore, and having trimmed
     exploration of 2 of C's children
```