

Full Persistent Data-structure

Pointer Machine Model

PROJECT MEMBERS: ANANNYO DEY, SOUMYAJIT RUDRA SARMA, DEBASMIT ROY, KANKO GHOSH AND KUSHAL DAS

Full Persistent Linked List

Operations:

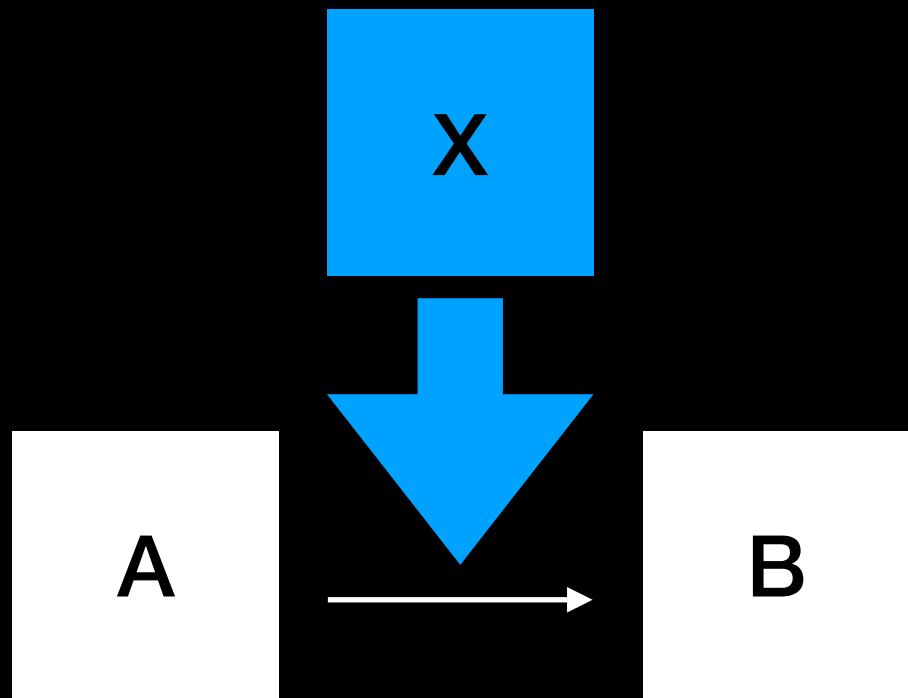
- `start = init()` : To initiate linked list and “start” pointer holds the starting position in `v0`
- `add(x, y, a, v)` : Add new node `x` after `y` at version `v` with `f1 = a` and `f2 = NULL`, and update the version in version tree.
- `create_node(x, a)`: Allocate a new node `x` with value = `a`, and set its default version = current time
- `remove(x,v)` : Remove node `x` and update the version at version `v` and update the version in version tree.
- `iterate_over_LL(v)` : Iterate over the whole linked list in version `v`
- `update(x,f_i,val,v)` : Update the `i`-th field in node `x` to new value ‘`val`’ at version `v` and update the version in version tree.

Interesting thing!

`add(x,y,v)` and `remove(x,v)` are not Elementary operations

`add(X,C,123)` consists of

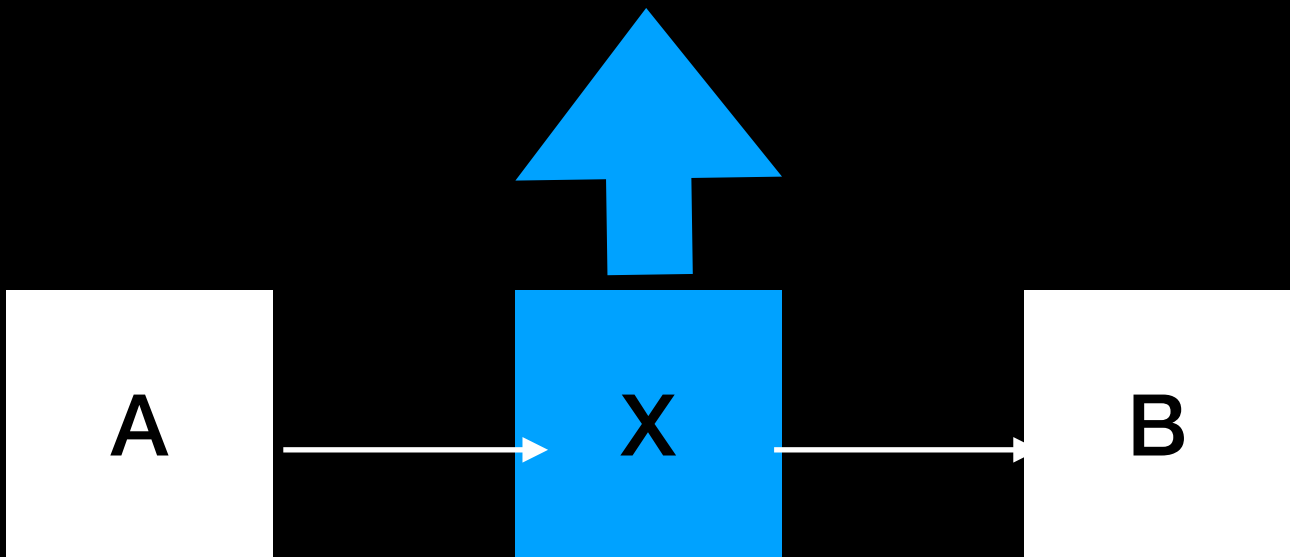
Create node X with value 123
Modify f2 of A to X : <code>update(f2, A, X,v)</code>
Modify f2 of X to C : <code>update(f2, X, C,v)</code>
Modify BP1 of X to A: <code>update(bp1, X, A,v)</code>
Modify BP1 of B to X: <code>update(bp1, B, X,v)</code>
Set f1 of X(optional)
Add current ver to the respective position in version tree



`remove(x)` consists of

Modify F2 of Parent C (i.e., X) -> F2 of C (successor of C after version v)

<code>update(f2, A, B,v)</code>
Modify BP1 of B to A: <code>update(bp1, B, A,v)</code>
If all shared reference of X is removed Then free up the memory associated with X



Elementary Operations:

- `start = init()`

- `create_node(x, a)`

- `iterate_over_LL(v)`

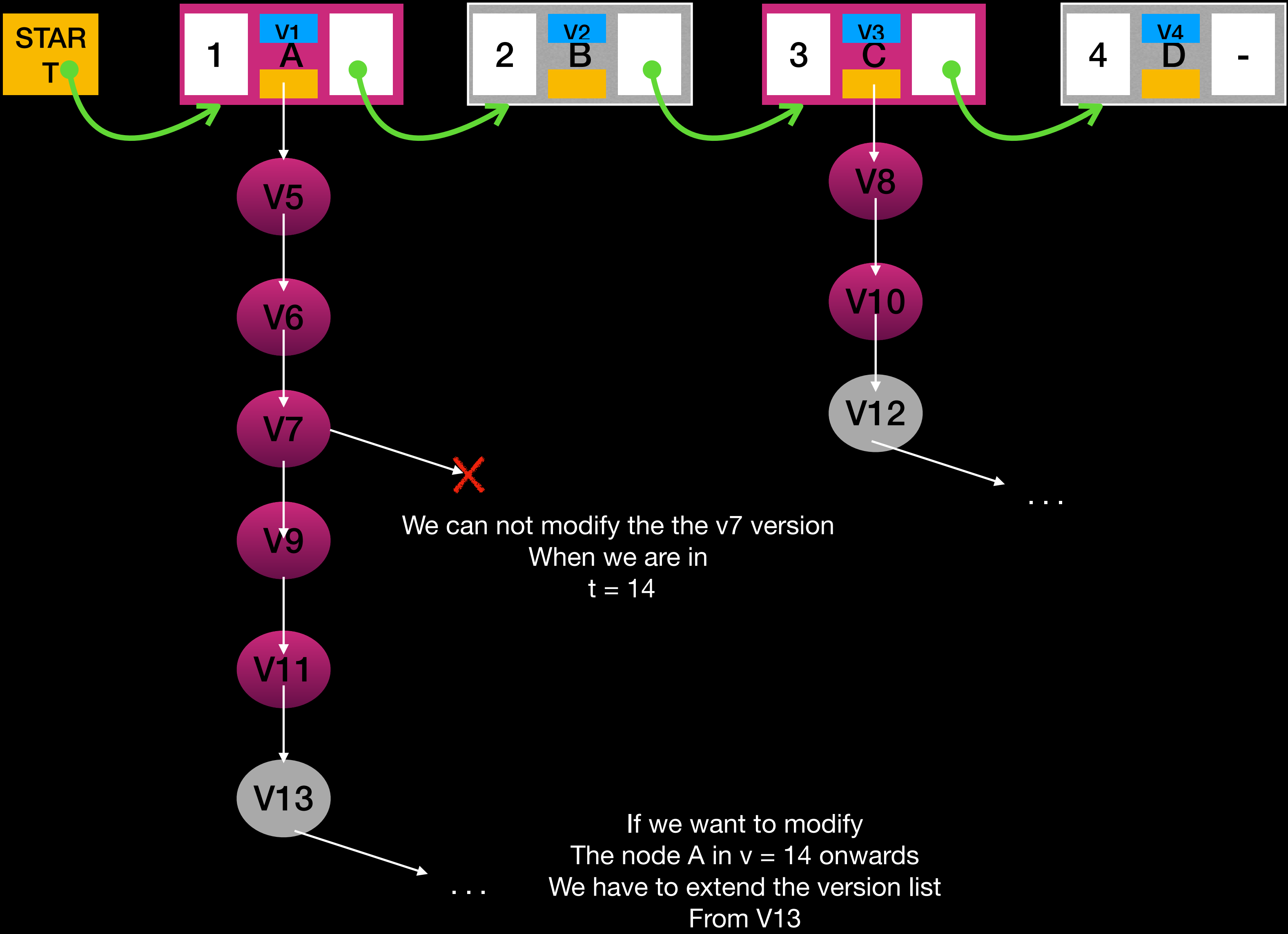
- `update(x,f_i,val,v)`

Why we need Full Persistent Data-structure?

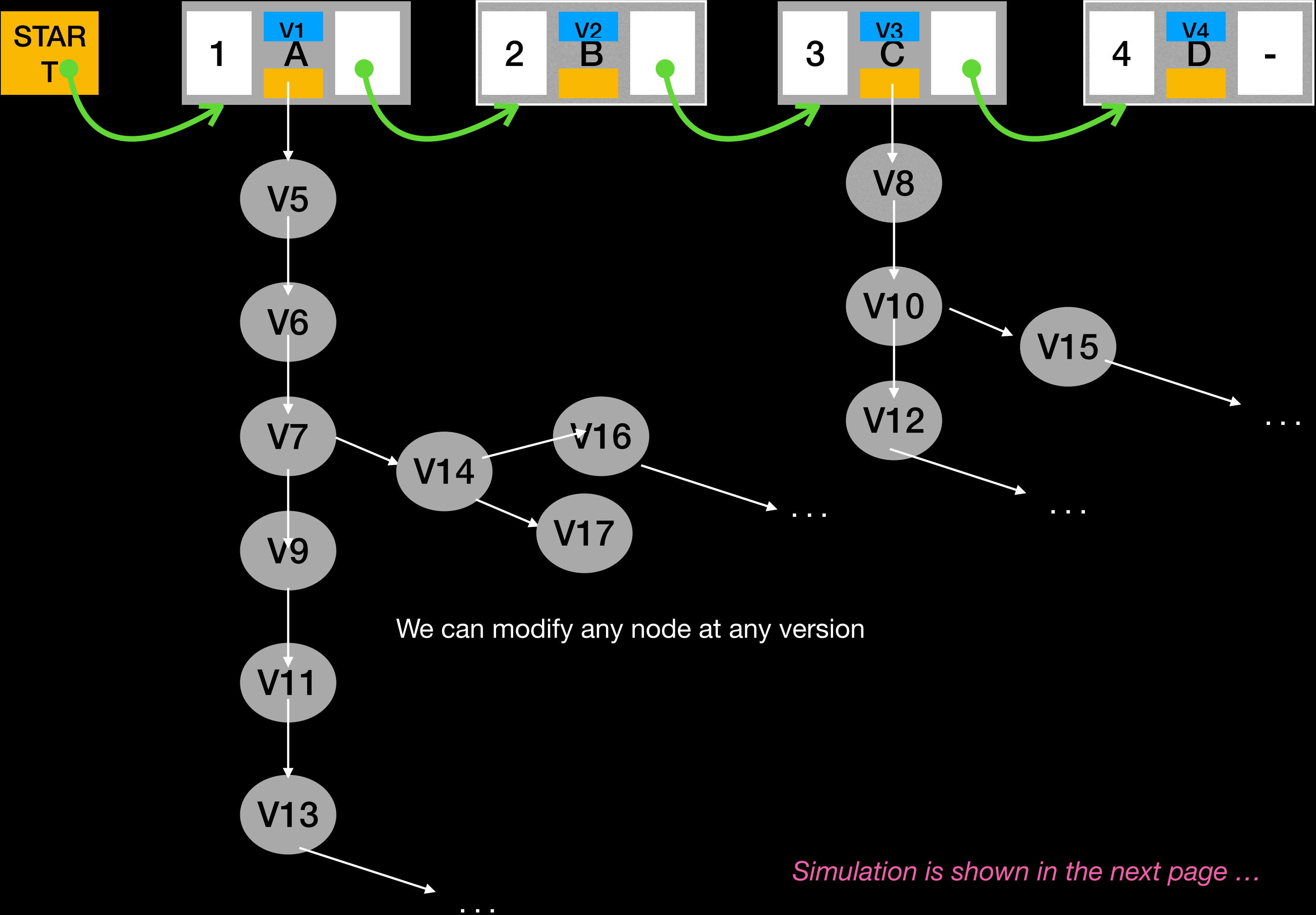
Basic Idea

- Now we can modify any pervious version.
- Branching of versions - is possible
- In Partial Persistent Data Structure we saw Linear Ordering of versions, pervious versions were in read-only state. We could modify the latest version of any node.
- But, in Full Persistent Mode, we can branch the version order using Version Tree (*optimisation can be done using Order Maintenance List*) and modify any node at any version.

Problem in Partial Persistent Mode



Remedy in Full Persistent Mode



No concept of Read Only Version

We can modify any node at any version

Simulation is shown in the next page ...

	Read Only
	Current Node (both read and modify)

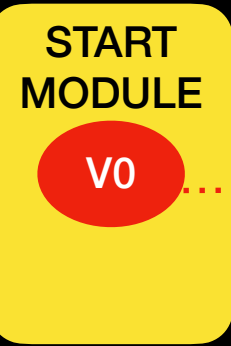
Query: init_LL()

Note: Here **VERSION** at each nodes/modules
are not jus numbers, rather they are
the **ADDRESS** of corresponding version nodes in
version tree

Current time, t = 0

Version tree

v0

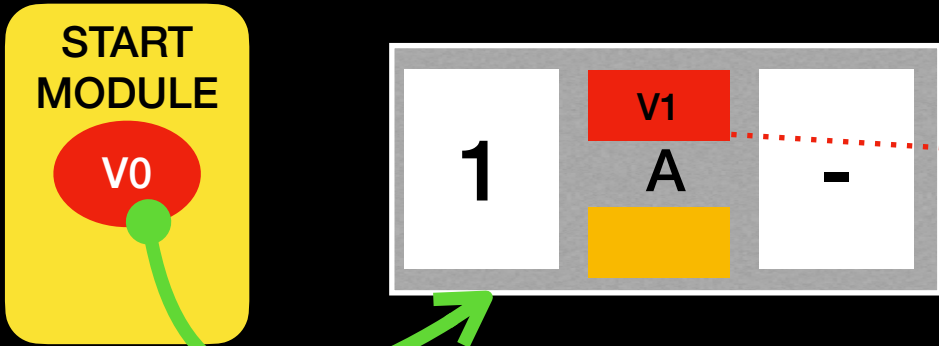
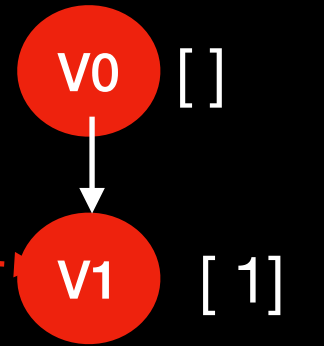


Query: add(A,_,v0,1)

Note: Here **VERSION** at each nodes/modules are not just numbers, rather they are the **ADDRESS** of corresponding version nodes in version tree

Current time, $t = 1$

Version tree

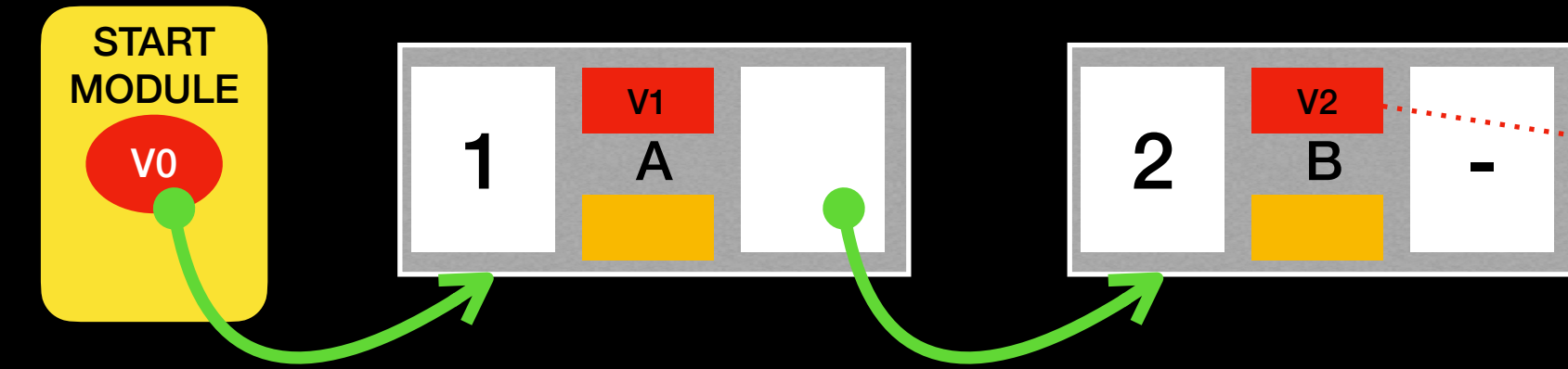
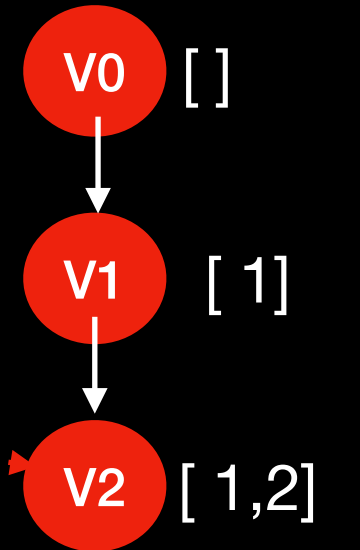


Query: add(B,A,v1,2)

Note: Here **VERSION** at each nodes/modules are not just numbers, rather they are the **ADDRESS** of corresponding version nodes in version tree

Current time, $t = 2$

Version tree

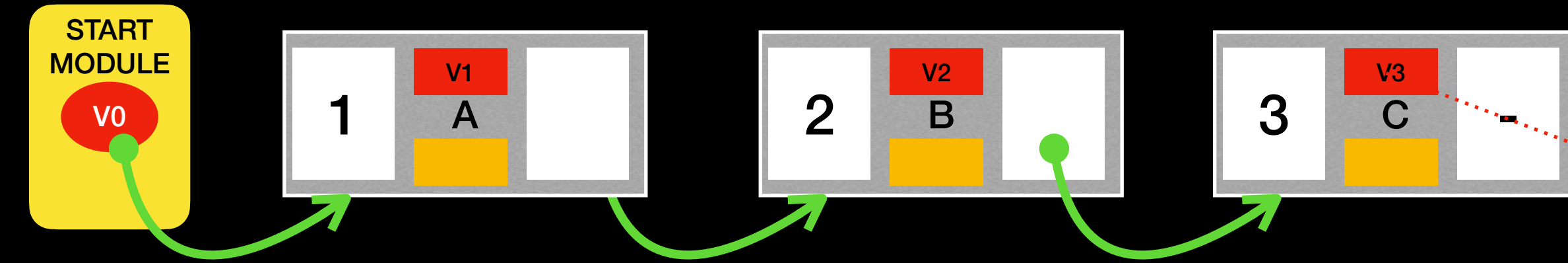
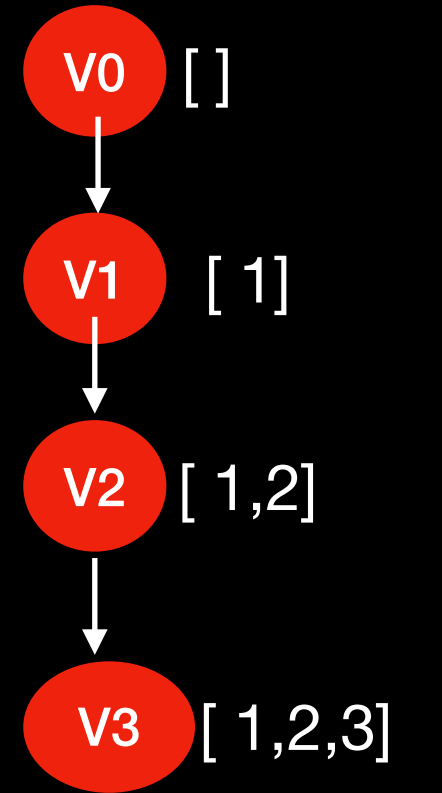


Query: add(C,B,v2,3)

Note: Here **VERSION** at each nodes/modules are not just numbers, rather they are the **ADDRESS** of corresponding version nodes in version tree

Current time, $t = 3$

Version tree



Query: add(D,C,v3,4)

Note: Here **VERSION** at each nodes/modules are not jus numbers, rather they are the **ADDRESS** of corresponding version nodes in version tree

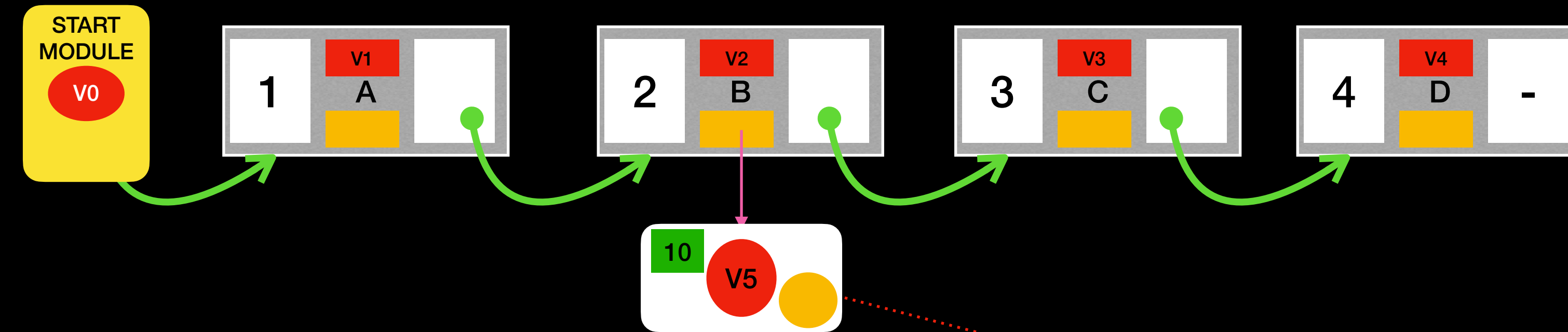
Current time, t = 4



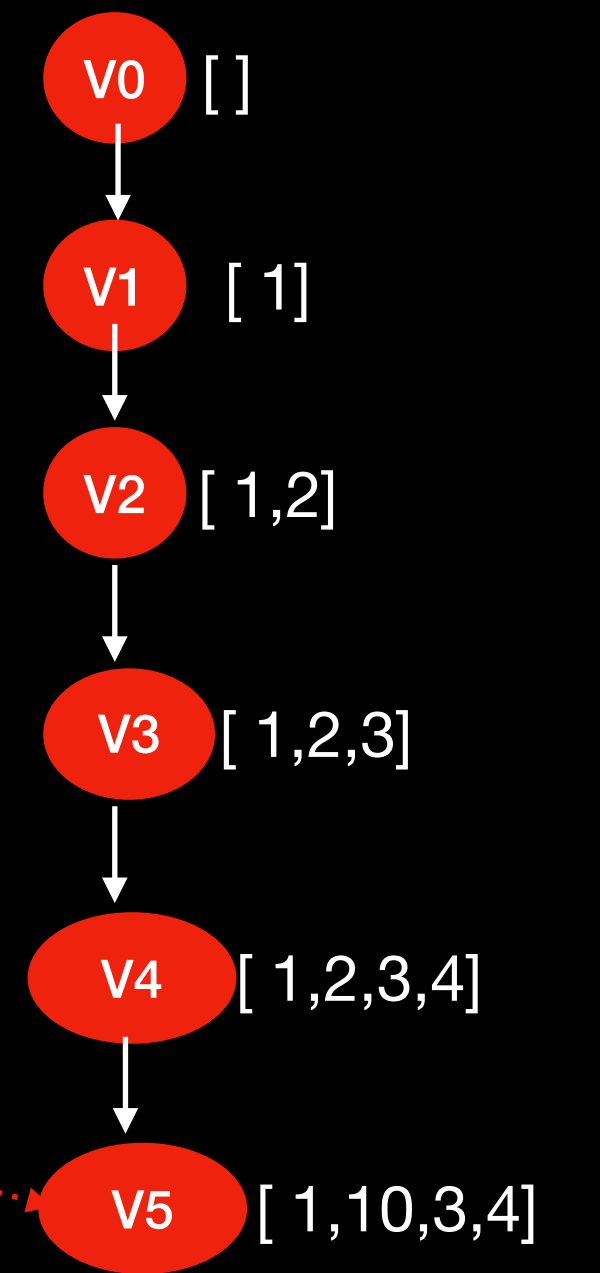
Query: update(B,f1,10,v4)

Note: Here **VERSION** at each nodes/modules are not just numbers, rather they are the **ADDRESS** of corresponding version nodes in version tree

Current time, $t = 5$



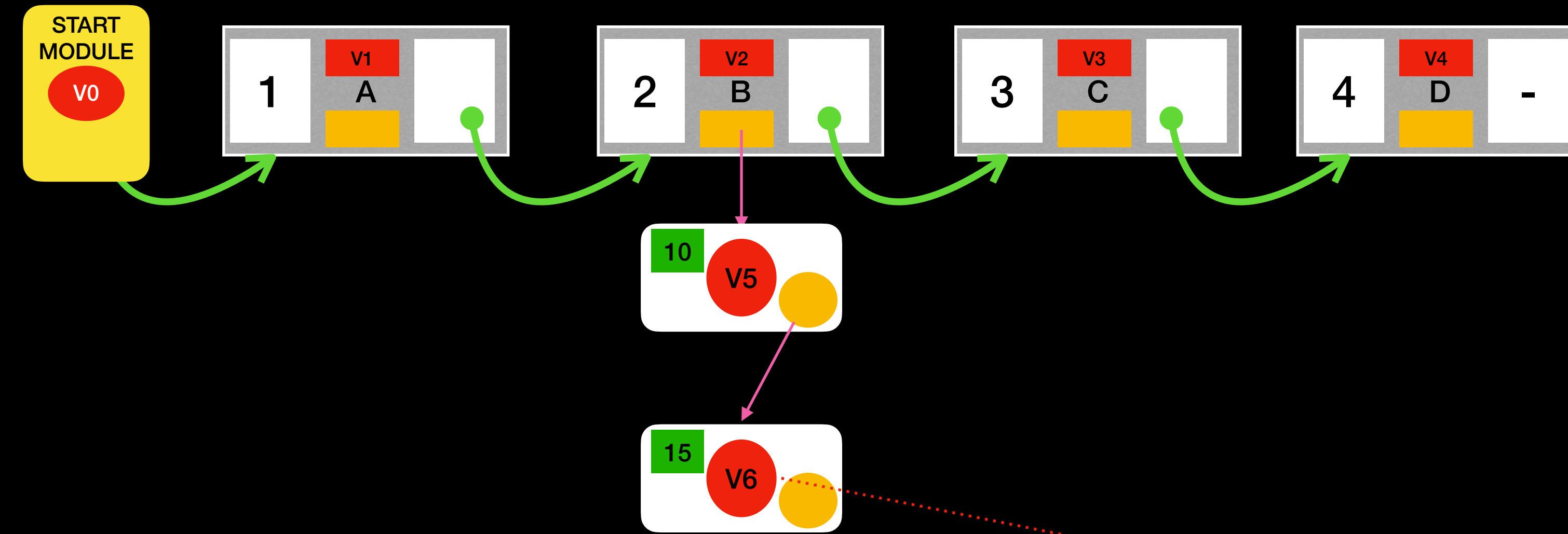
Version tree



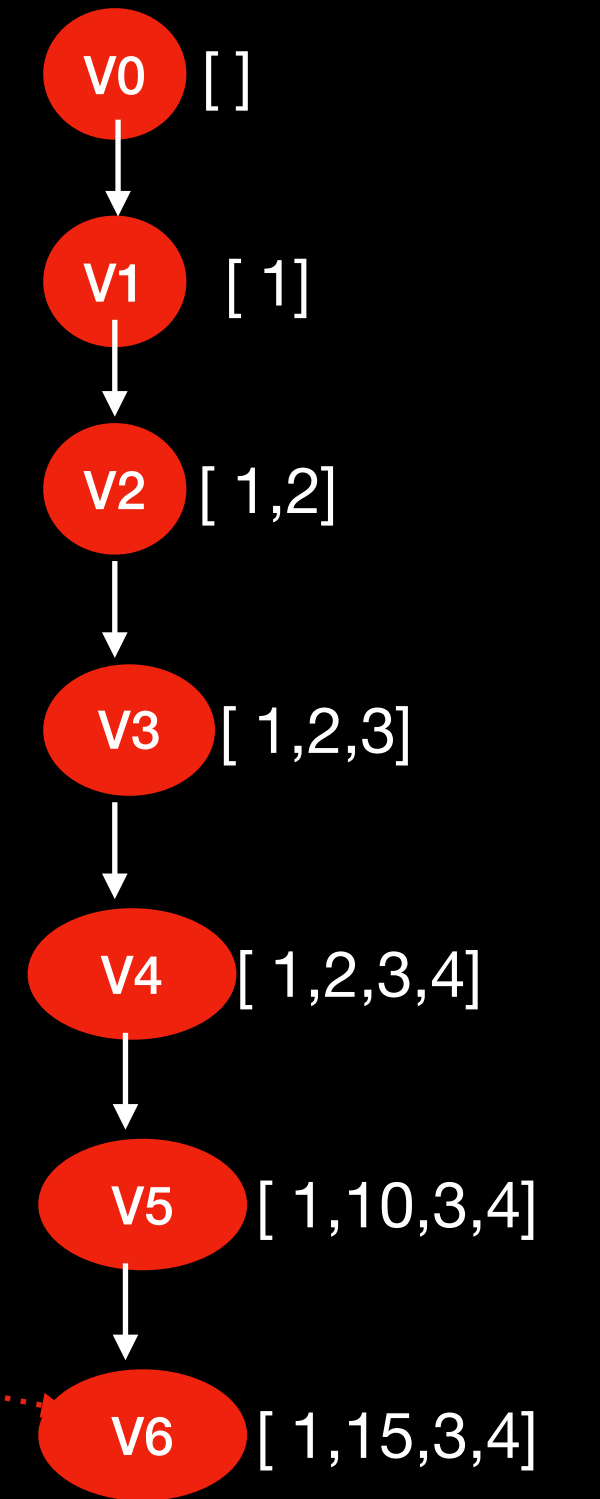
Query: update(B,f1,15,v5)

Note: Here **VERSION** at each nodes/modules are not jus numbers, rather they are the **ADDRESS** of corresponding version nodes in version tree

Current time, t = 6



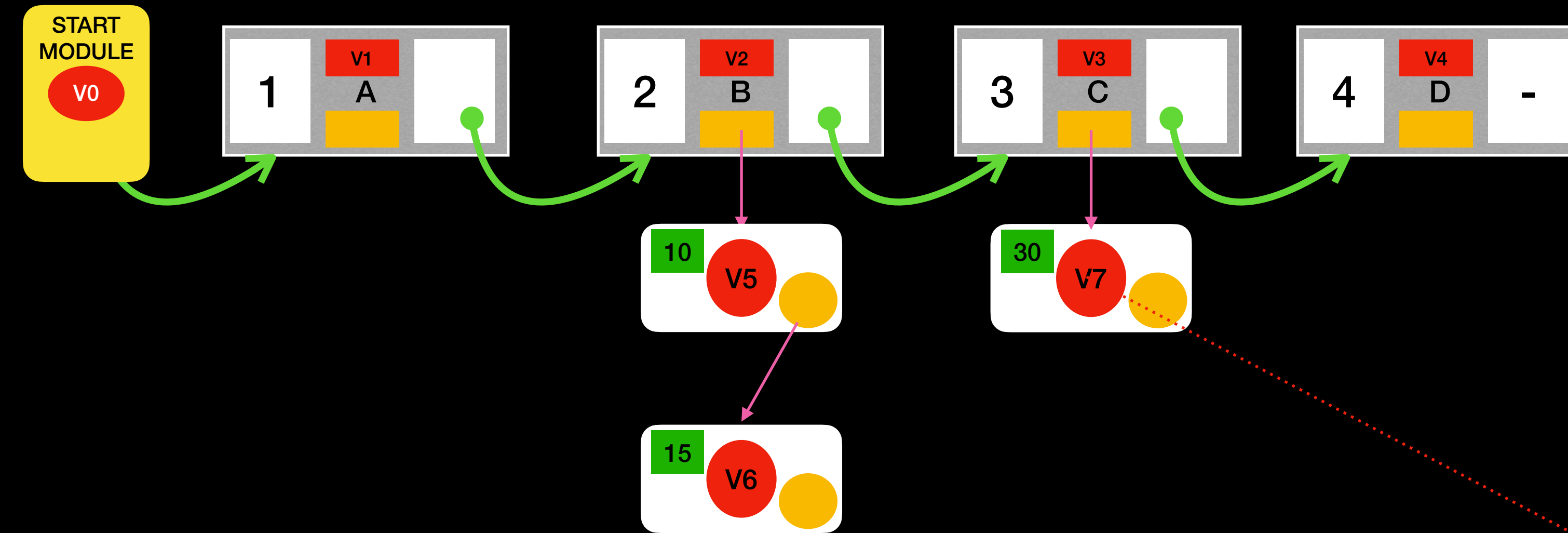
Version tree



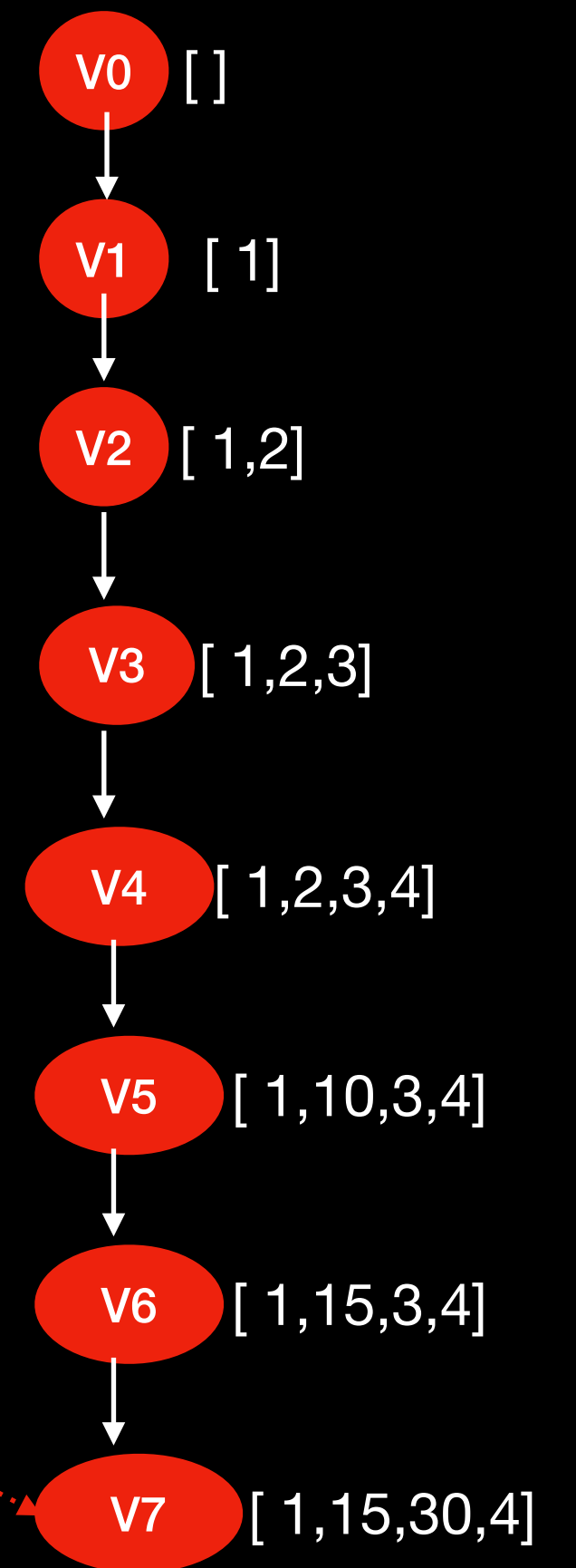
Query: update(C,f1,30,v6)

Note: Here **VERSION** at each nodes/modules are not jus numbers, rather they are the **ADDRESS** of corresponding version nodes in version tree

Current time, t = 7

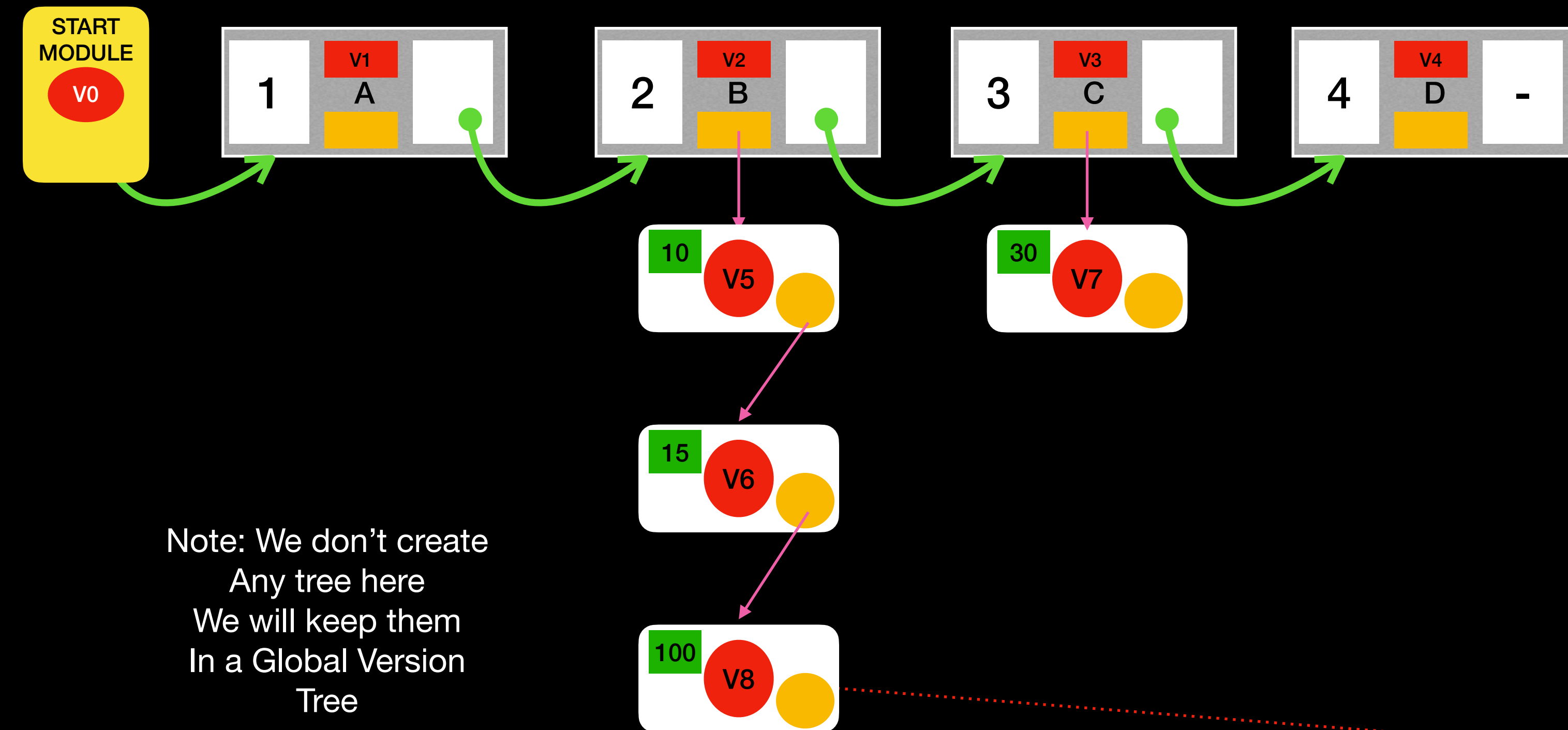


Version tree



Query: update(B,f1,100,v5)

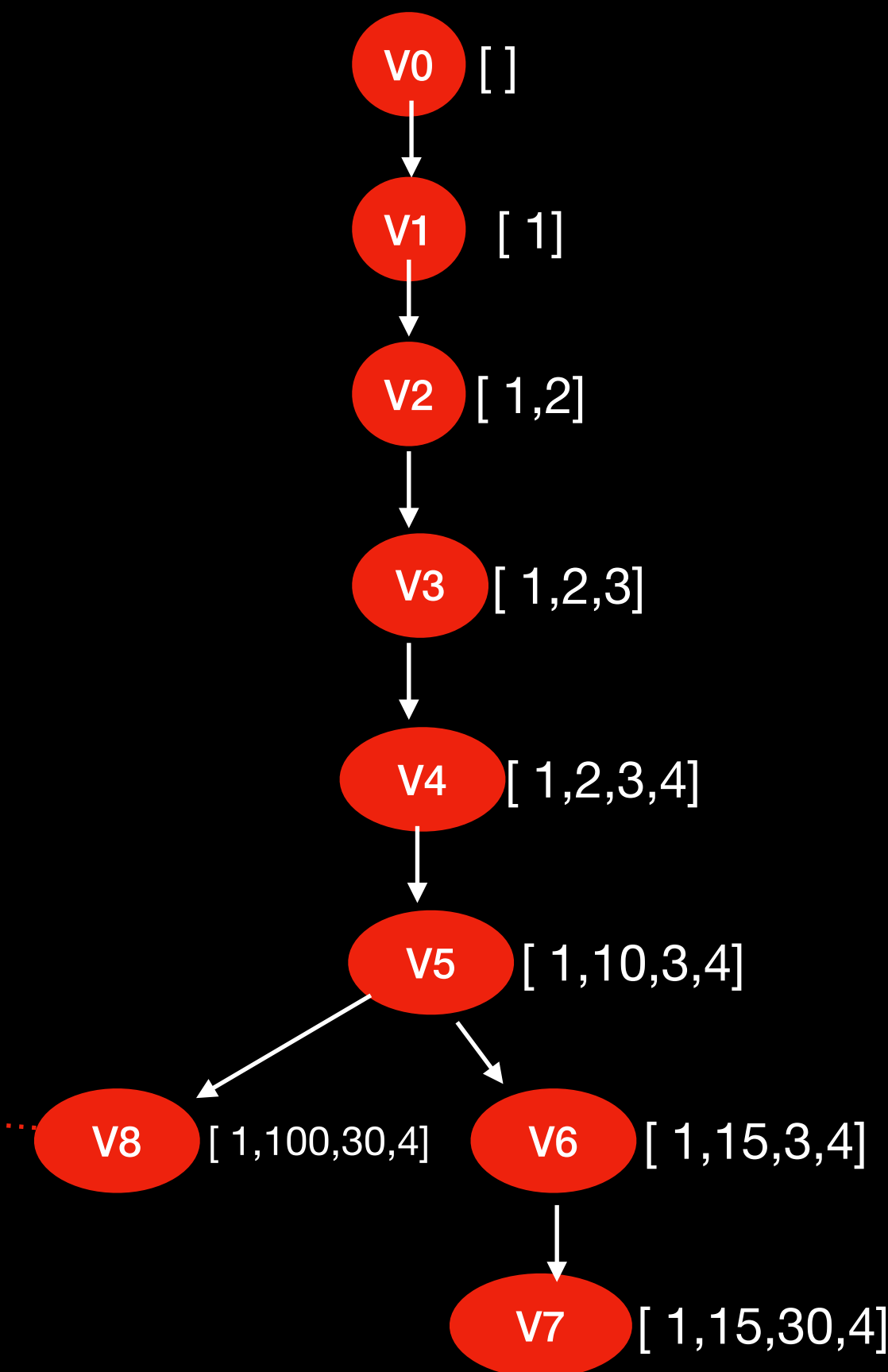
Current time, $t = 8$



Note: We don't create
Any tree here
We will keep them
In a Global Version
Tree

As, That should be
Accessible to all nodes

Version tree



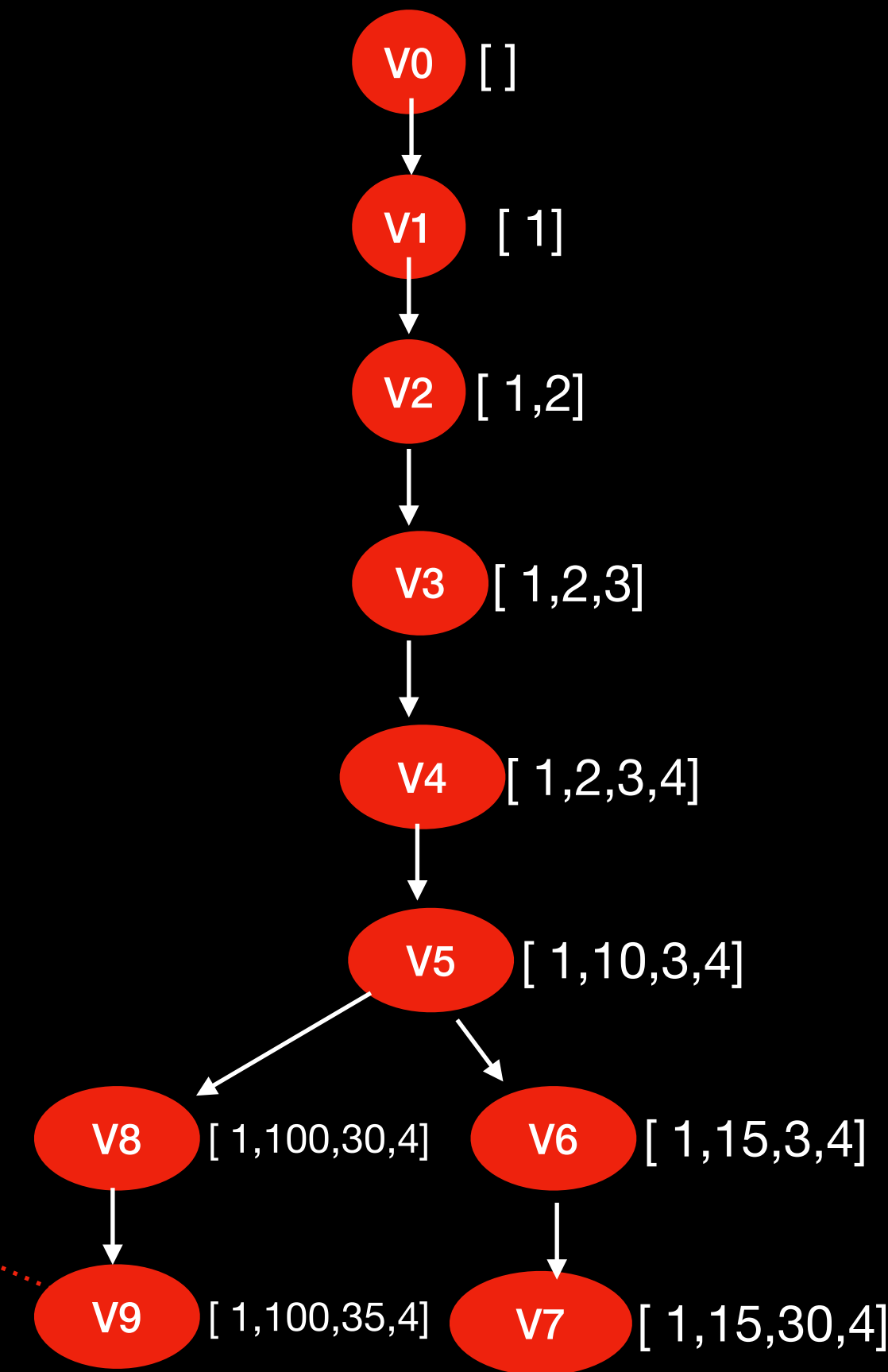
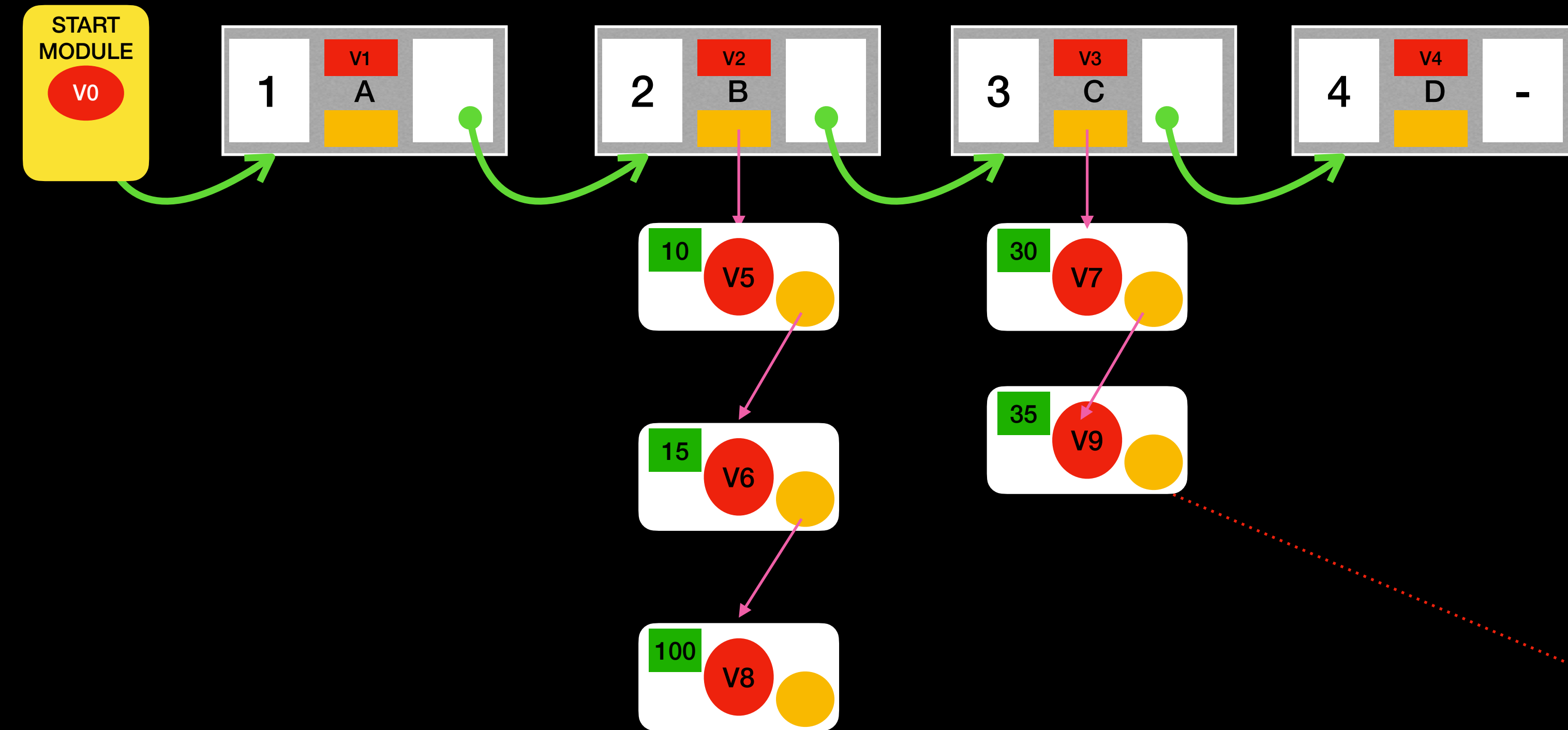
How to do branching ?

1. Create new Update Module with version = v8 and updated fields
2. Just add v8 node to v5 node in v-tree

Query: update(C,f1,35,v8)

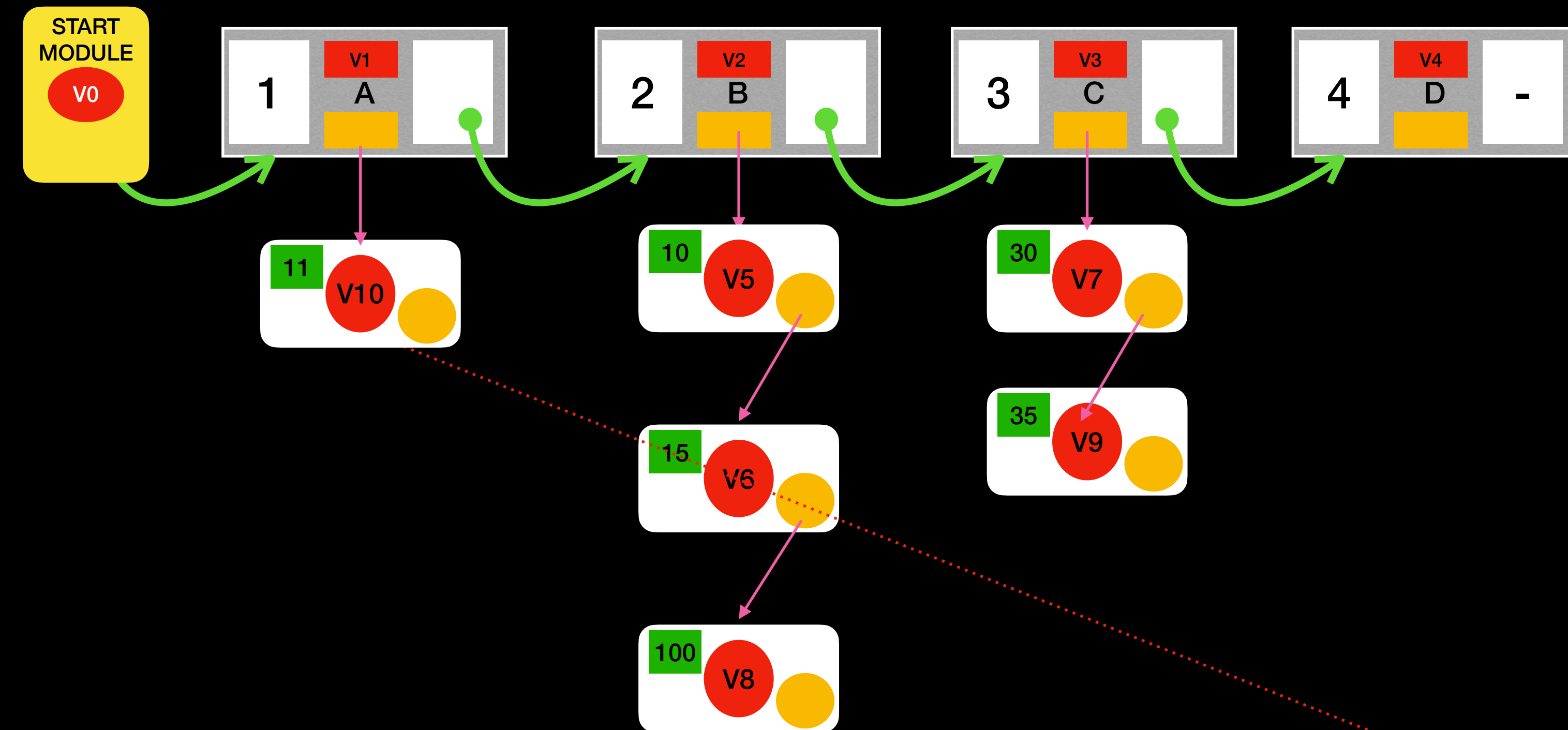
Current time, $t = 9$

Version tree

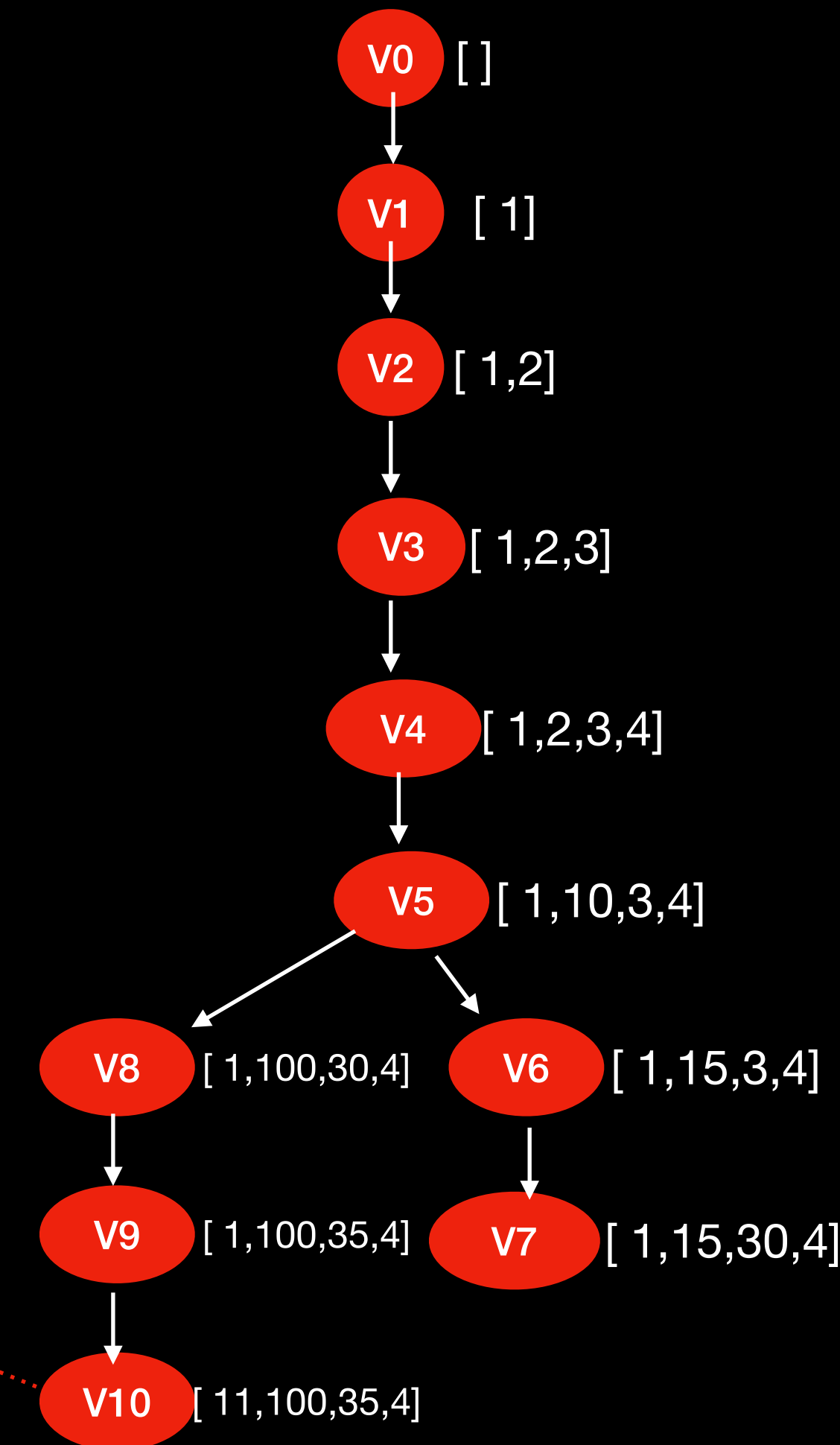


Query: update(A,f1,11,v9)

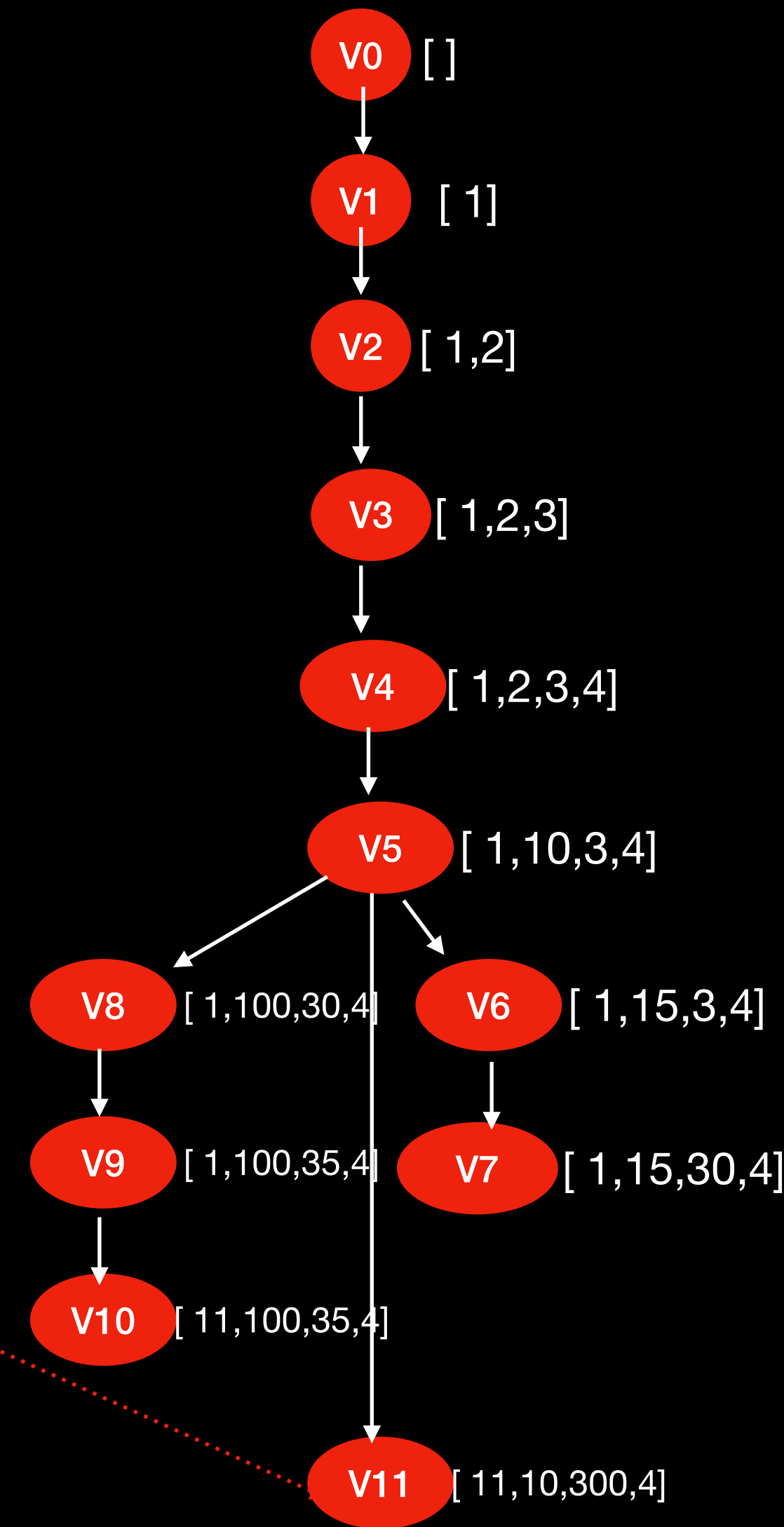
Current time, $t = 10$



Version tree

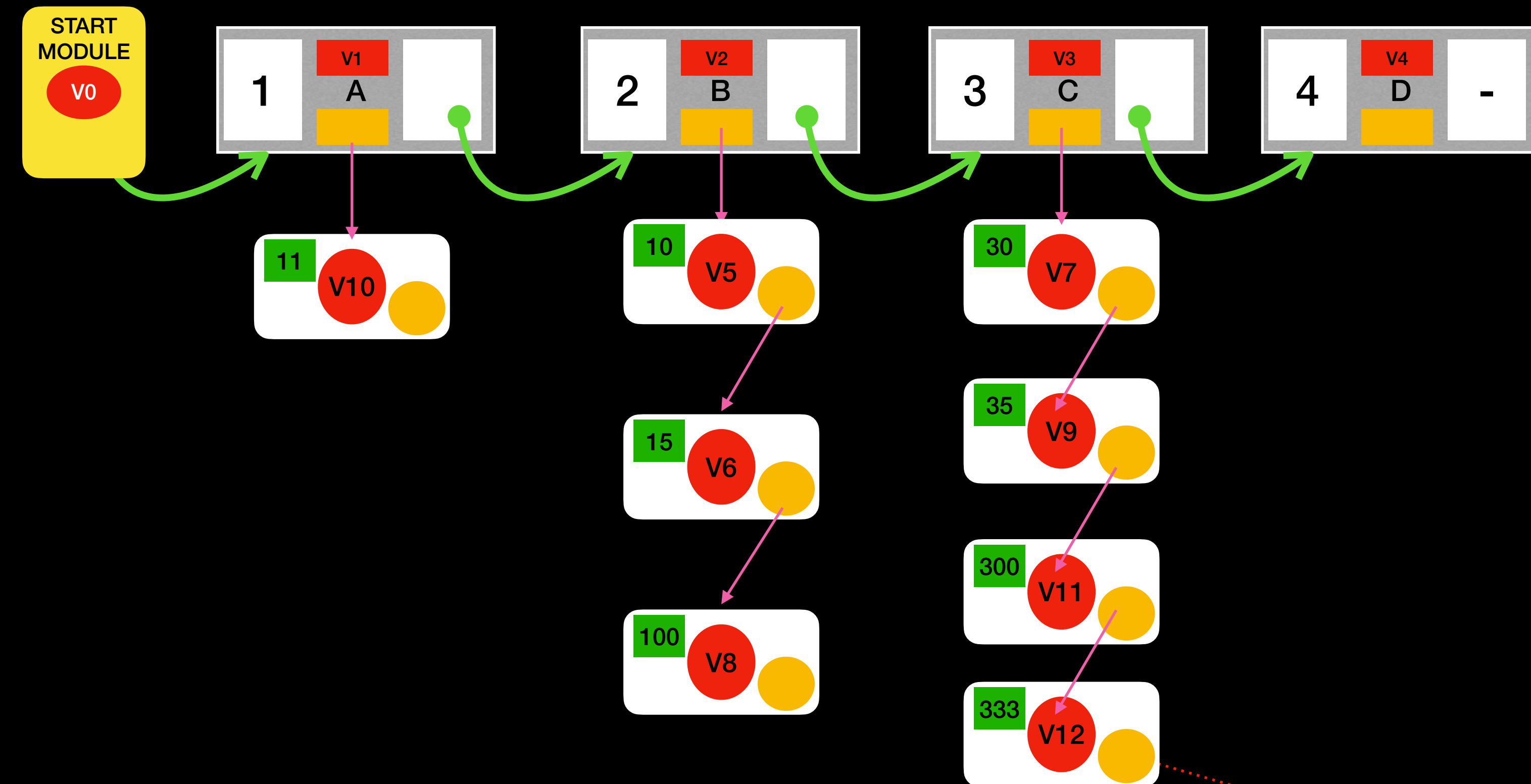


Current time, $t = 11$

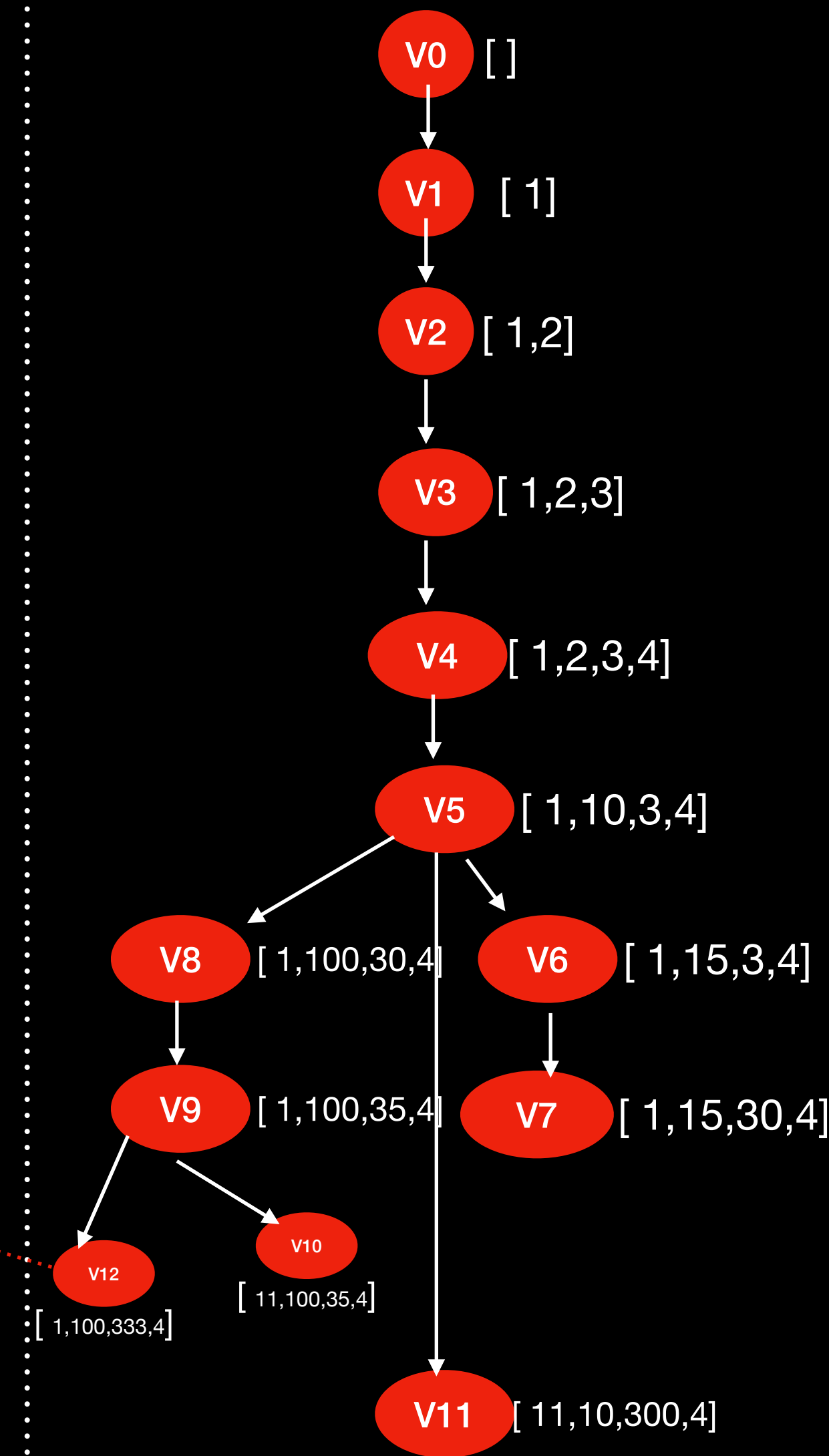


Query: update(C,f1,333,v9)

Current time, $t = 12$

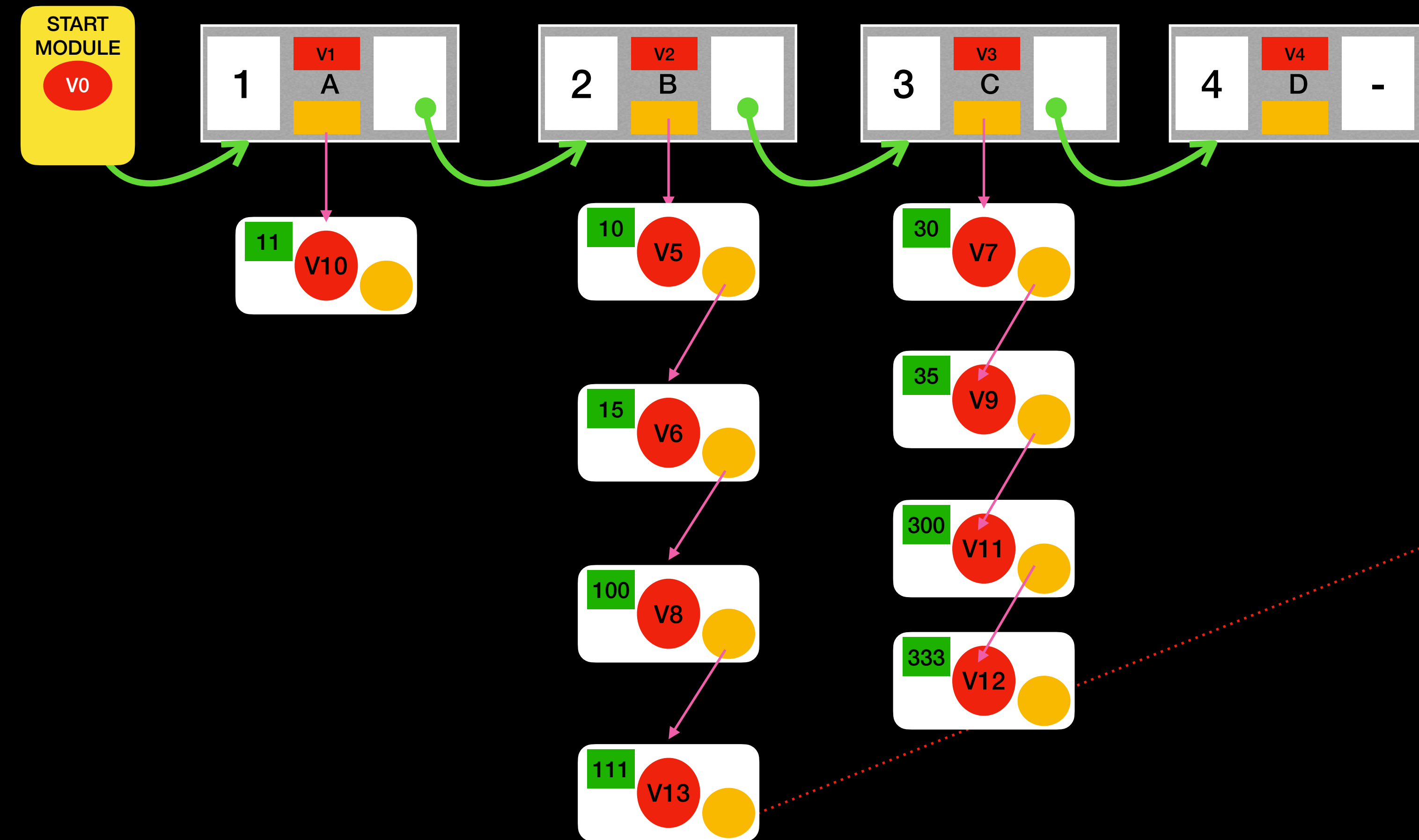


Version tree

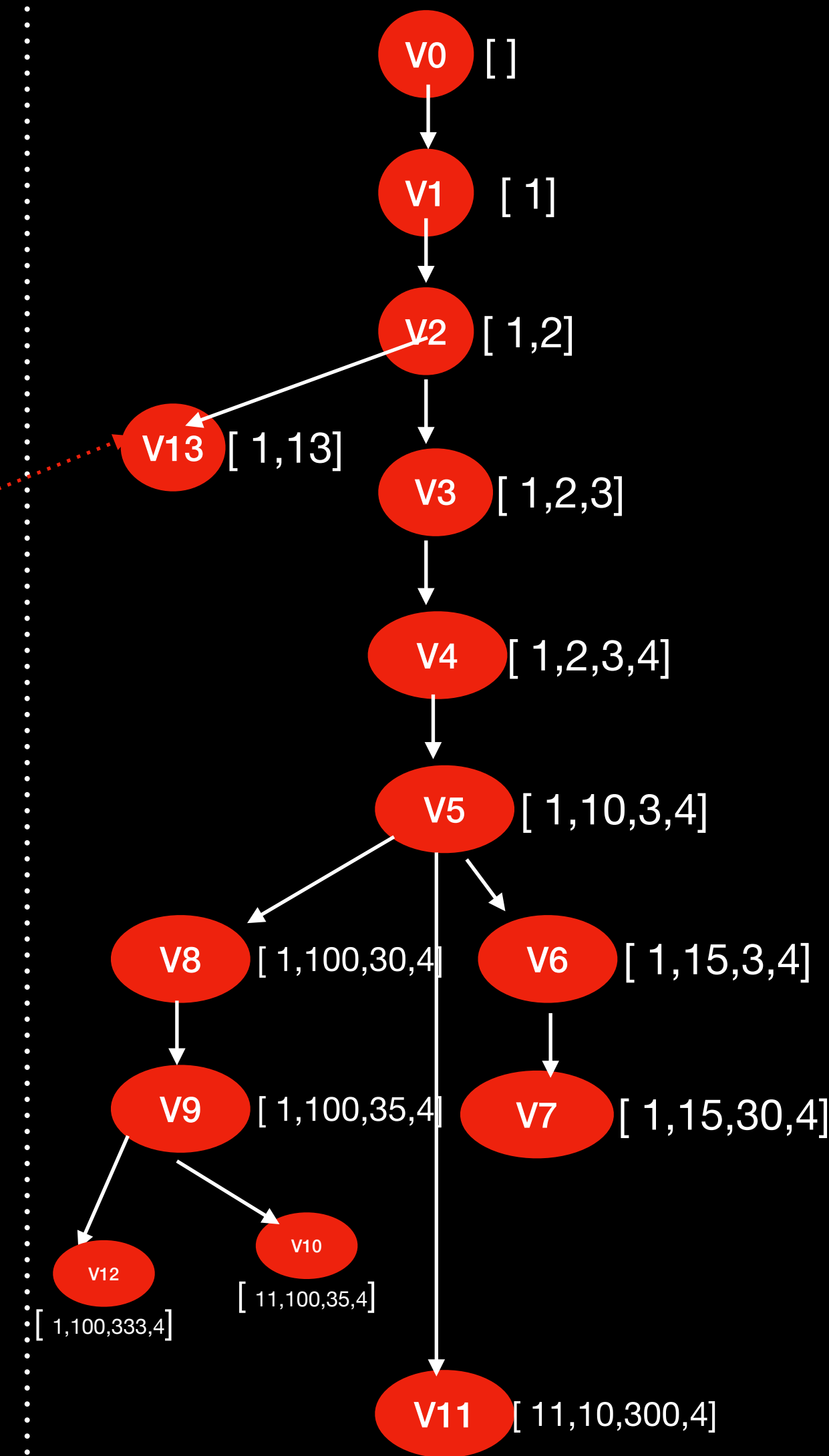


Query: update(B,f1,111,v2)

Current time, $t = 13$

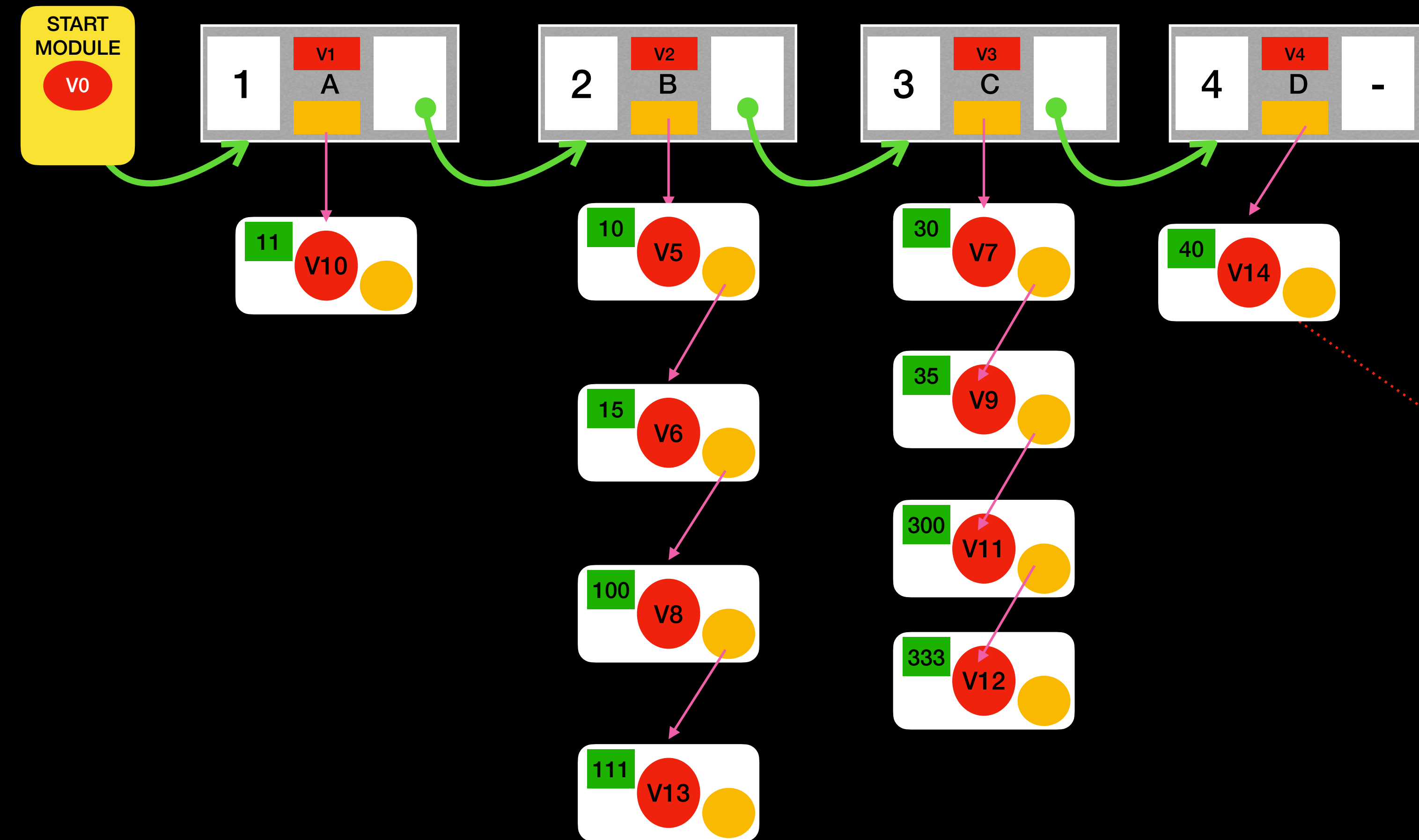


Version tree

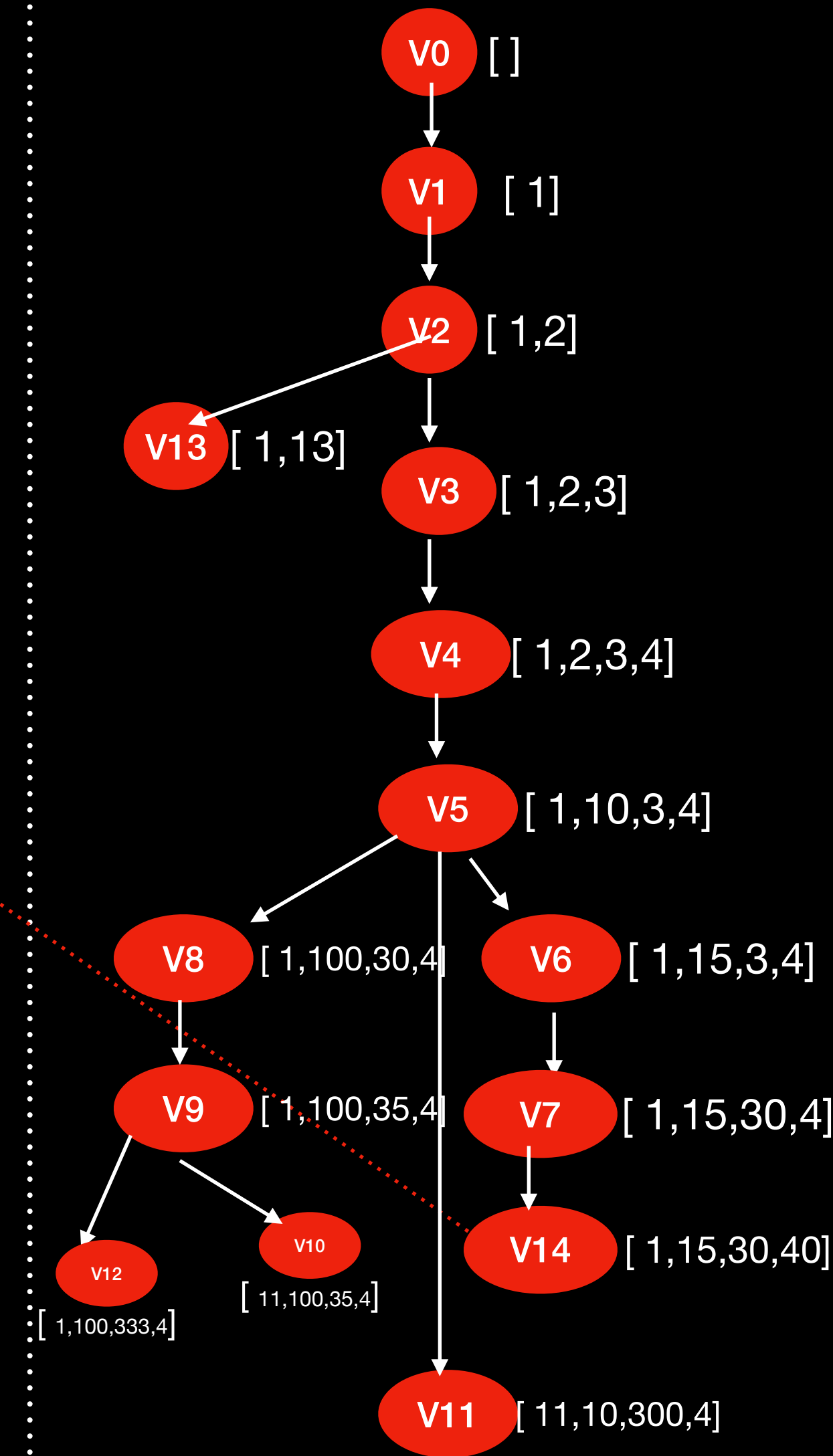


Query: update(D,f1,40,v7)

Current time, $t = 14$

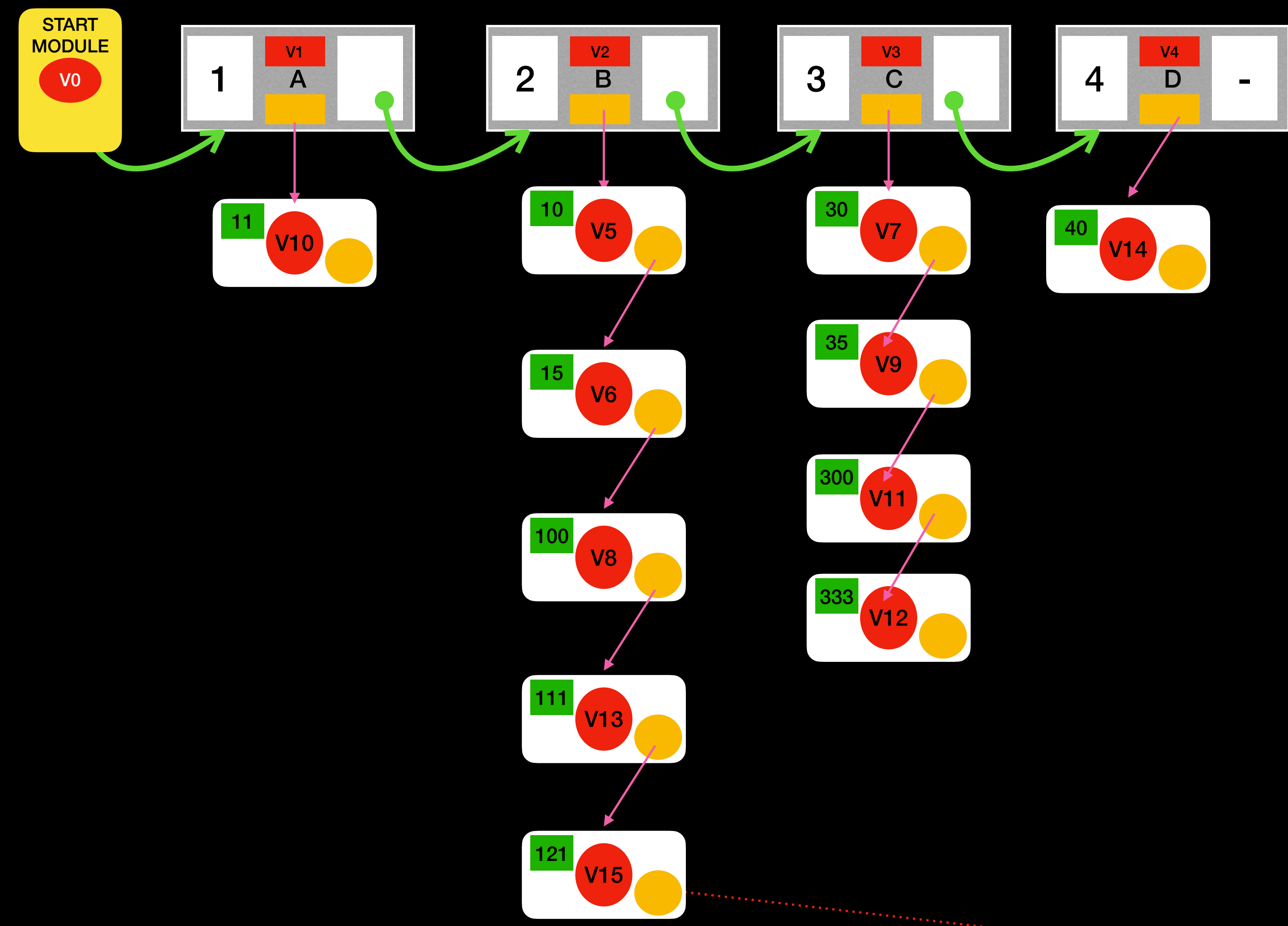


Version tree

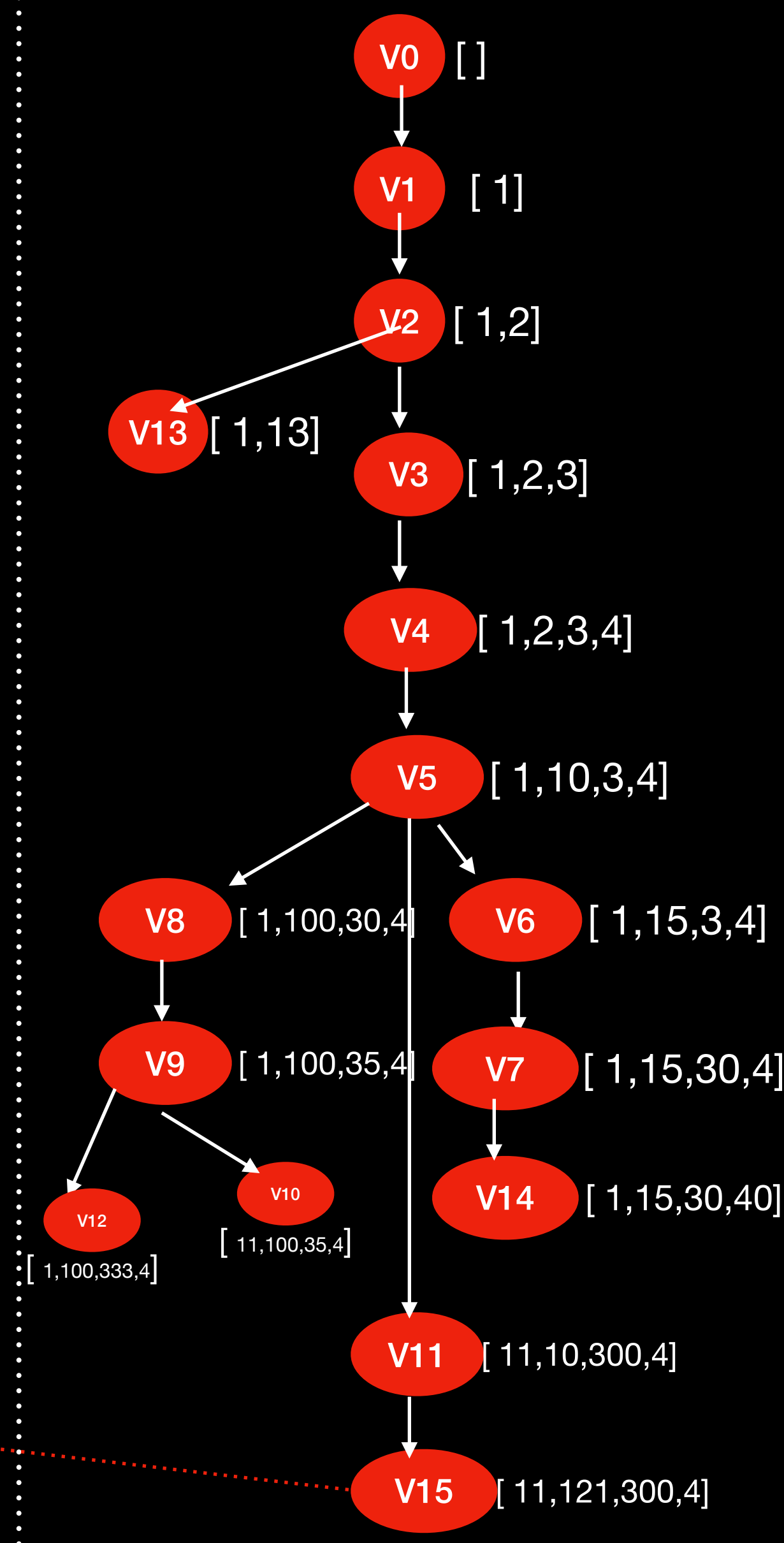


Query: update(B,f1,121,v11)

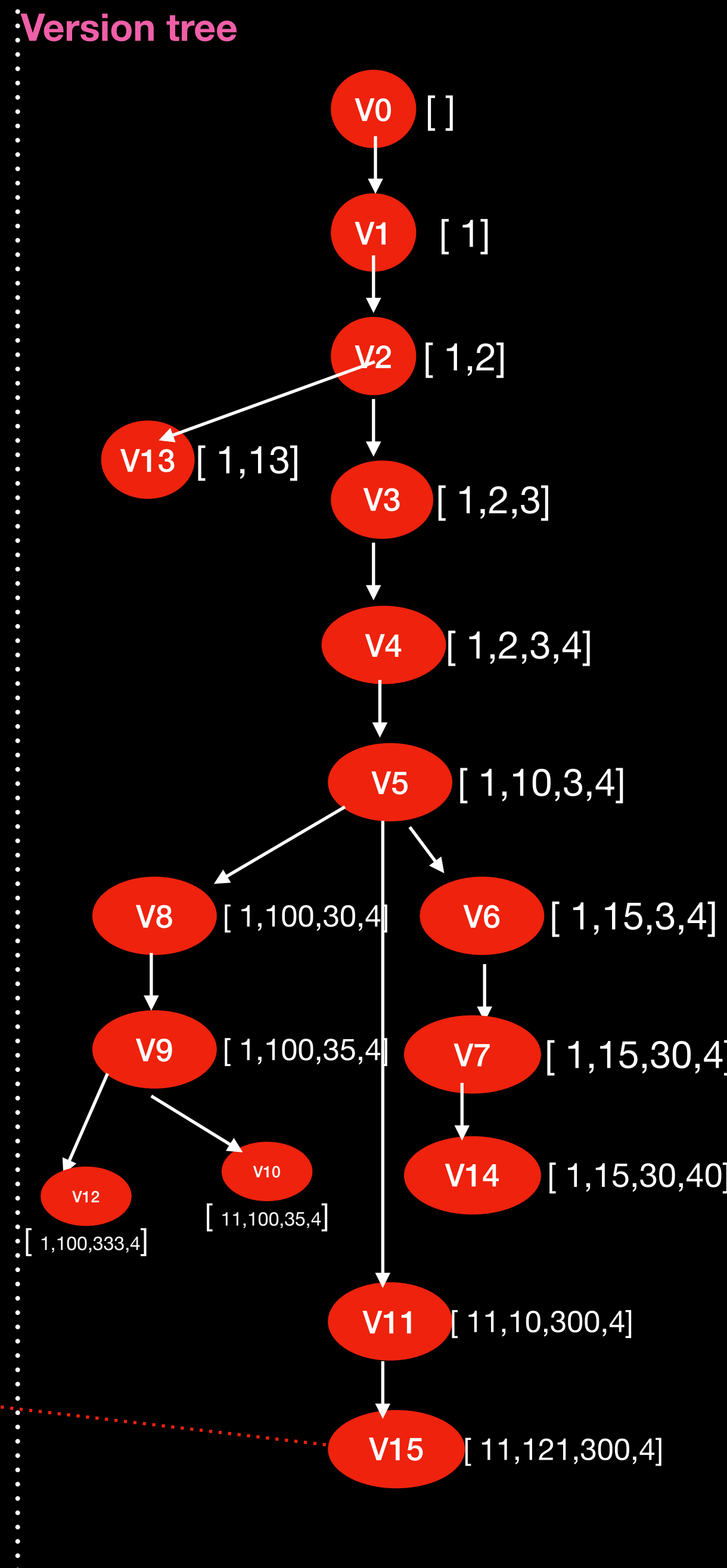
Current time, t = 15



Version tree



Current time, $t = 15$



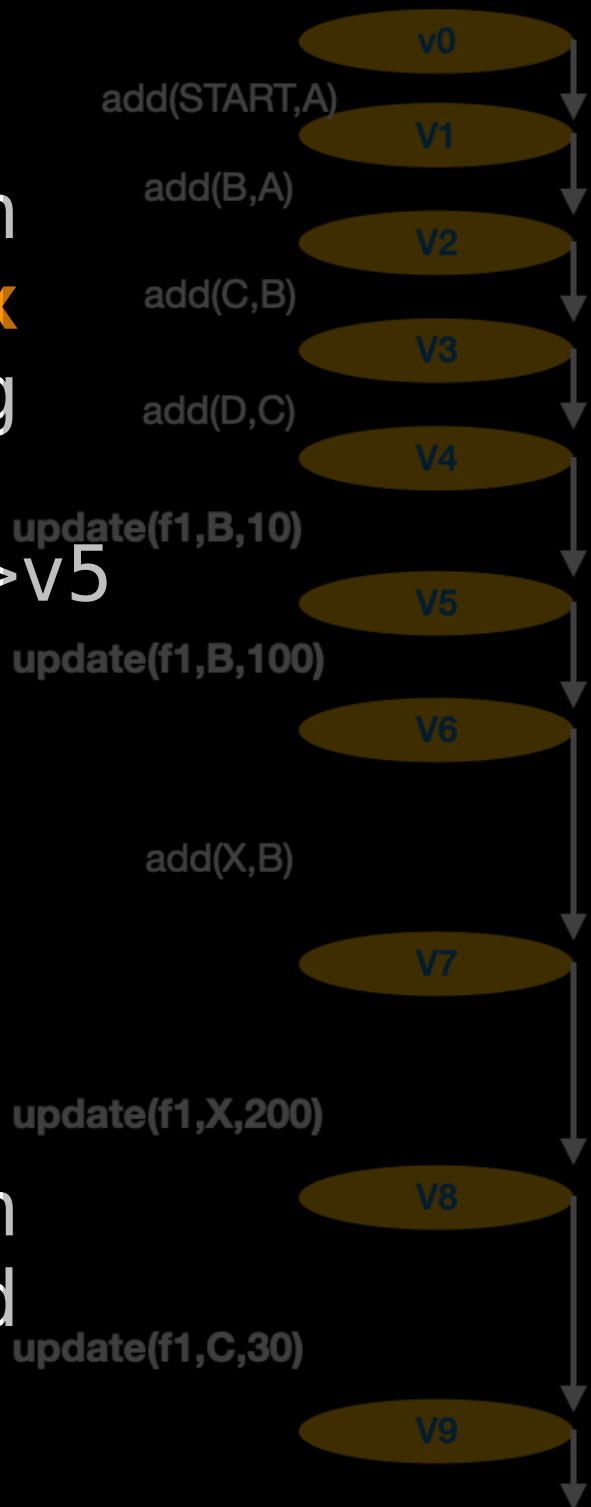
Ok cool !! How to iterate through list in version v

Iteration in Partial Mode vs Full Mode

iterate_LL_at_v(vx)

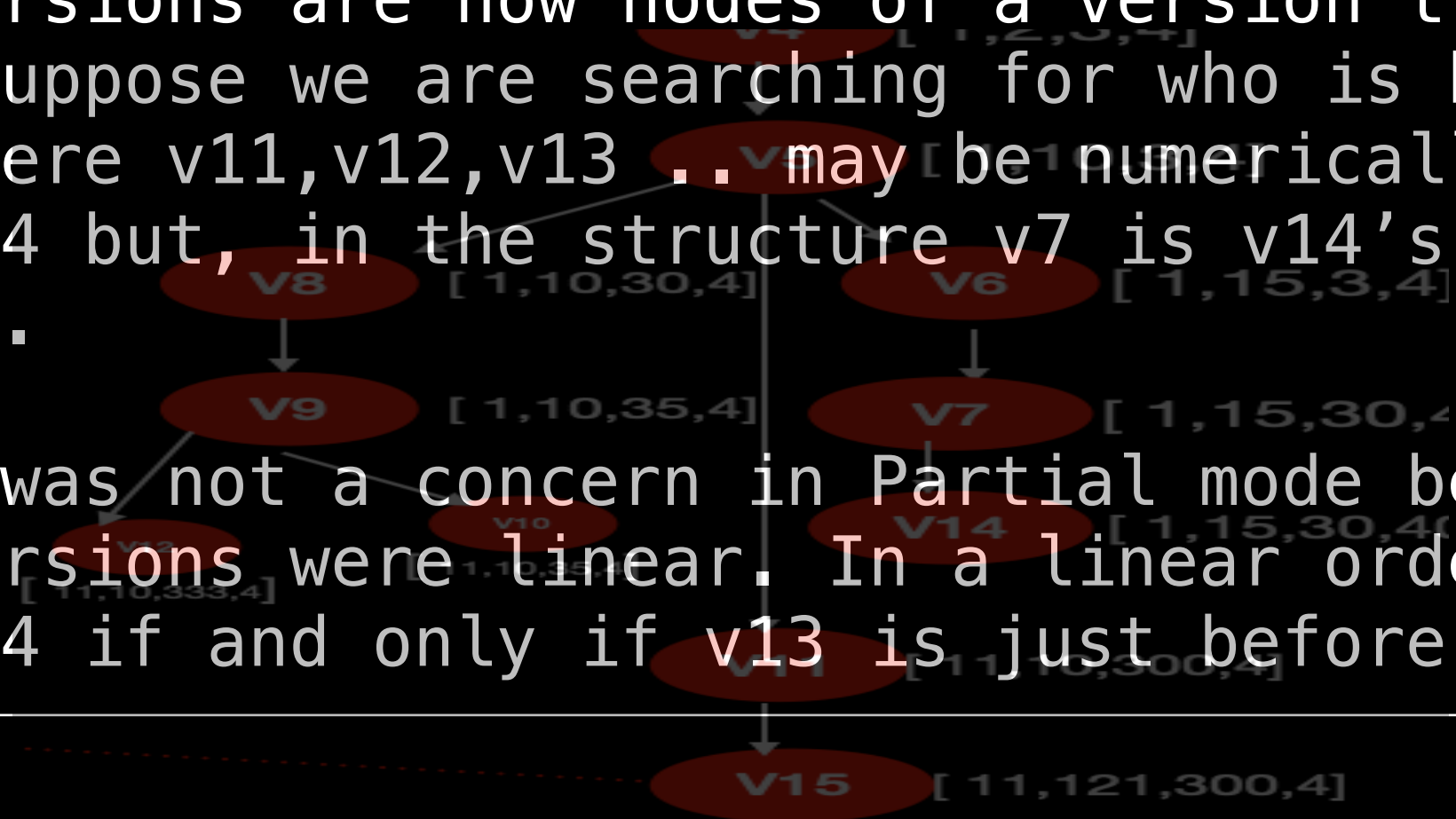
Partial Mode

1. Start from start module
2. Choose those lines whose version **IS JUST LESS THAN OR EQUAL TO v_x** (as, suppose if we are traversing for v5 , either v0,v1,v2,v3, v4 or v5 can be on that path, lines>v5 can't be on that).
3. The word "JUST LESS" is written because if a NODE has two version lines in that, e.g. v2 and v4 and we are searching for v5, then we should prefer v4 line over v2.



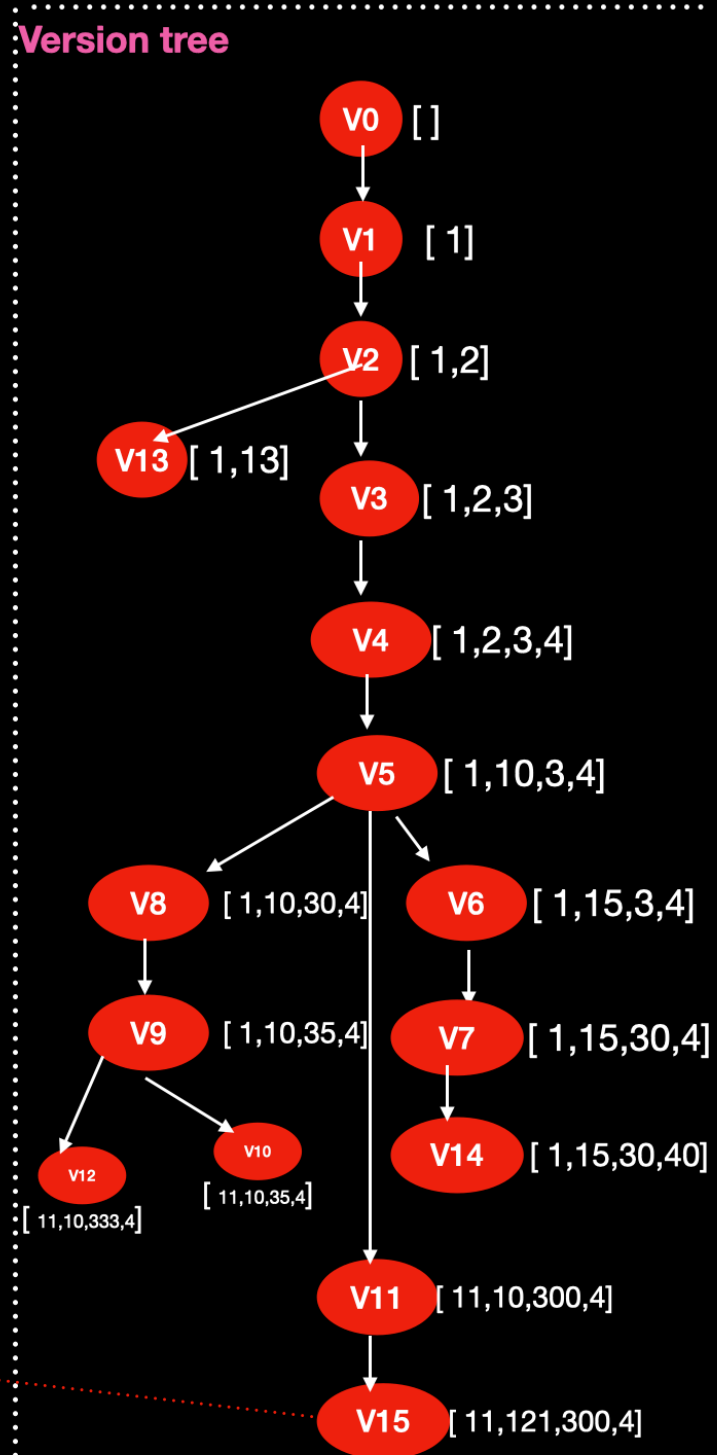
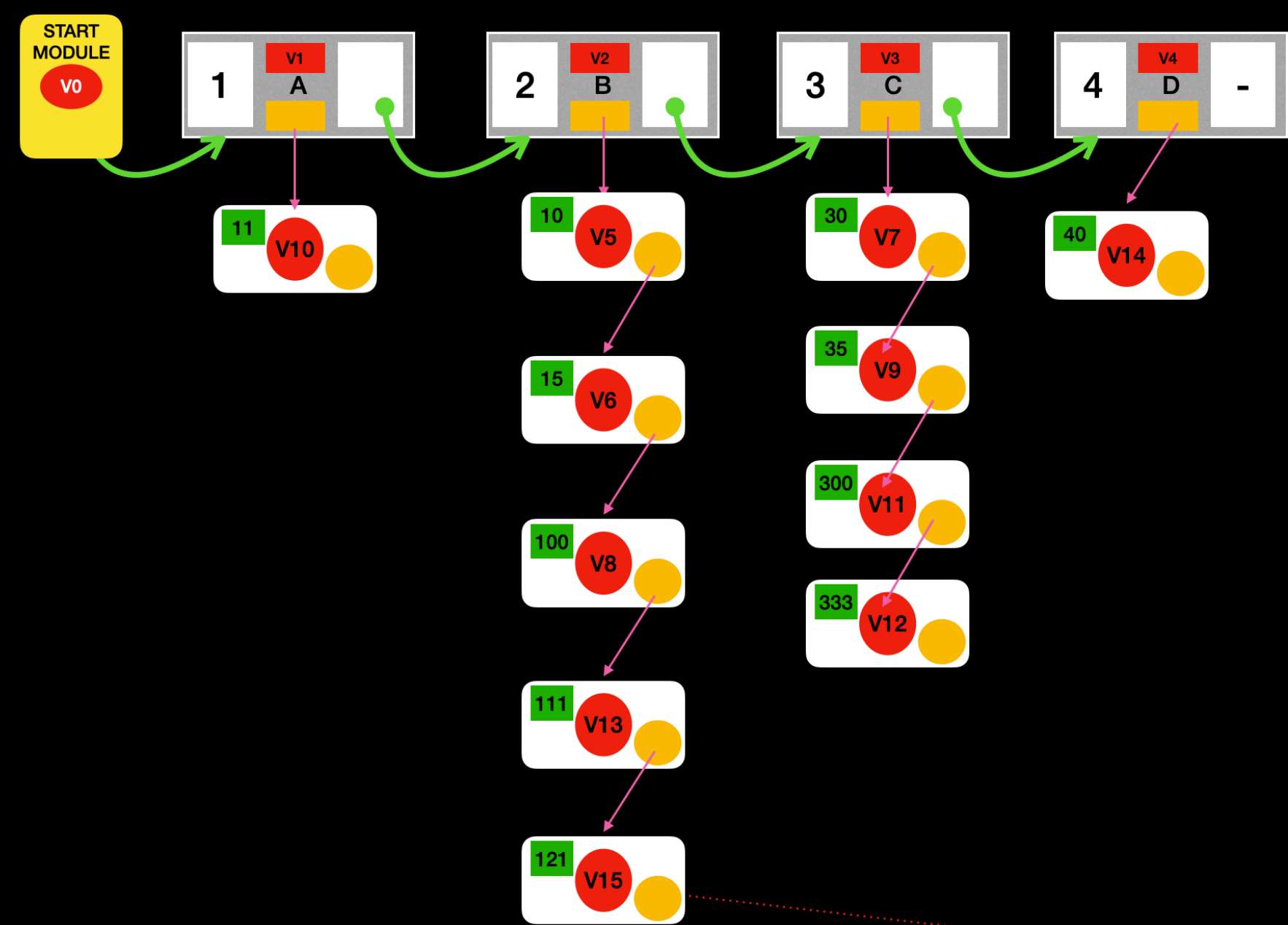
Full Mode

1. Start from start module
2. Choose those lines whose VERSION IS **IS NEAREST/LOWEST ANCESTOR OR EQUAL TO THAT OF v_x**
3. Here, we are searching for LOWEST ANCESTOR Because, versions are now not just number. The versions are now nodes of a version tree. E.g. suppose we are searching for who is before v14, here v11,v12,v13 .. may be numerically less than 14 but, in the structure v7 is v14's parent.
4. It was not a concern in Partial mode because the versions were linear. In a linear order v13<v14 if and only if v13 is just before v14.



Question!!

How to choose **NEAREST/LOWEST ANCESTOR** from all possible ancestors ?



Suppose, we are searching for v14
Status in node B,.

Note that, v2, v5, v6 are possible ancestors.
But, v6 is Nearest Ancestor of v14

How Do I Find that?

**ALL POSSIBLE ANCESTORS MUST BE IN A LINEAR ORDER.
AND, WE KNOW THAT THE VERSION WITH HIGHEST MAGNITUDE IS THE
MOST RECENT IN LINEAR ORDER**

Hence, we can say

Full Persistent Data-structure

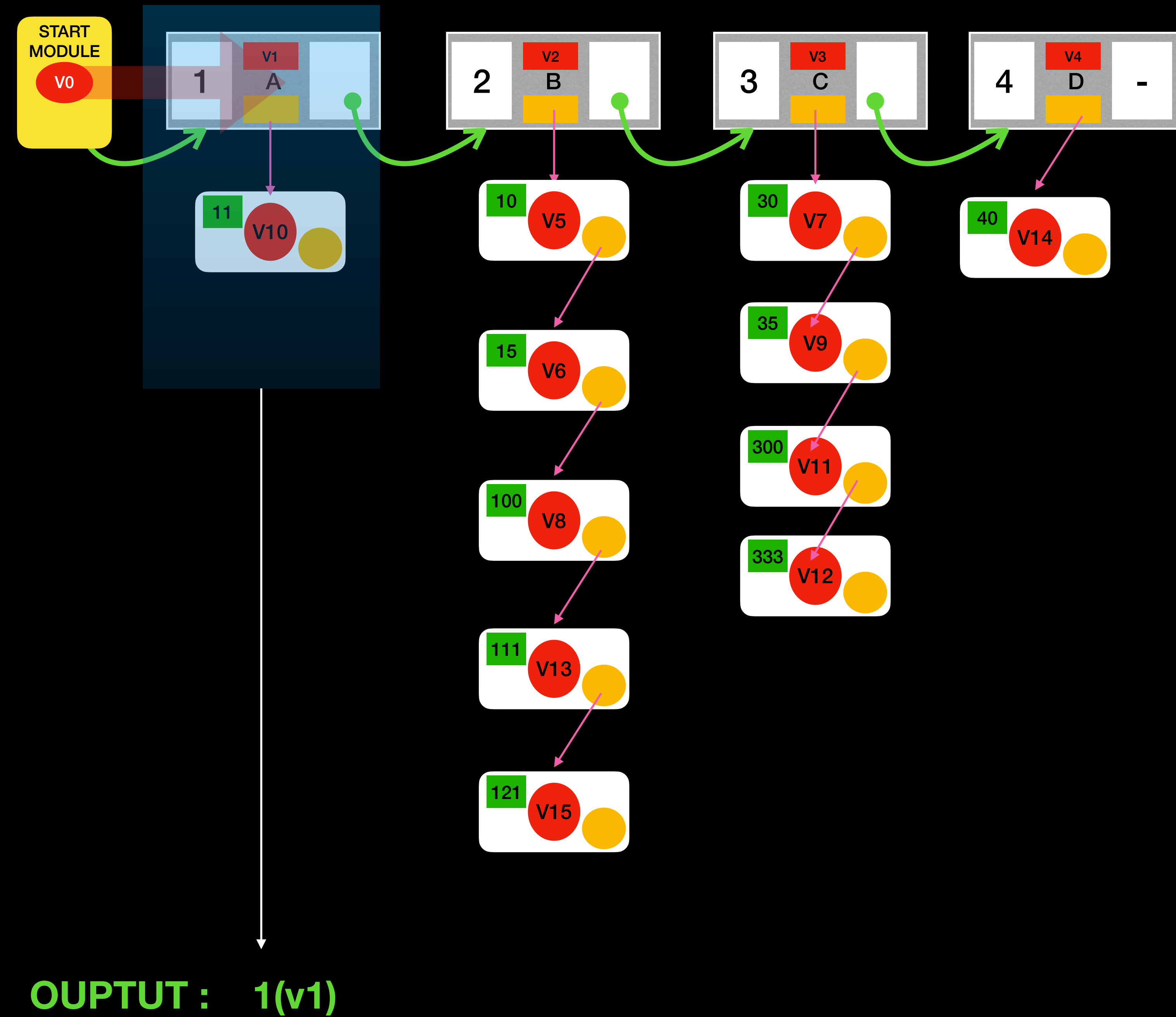
is a general form of

Partial Persistent Data-structure



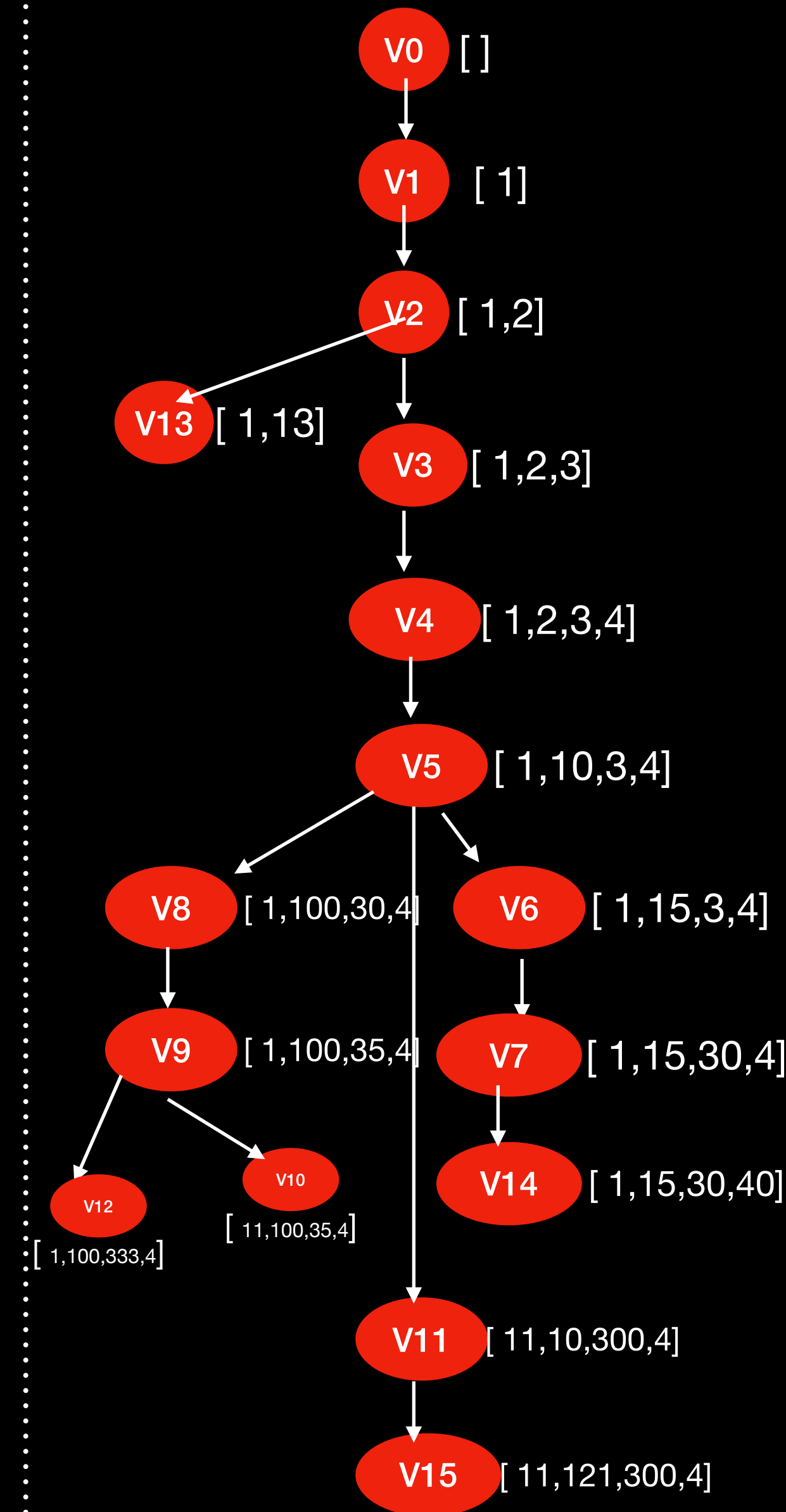
iterate_LL_at_v(v12)

iterate_LL_at_v(v12)

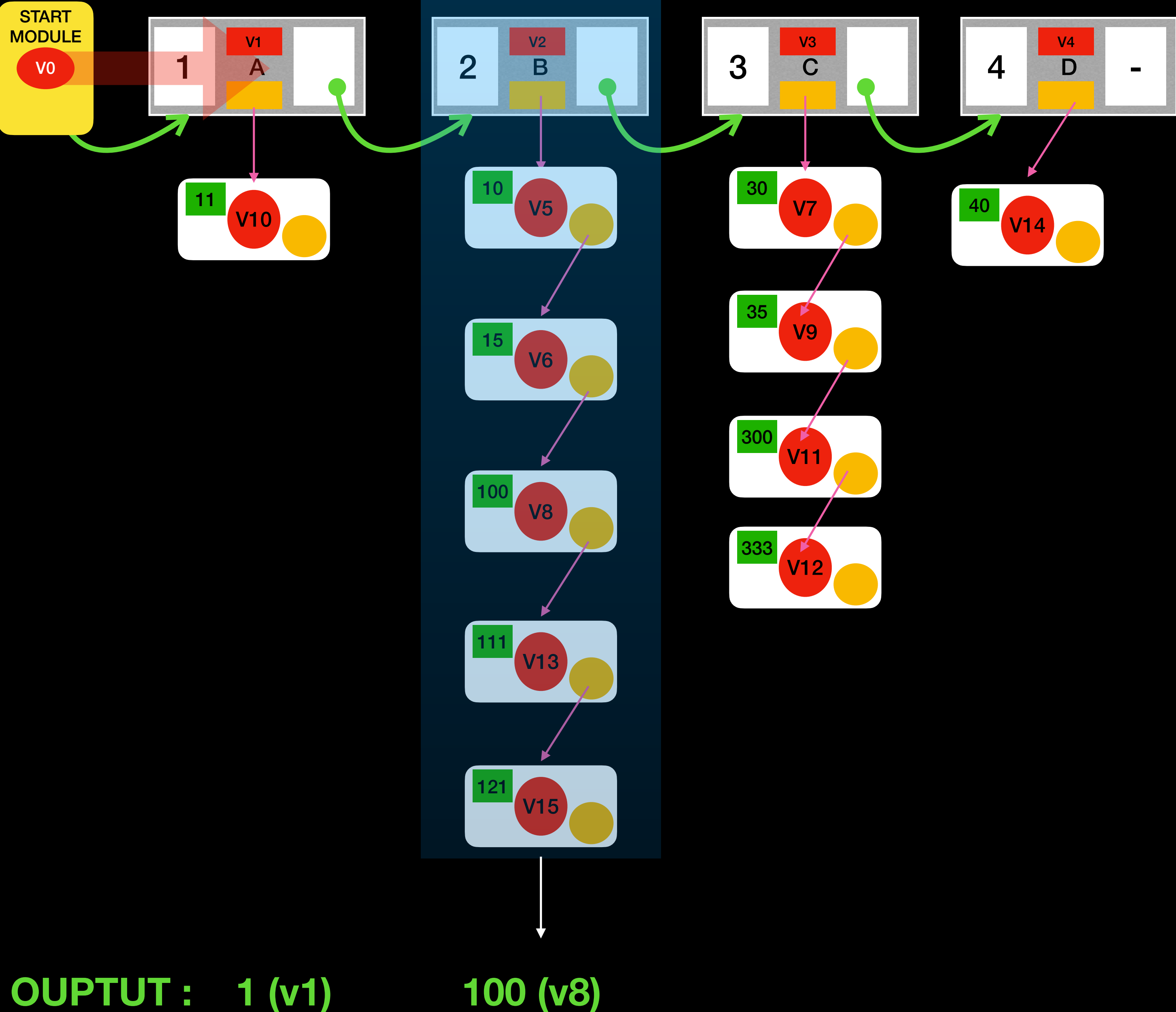


Current time, $t = 15$

Version tree

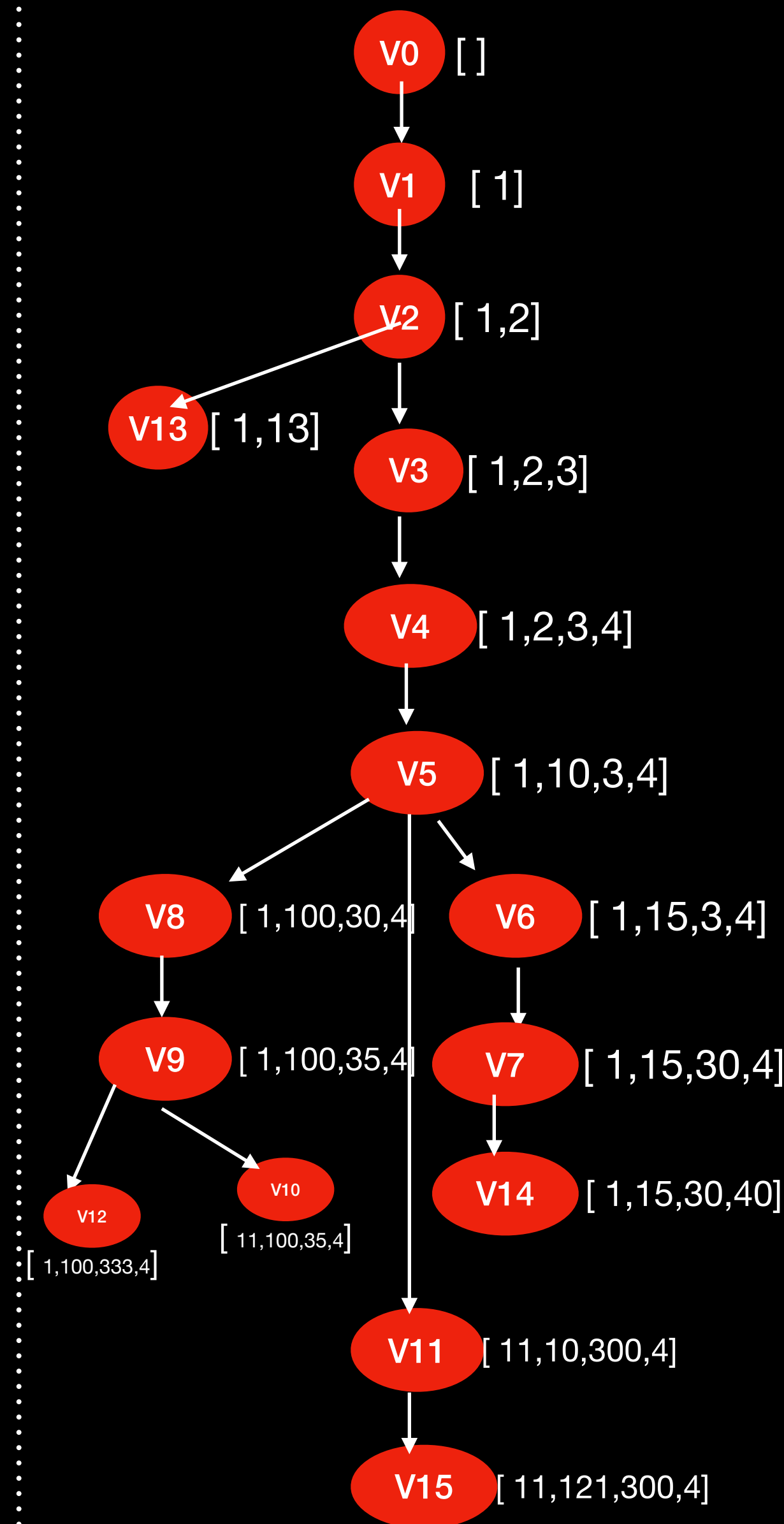


```
iterate_LL_at_v(v12)
```

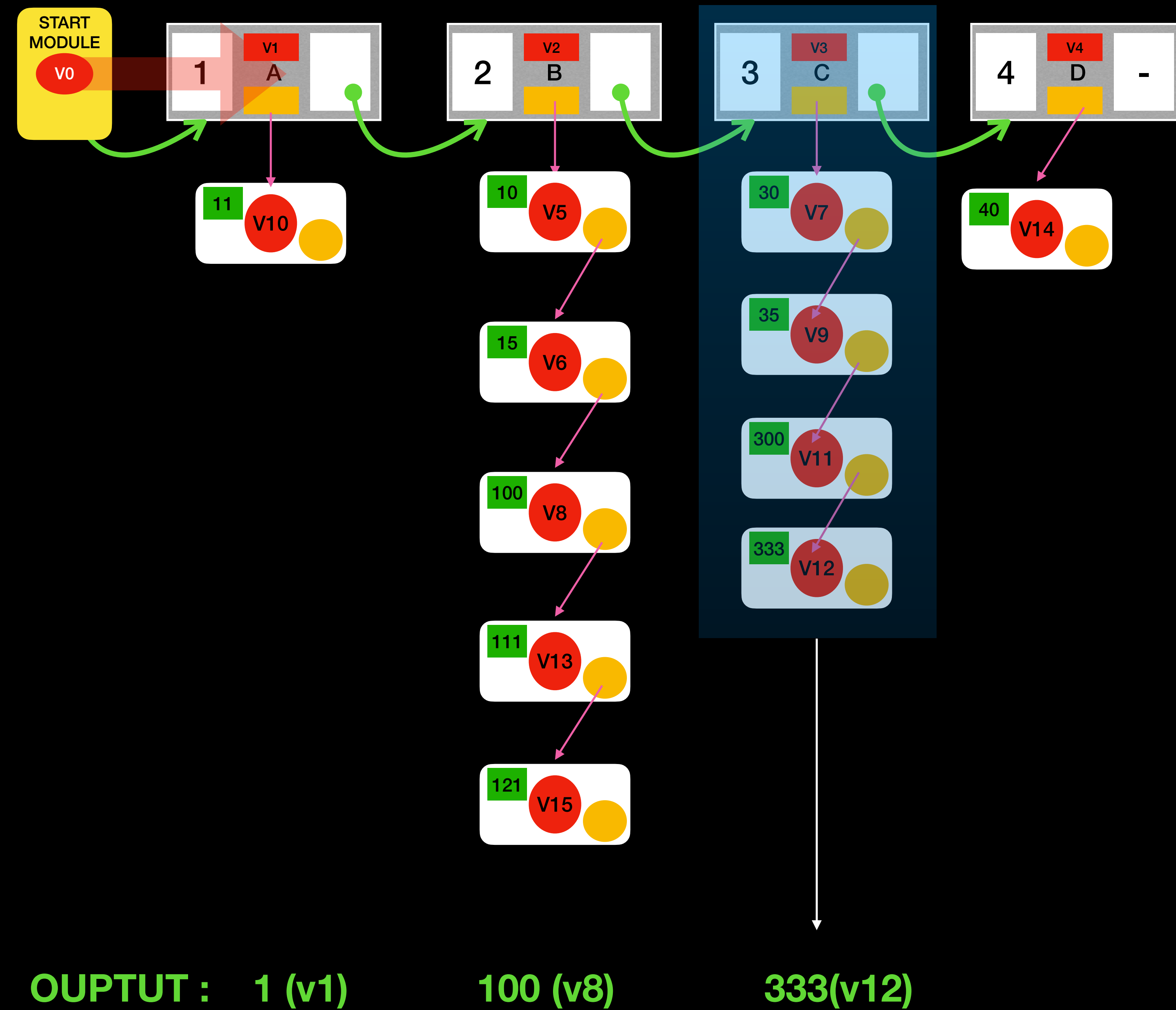


Current time, $t = 15$

Version tree

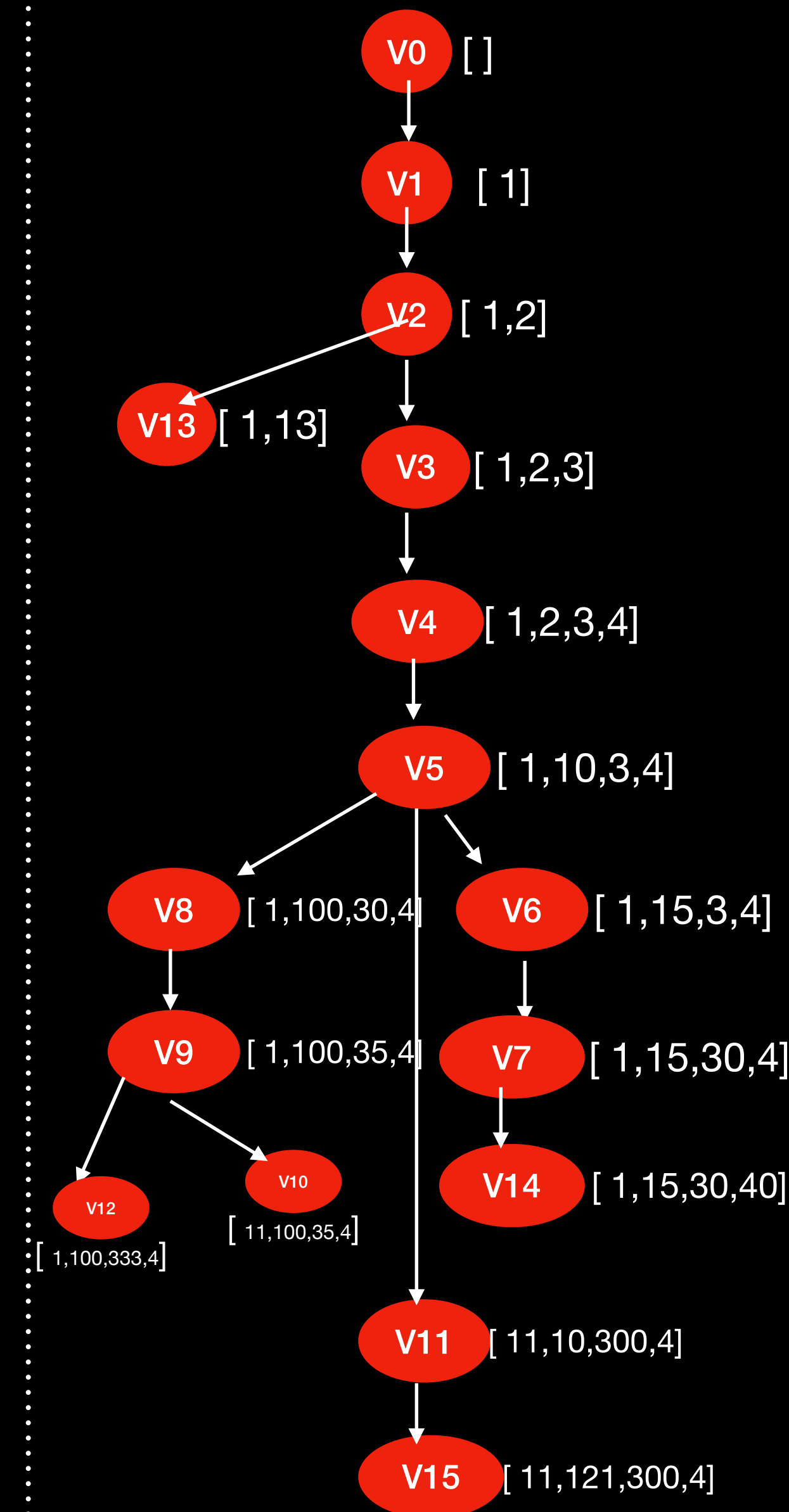


iterate_LL_at_v(v12)

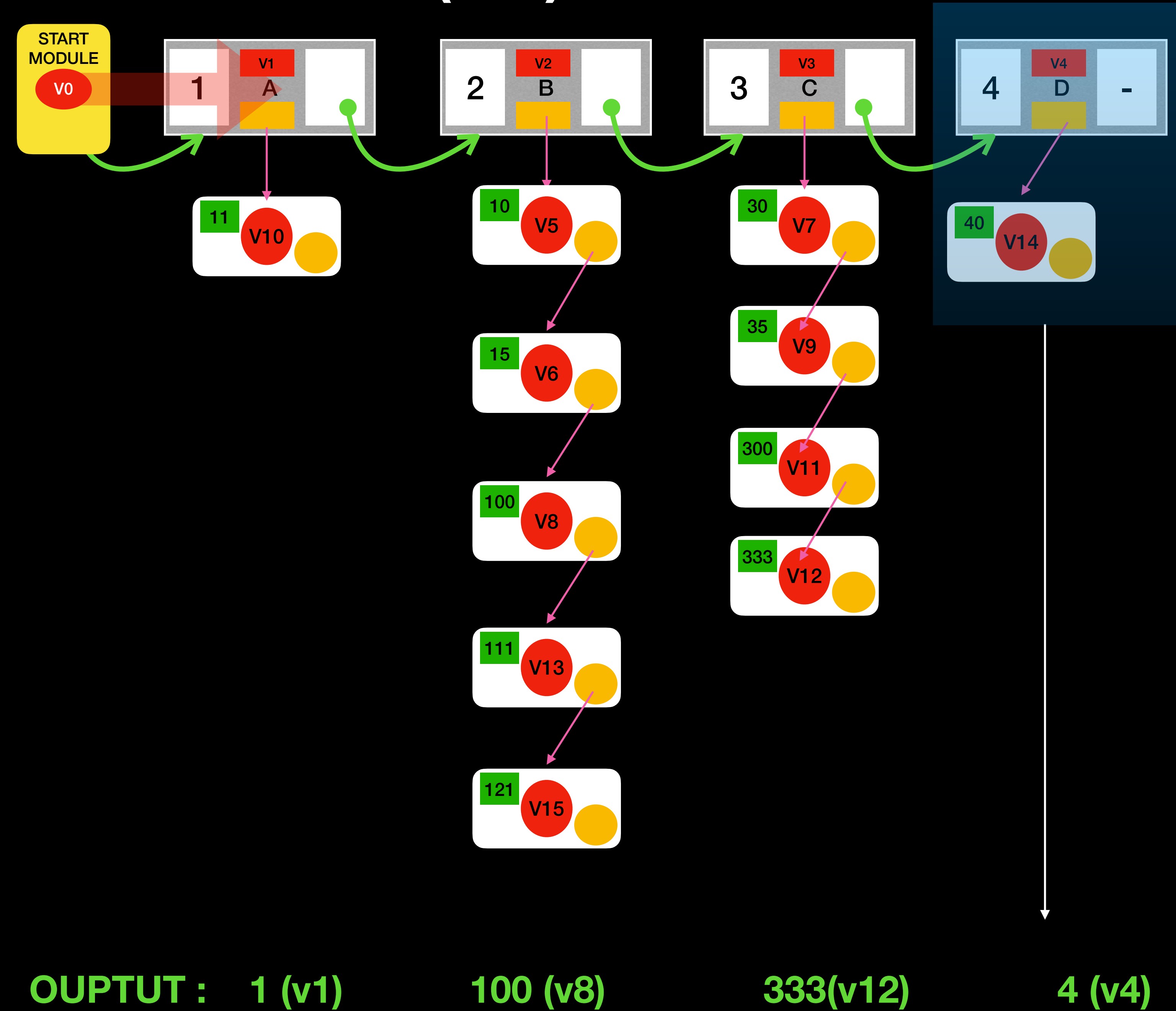


Current time, $t = 15$

Version tree

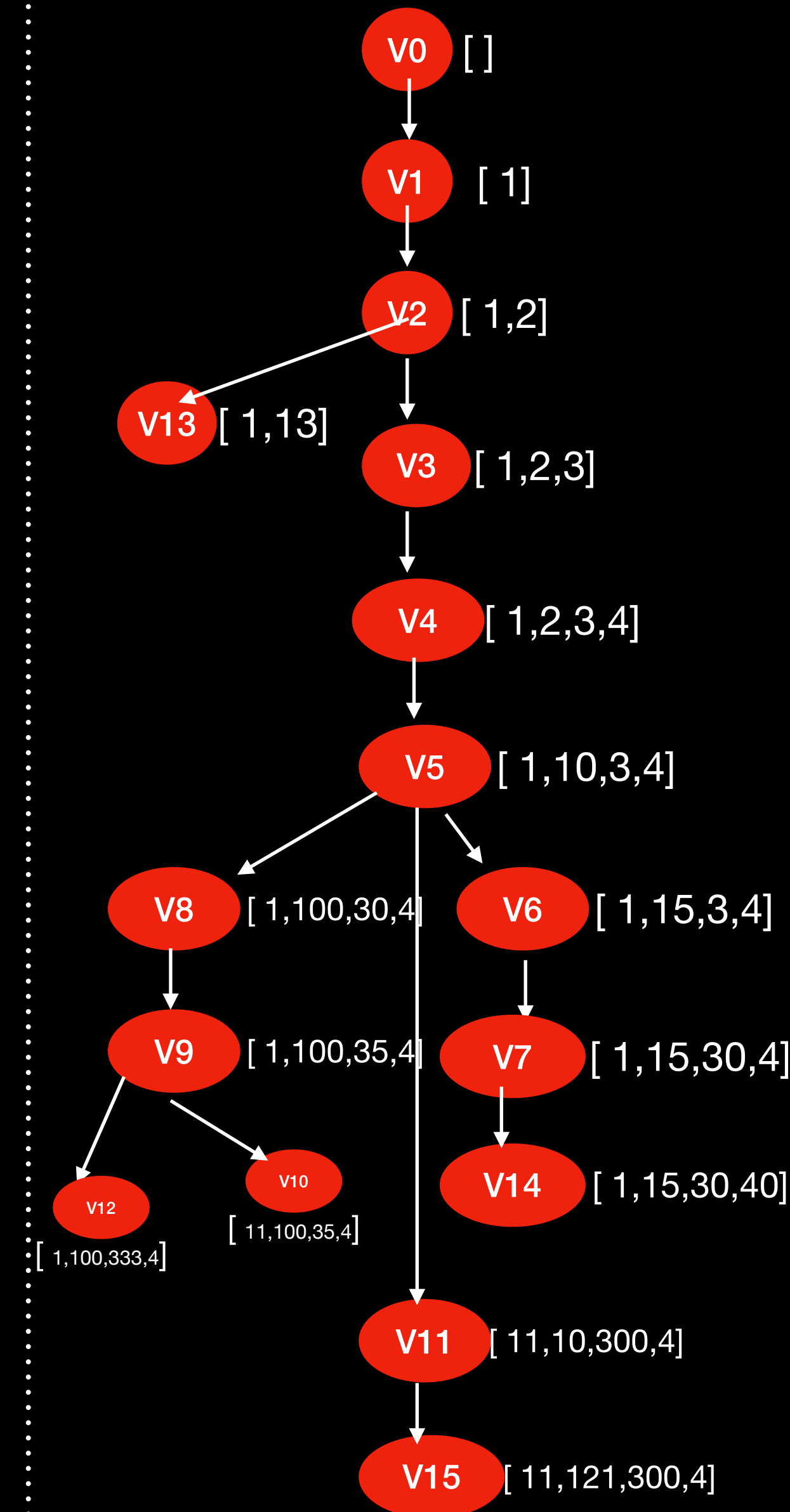


iterate_LL_at_v(v12)



Current time, $t = 15$

Version tree



Implementation Using Pointer Machine

Basic Difference In Structure in Pointer machine

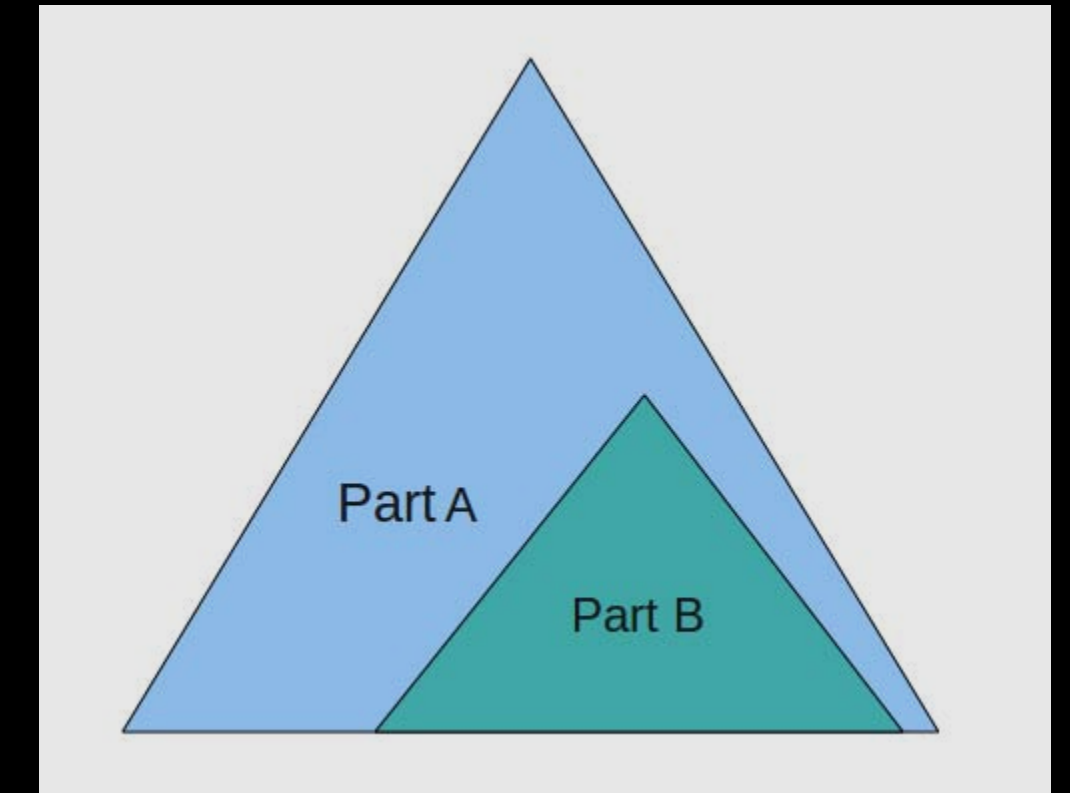
Partial Persistent PM	Full Persistent PM
Versions are Just Numbers.	Versions are reference to Nodes Of Version Maintenance Data structure (typically, V Tree)
Here we allow upto $2 * p$ modification in each node	Here we allow upto $2(d + p + 1)$ modification in each node. Additionally we now also version back-pointers.
We create a copy of the the node, when the Mod-log of a node is full, we create a copy of the node -> node' with the latest values of fields and BPs. And, Don't copy any thing to Mod_log in node'.	When the Mod-log of a node is full: Split the contents of node n's mod log into two parts. Partitioning into subtrees rather than arbitrarily is required. From the 'old' mod entries in node n, compute the latest values of each field and write them into the data and back pointer section of node m

Major Difference

We split the mod-log of older node into 1:1 or 2:1 partition

Transfer the 50% or 33% recent mods to the newly created node

Set the the fields of newly created node, according to the latest values from the previous 50% or 66% Mods left in the older node.



Why?

To reduce the pressure on a node of particular version.

As, we can modify a particular node of a specific version multiple number of time in Full Persistent Strategy.

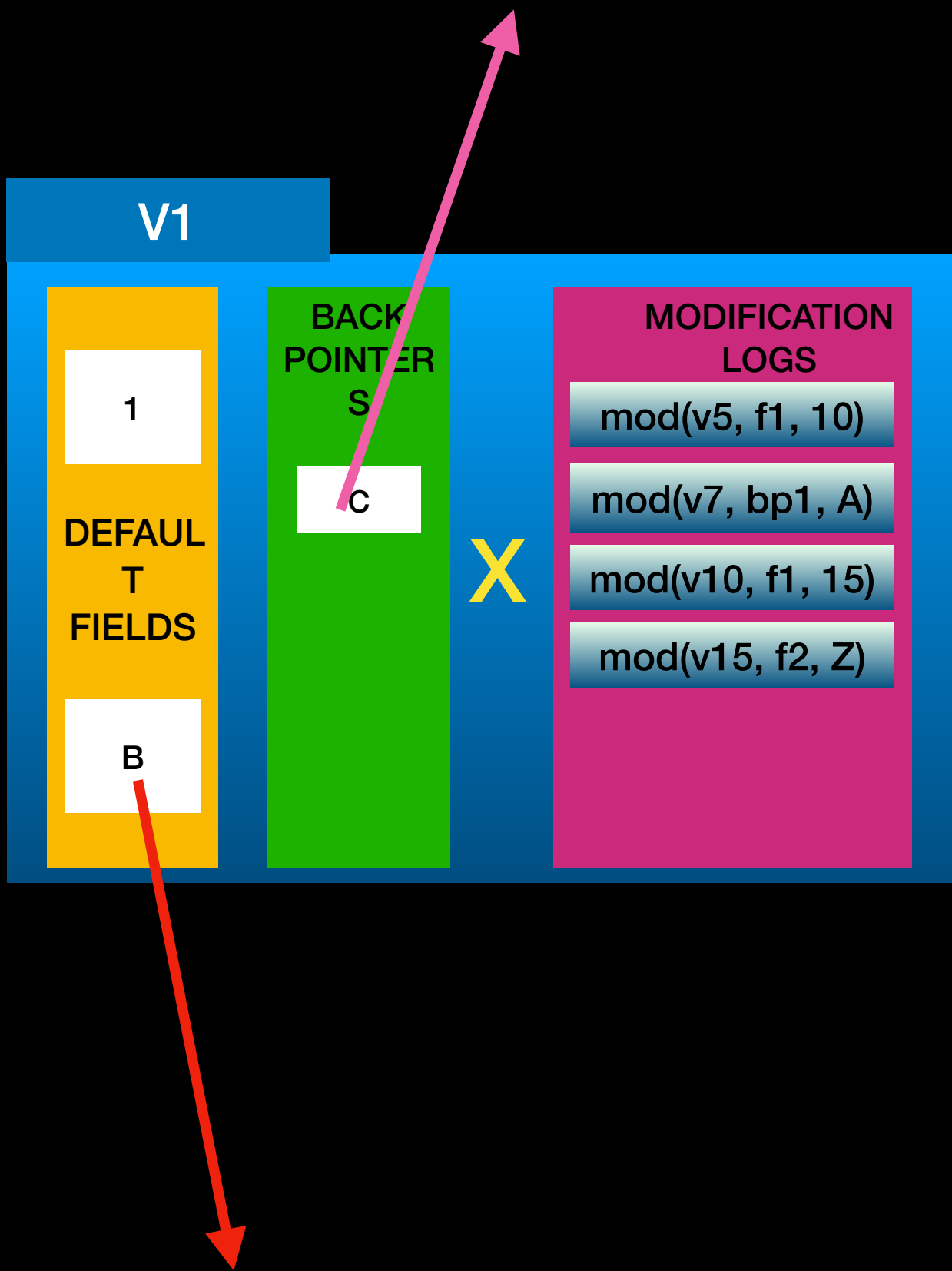
Bad Way (We just create copy with 1:0 split):

A particular node of a specific version with Full Mod-Log will create a empty-mod copy how many times we tried to modify that specific version.

Good Way (We just create copy with 1:1 / 2:1 split):

That particular node of a specific version with Full Mod-Log is now not Full anymore it has 50% or 33% space in Mod-log free to keep the upcoming modification.

Problem with 1:0 Approach (What was in Partial Persistent Mode)

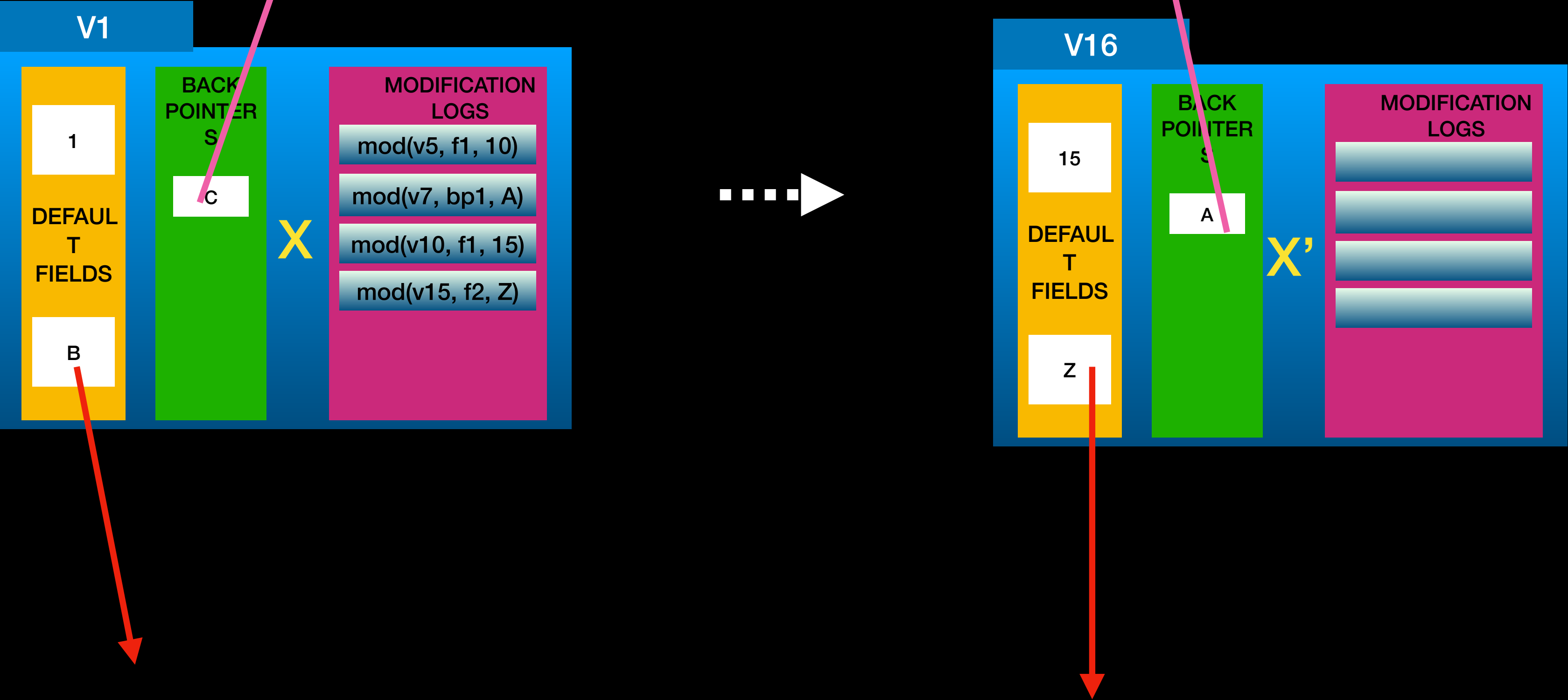


Suppose Node in any version (cur t >=15) looks like this

NOTE: THIS IS NOT DONE IN FULL PERSISTENT STRATEGY

Problem with 1:0 Approach (What was in Partial Persistent Mode)

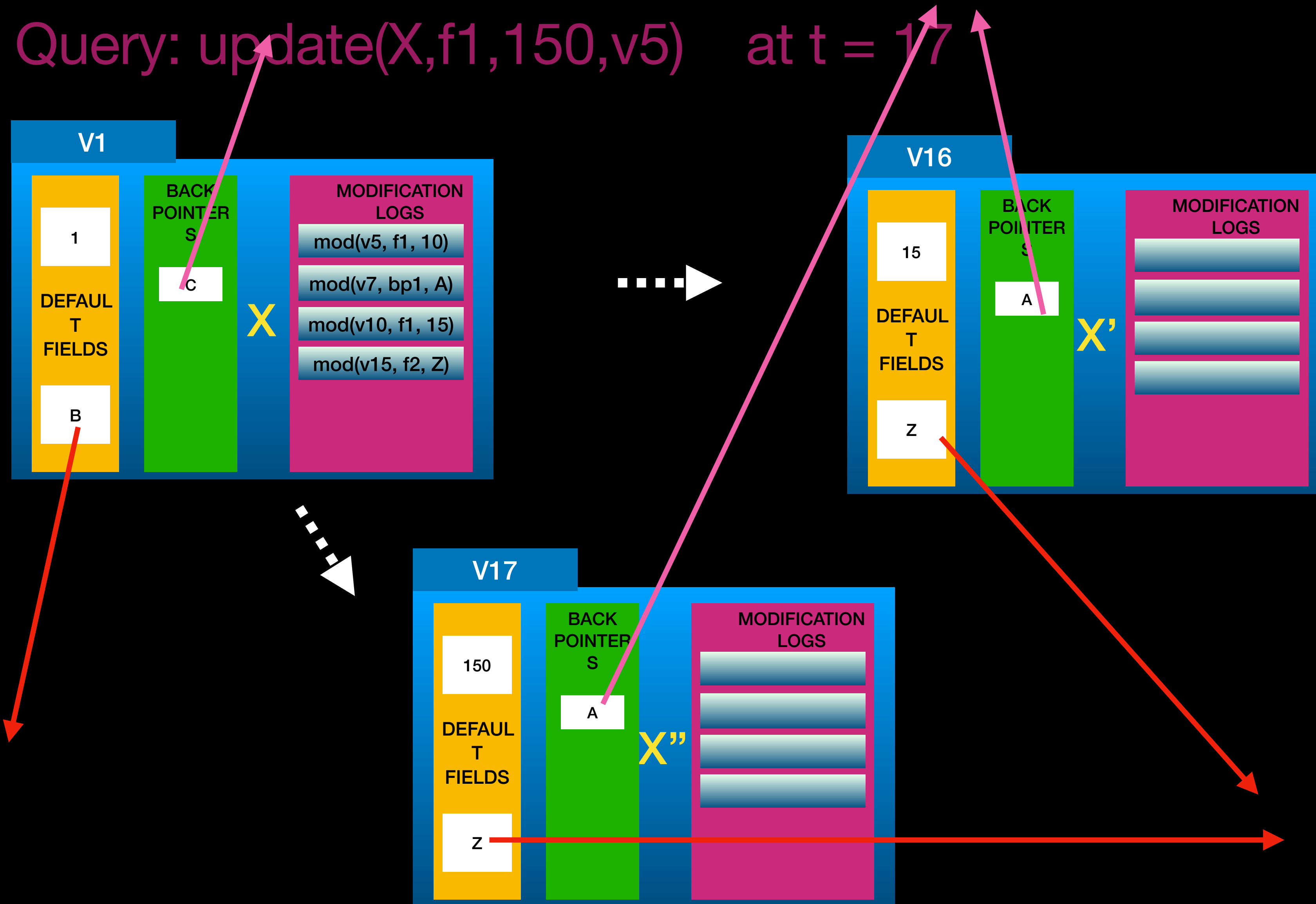
Query: update(X,f1,15,v5) at t = 16



NOTE: THIS IS NOT DONE IN FULL PERSISTENT STRATEGY

Problem with 1:0 Approach (What was in Partial Persistent Mode)

Query: update(X,f1,150,v5) at t = 17



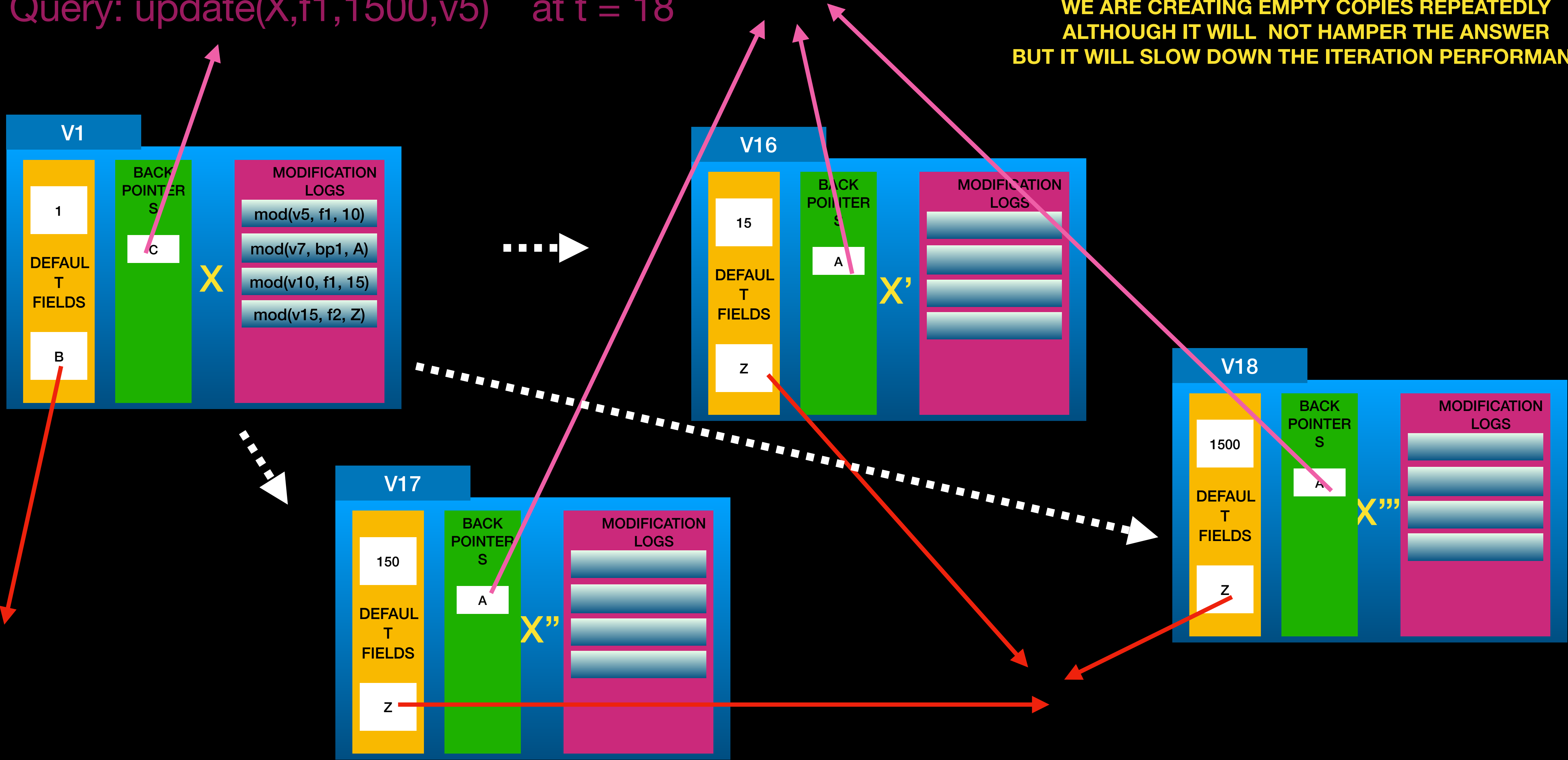
NOTE: THIS IS NOT DONE IN FULL PERSISTENT STRATEGY

Problem with 1:0 Approach (What was in Partial Persistent Mode)

Query: `update(X,f1,1500,v5)` at $t = 18$

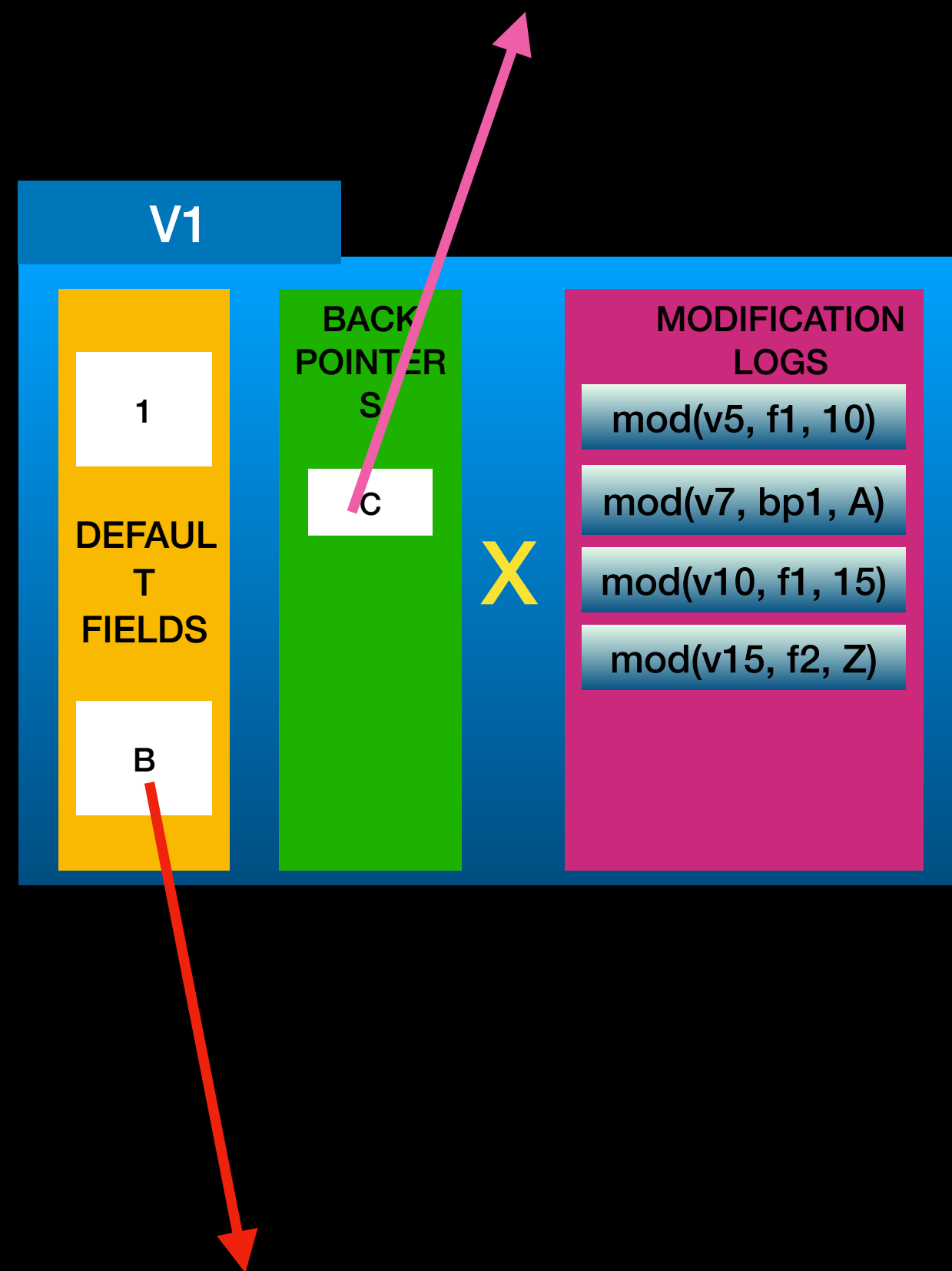
CONCERN:

WE ARE CREATING EMPTY COPIES REPEATEDLY
ALTHOUGH IT WILL NOT HAMPER THE ANSWER
BUT IT WILL SLOW DOWN THE ITERATION PERFORMANCE



NOTE: THIS IS NOT DONE IN FULL PERSISTENT STRATEGY

OPTIMISATION with 1:1 OR 2:1 Approach



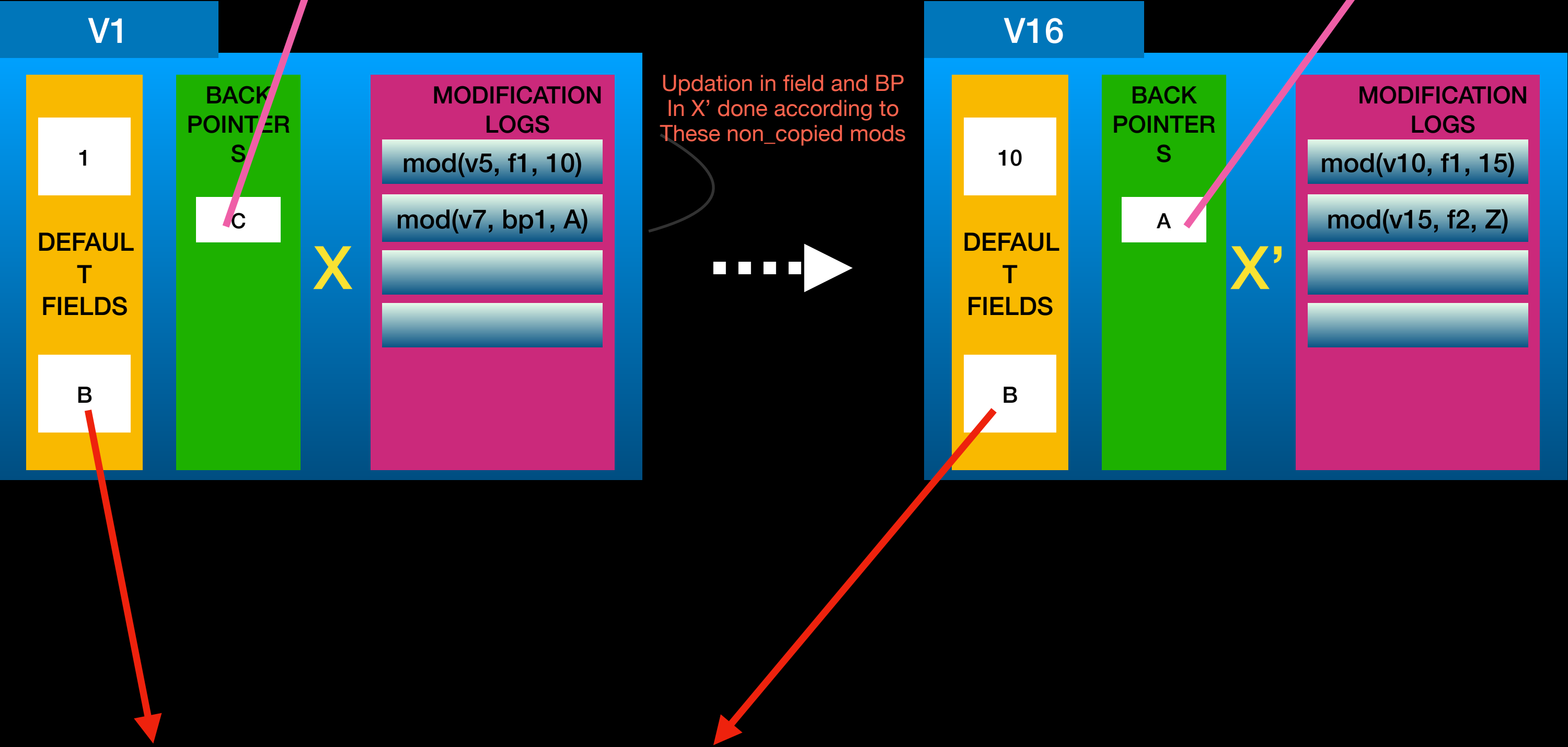
Suppose Node in any version ($current \geq 15$) looks like this

WE ARE GOING TO PRESENT 1:1 SPLIT

NOTE: THIS IS DONE IN FULL PERSISTENT STRATEGY

OPTIMISATION with 1:1 Approach

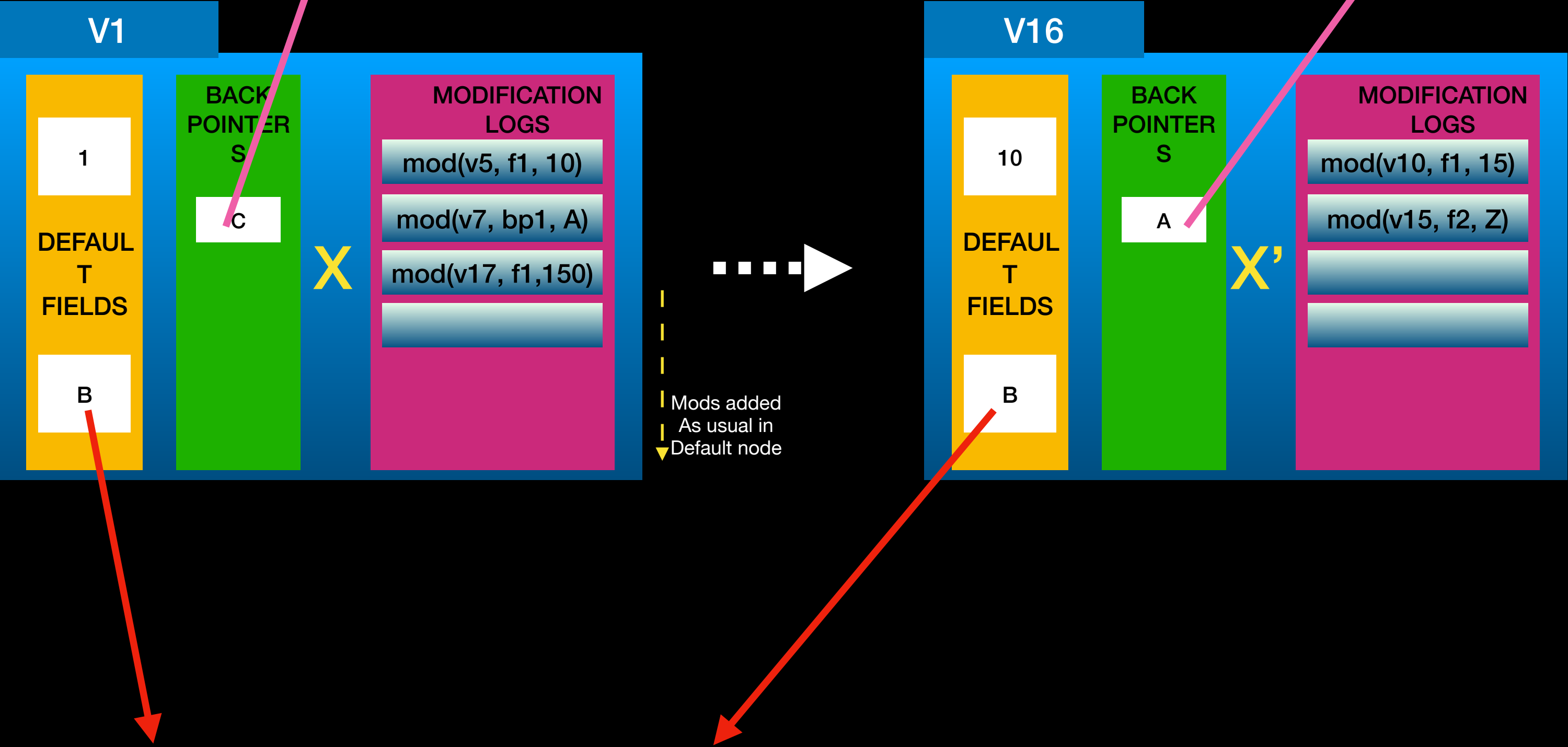
Query: `update(X,f1,15,v5)` at $t = 16$



NOTE: THIS IS DONE IN FULL PERSISTENT STRATEGY

OPTIMISATION with 1:1 Approach

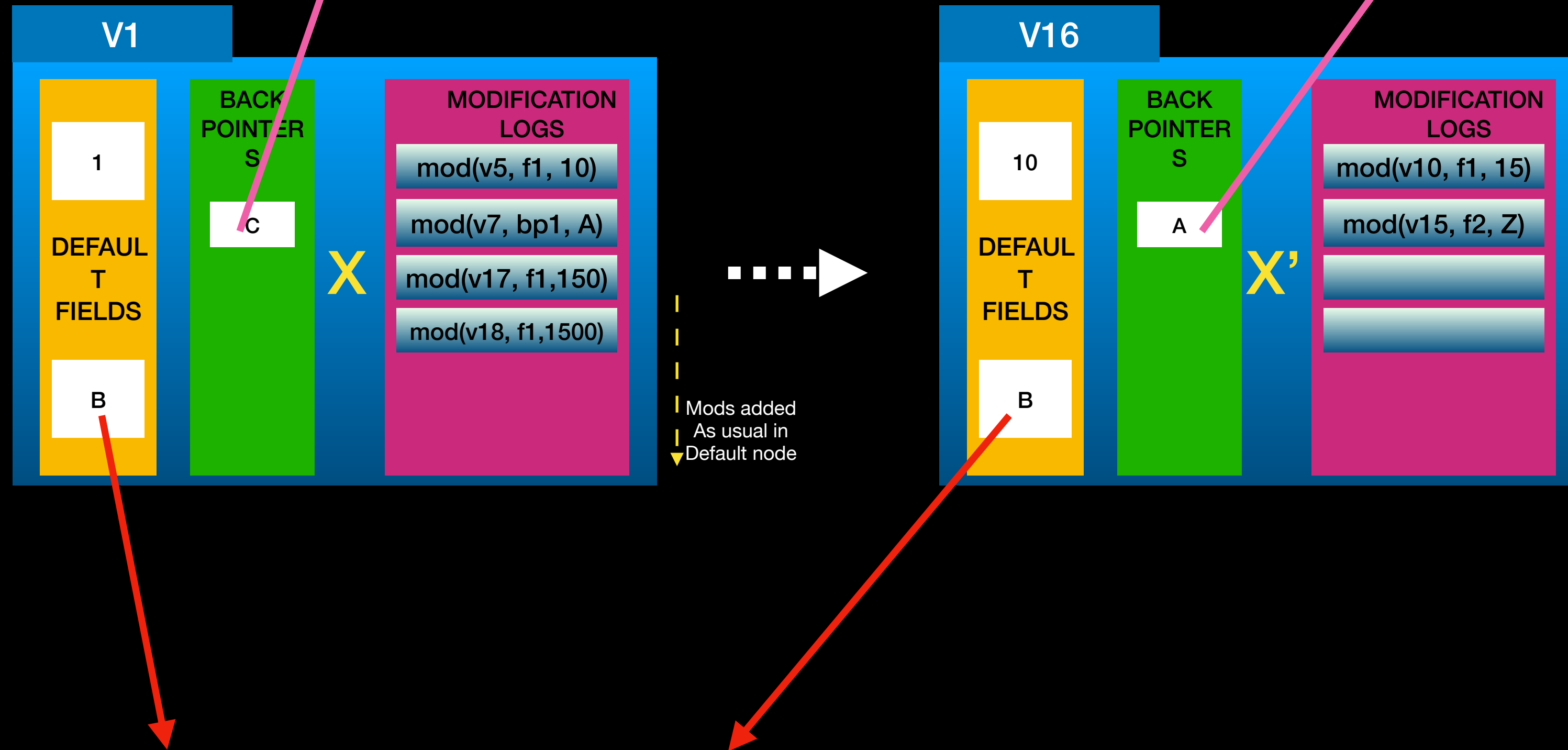
Query: update(X,f1,150,v5) at t = 17



NOTE: THIS IS DONE IN FULL PERSISTENT STRATEGY

OPTIMISATION with 1:1 Approach

Query: update(X,f1,1500,v5) at t = 18



CONCERN:

HERE WE ARE NOT CREATING SUCCESSIVE EMPTY_MOD NODE-copies

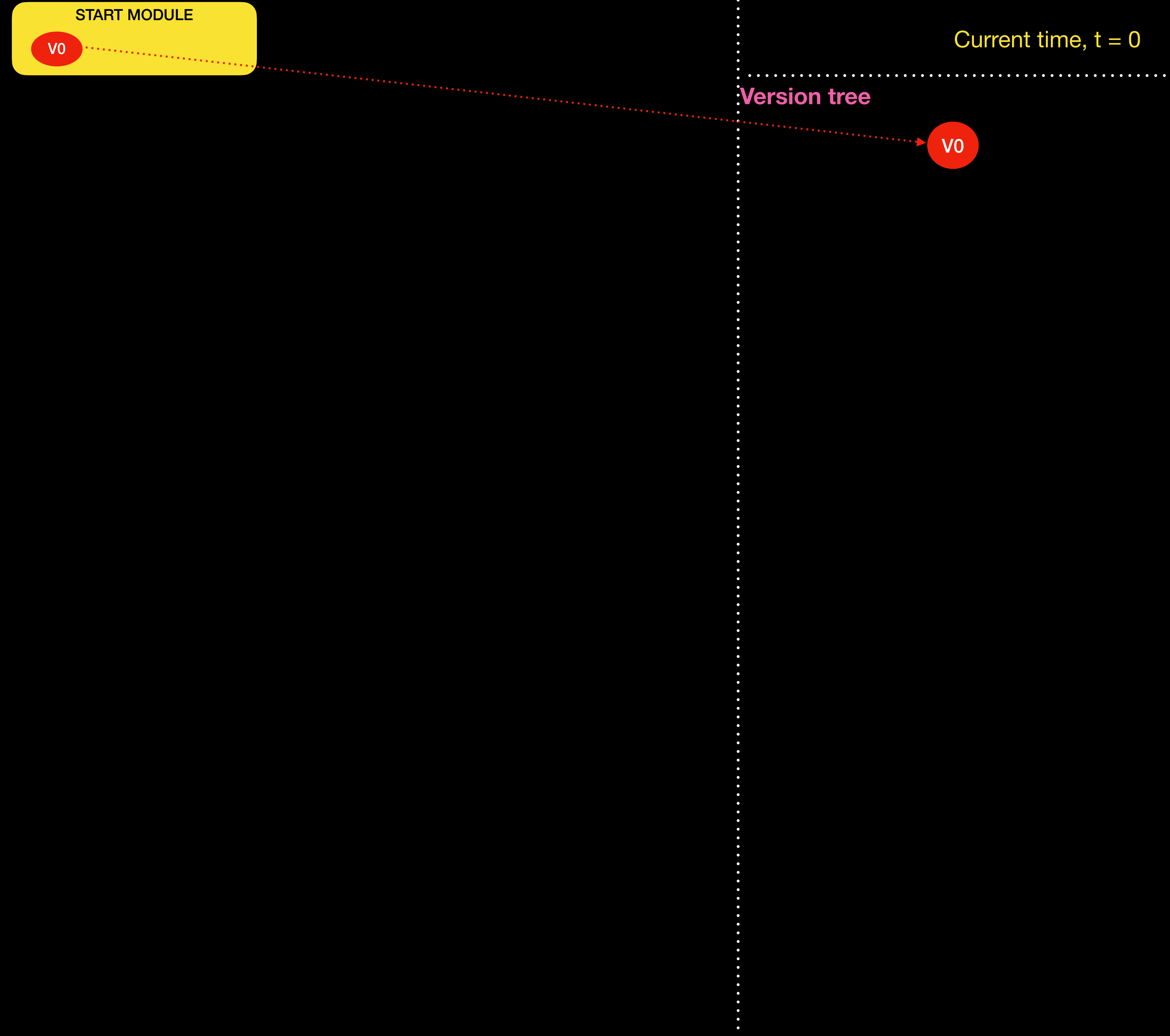


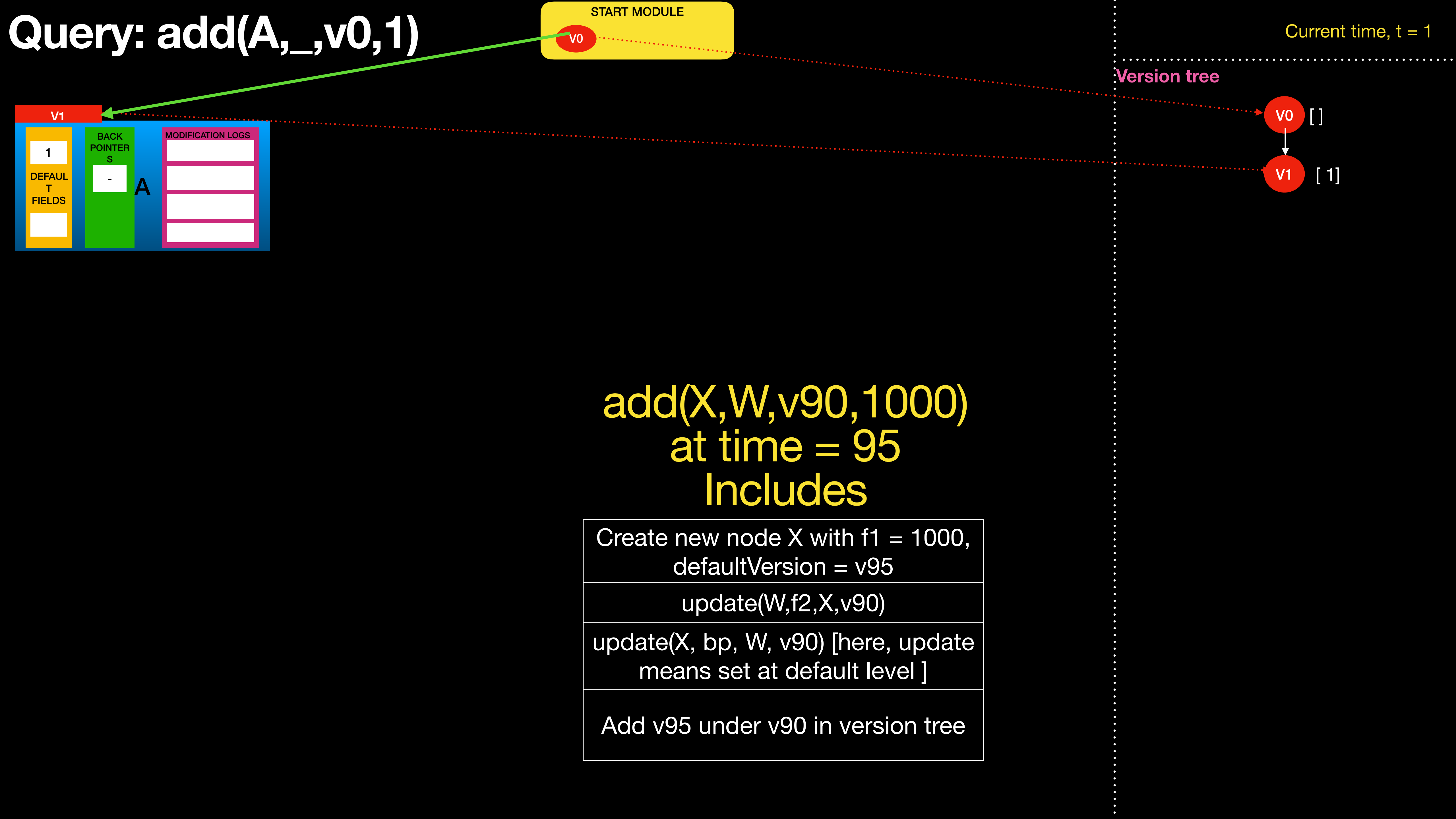
NOTE: THIS IS DONE IN FULL PERSISTENT STRATEGY

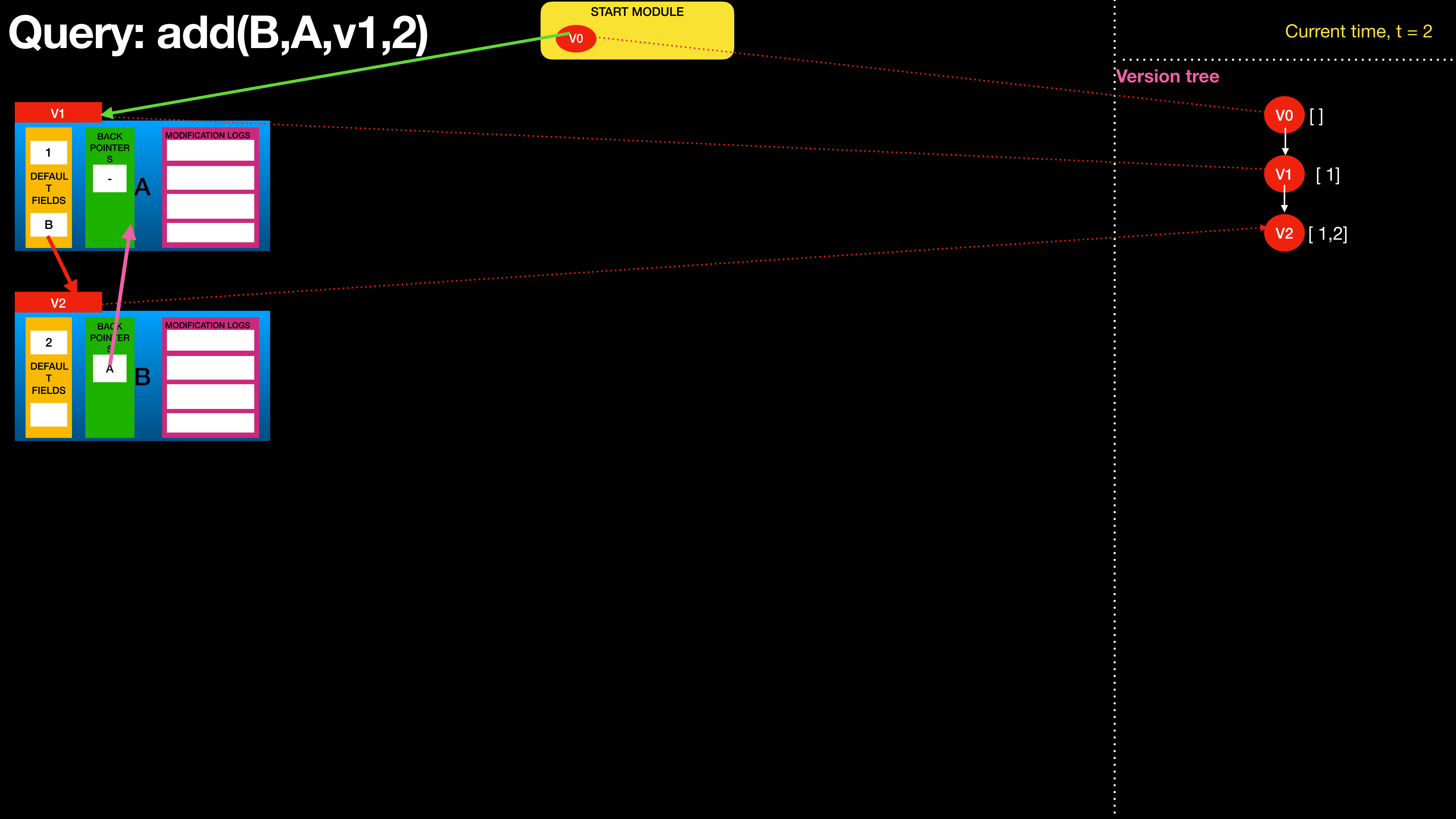
Simulation Using Full Persistent Model

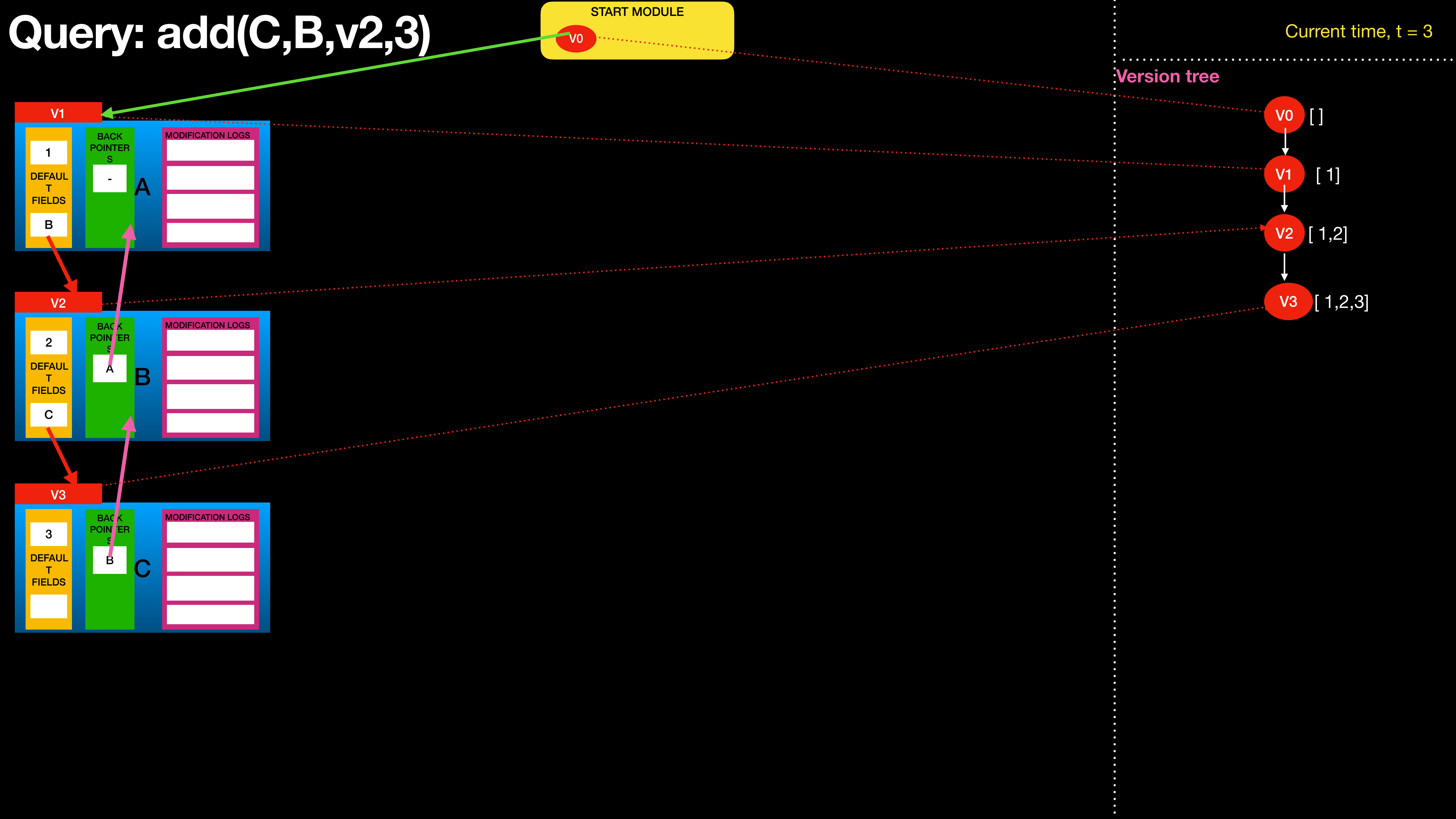


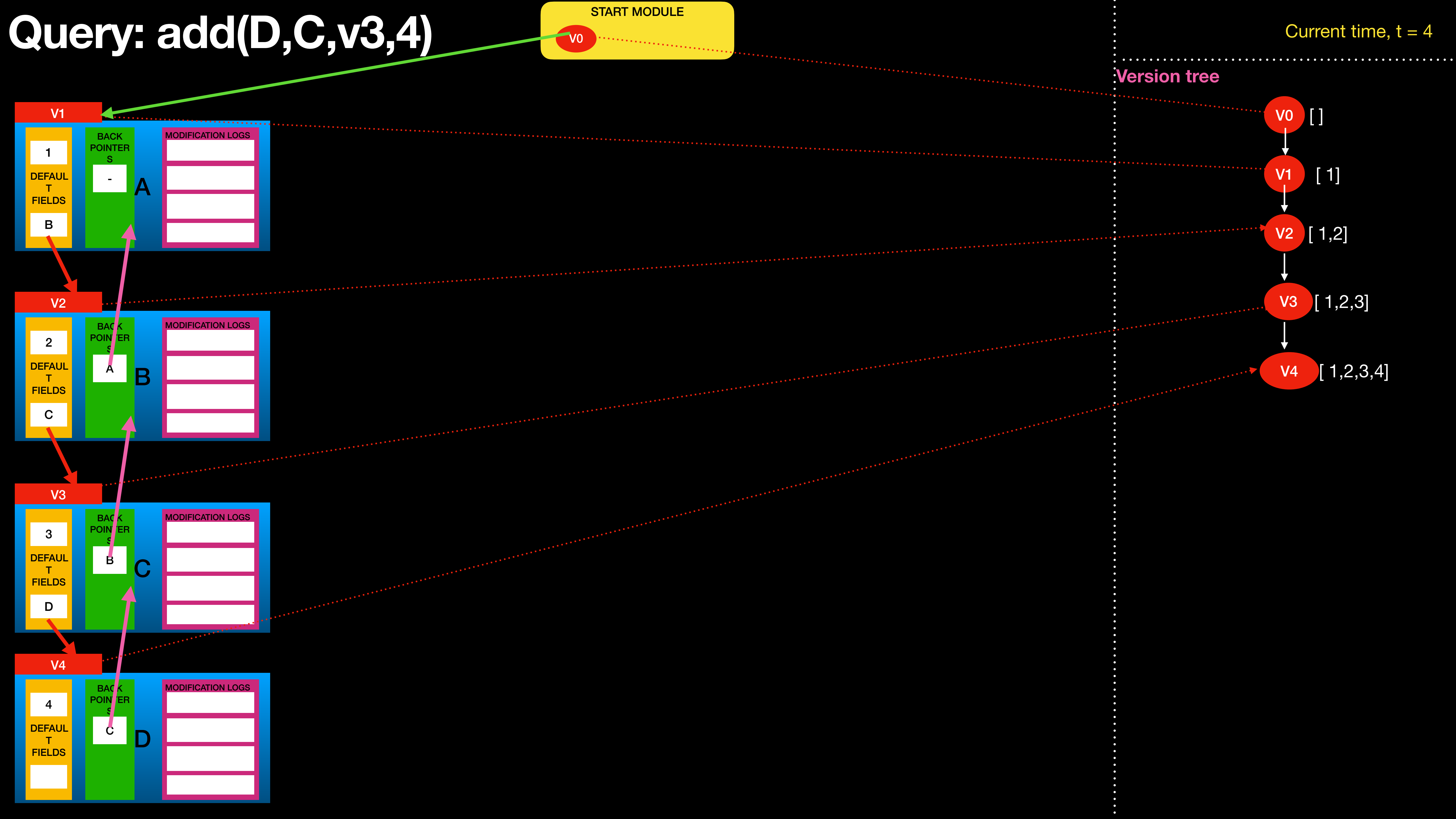
Query: init_LL()

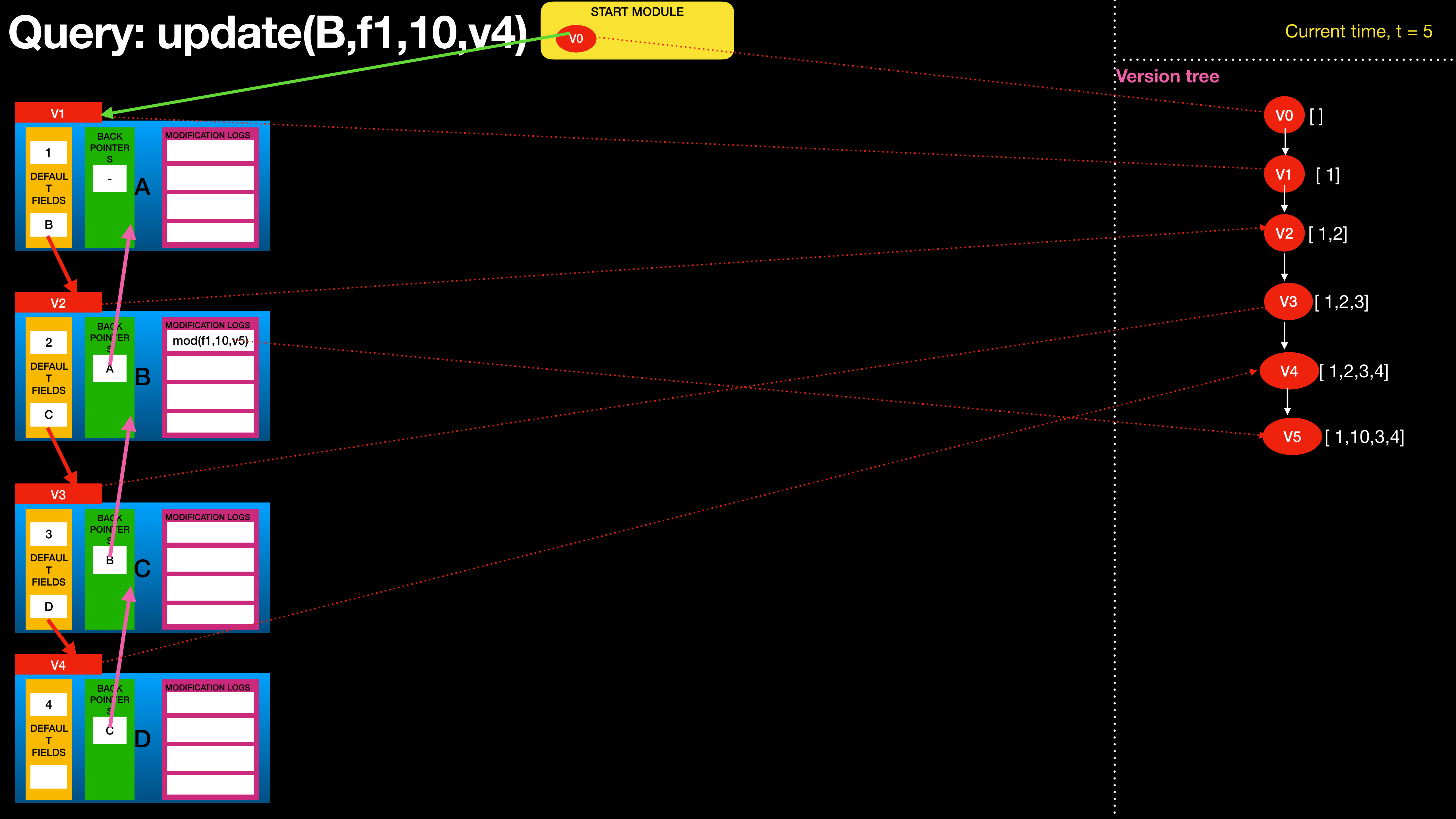


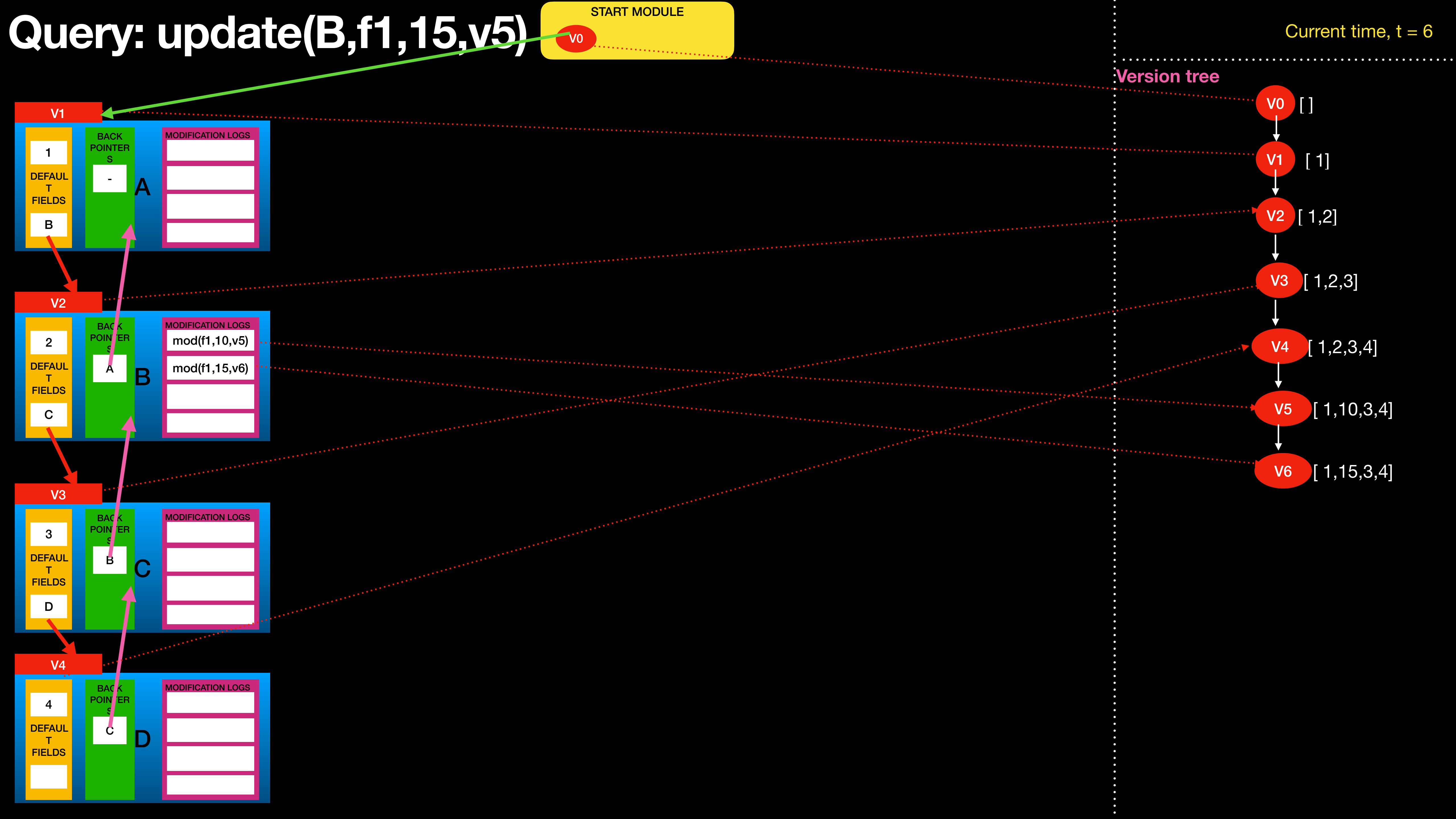


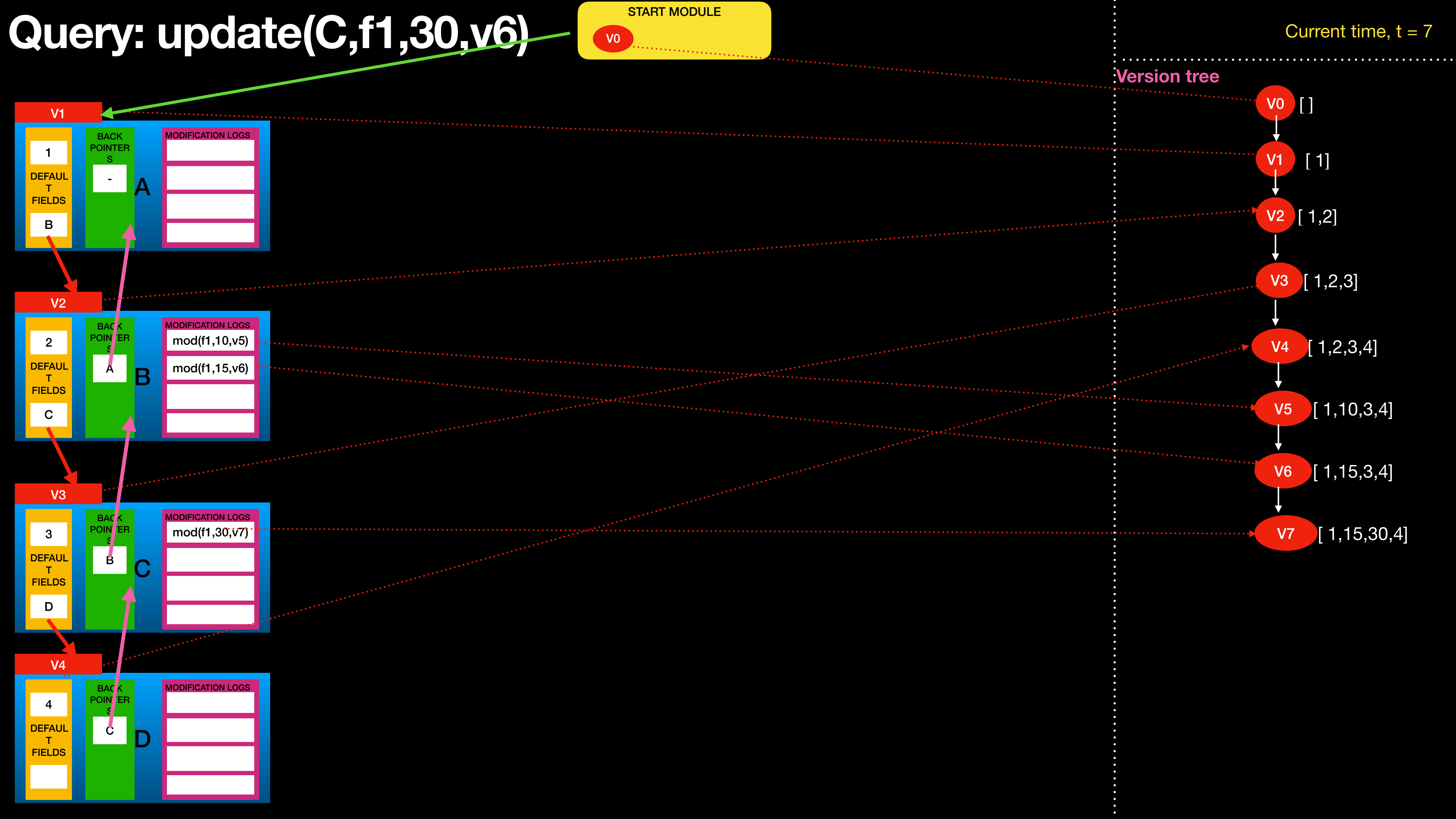




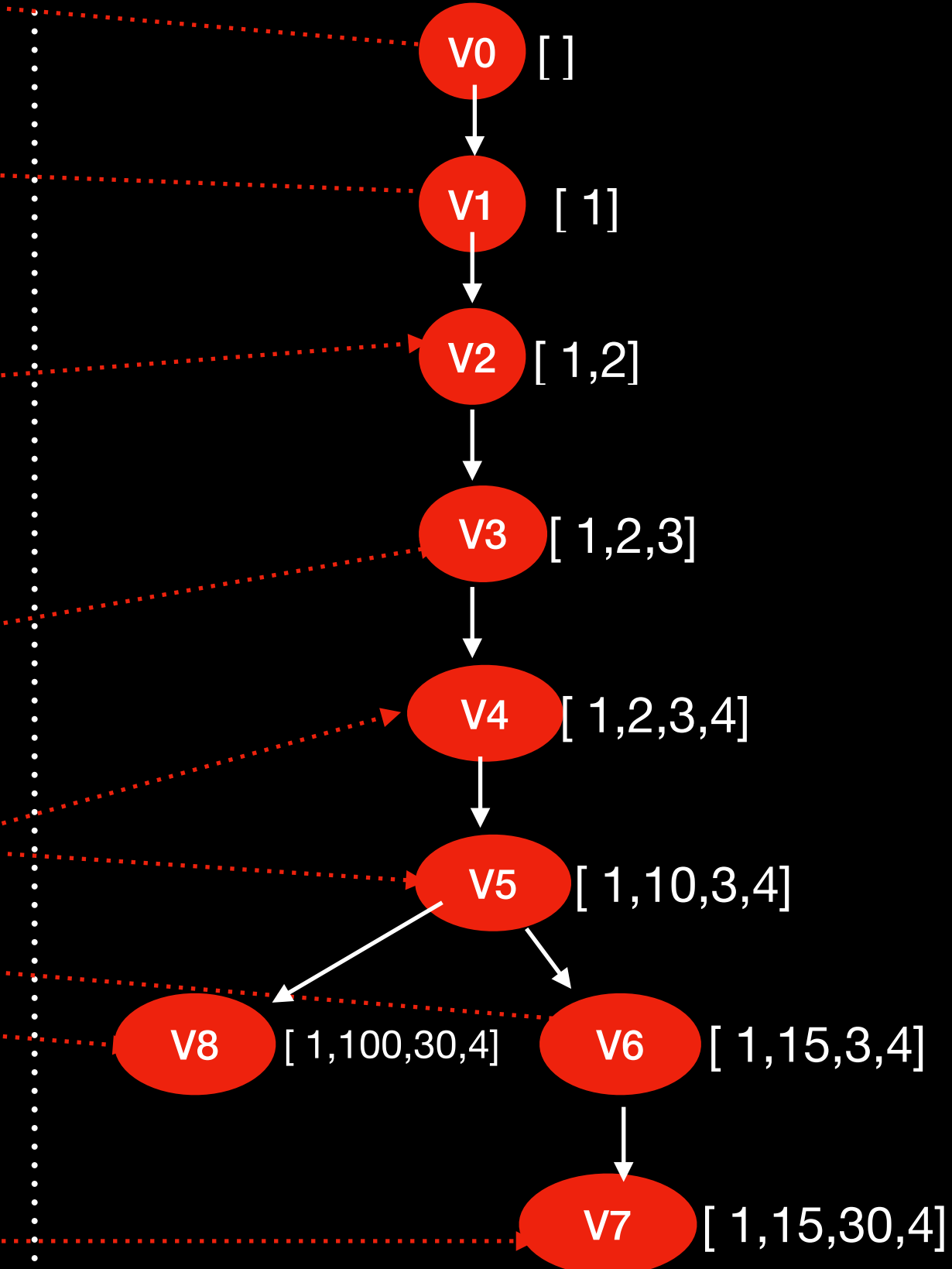








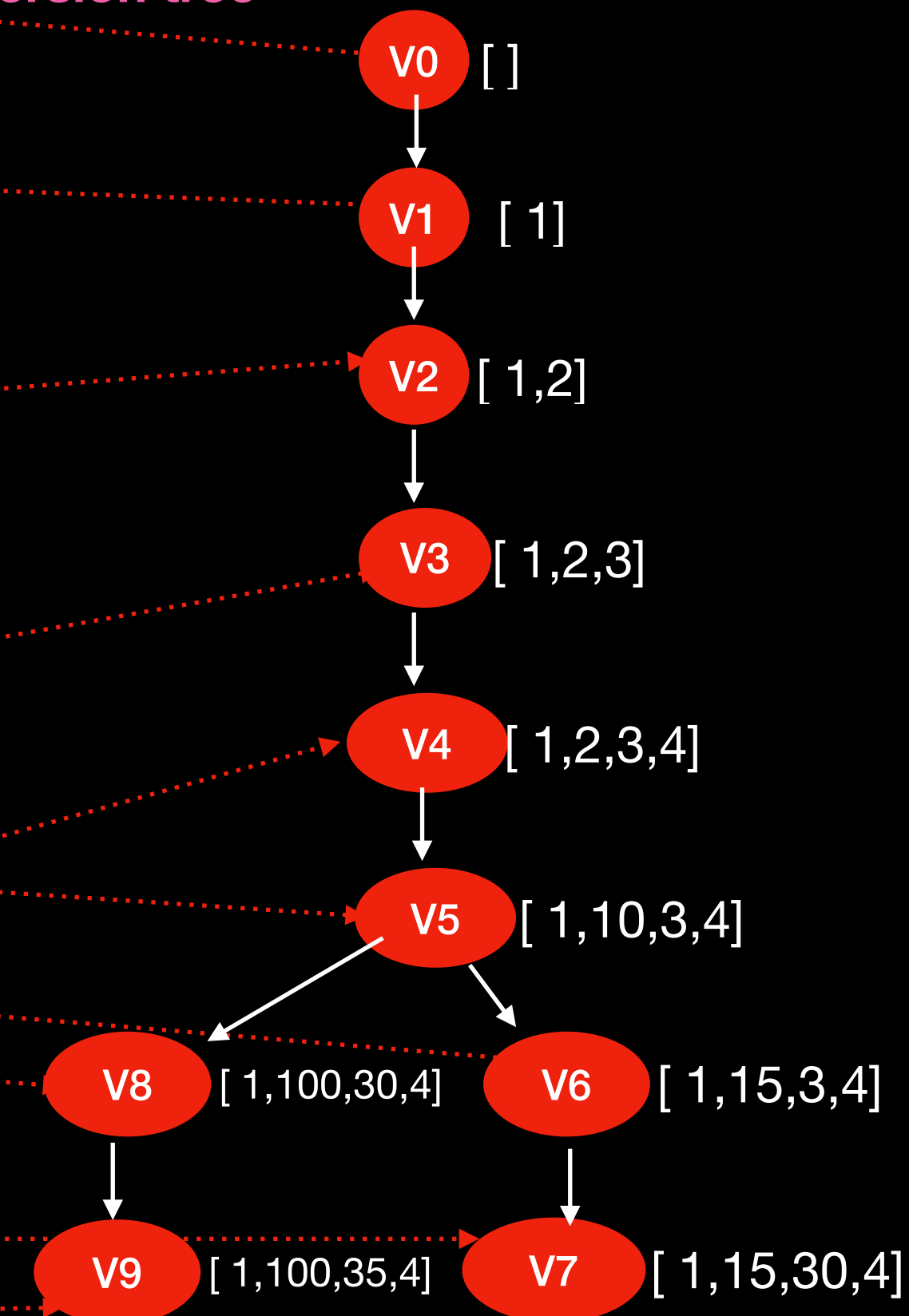
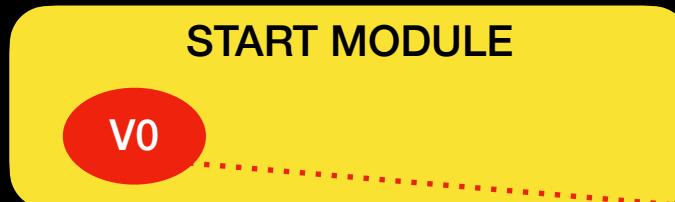
Current time, $t = 8$

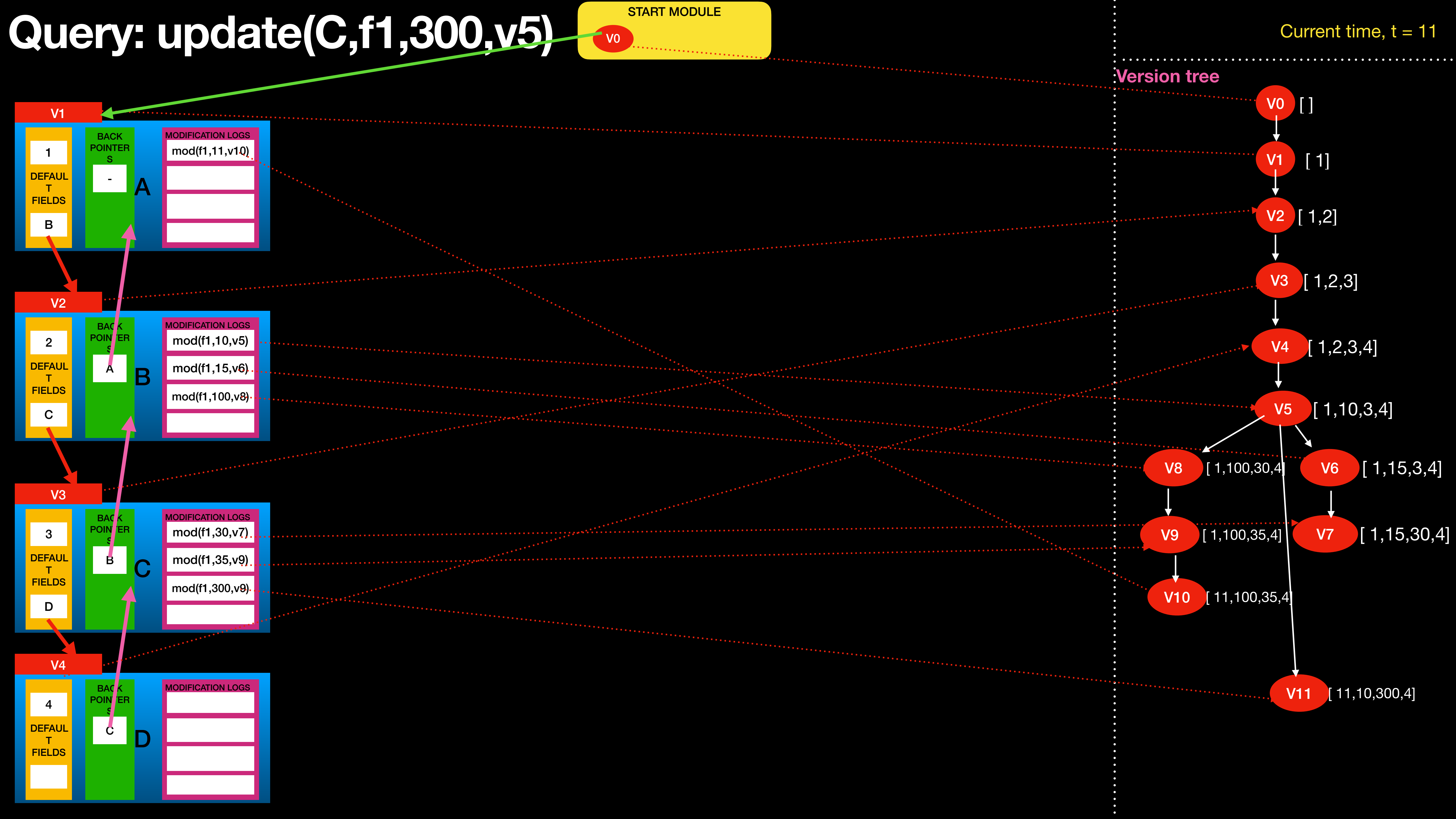


START MODULE

V0

Version tree

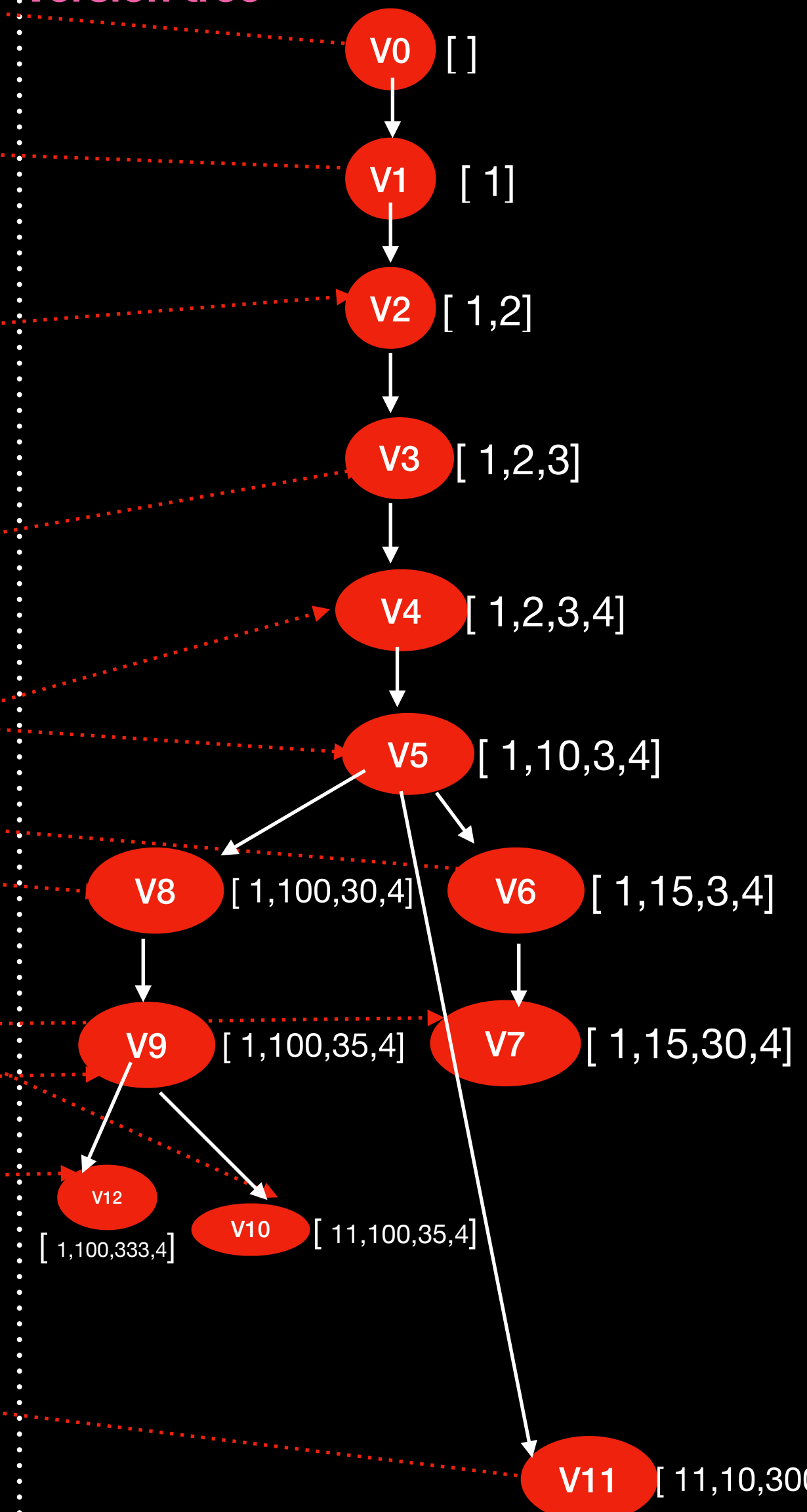


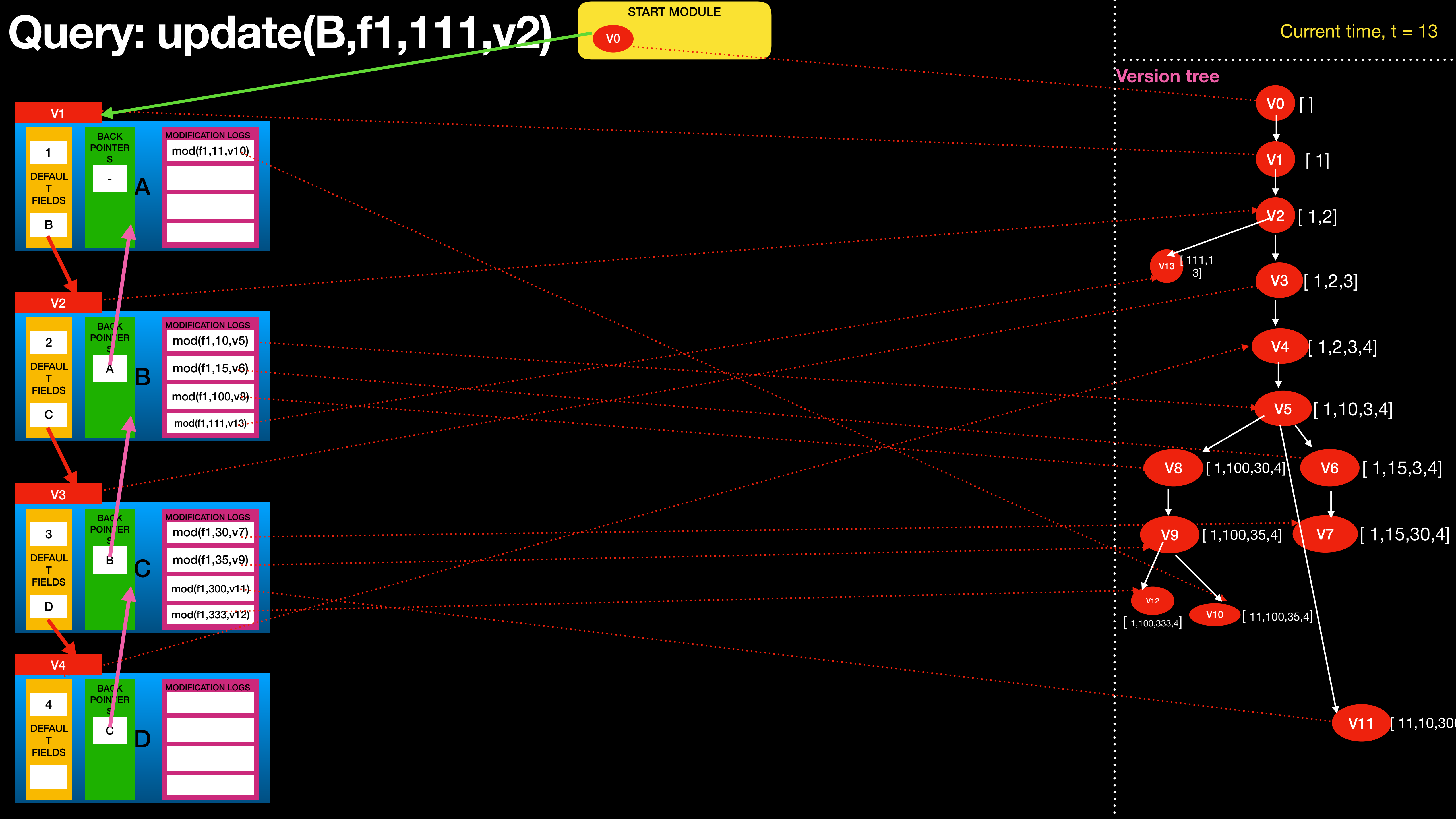


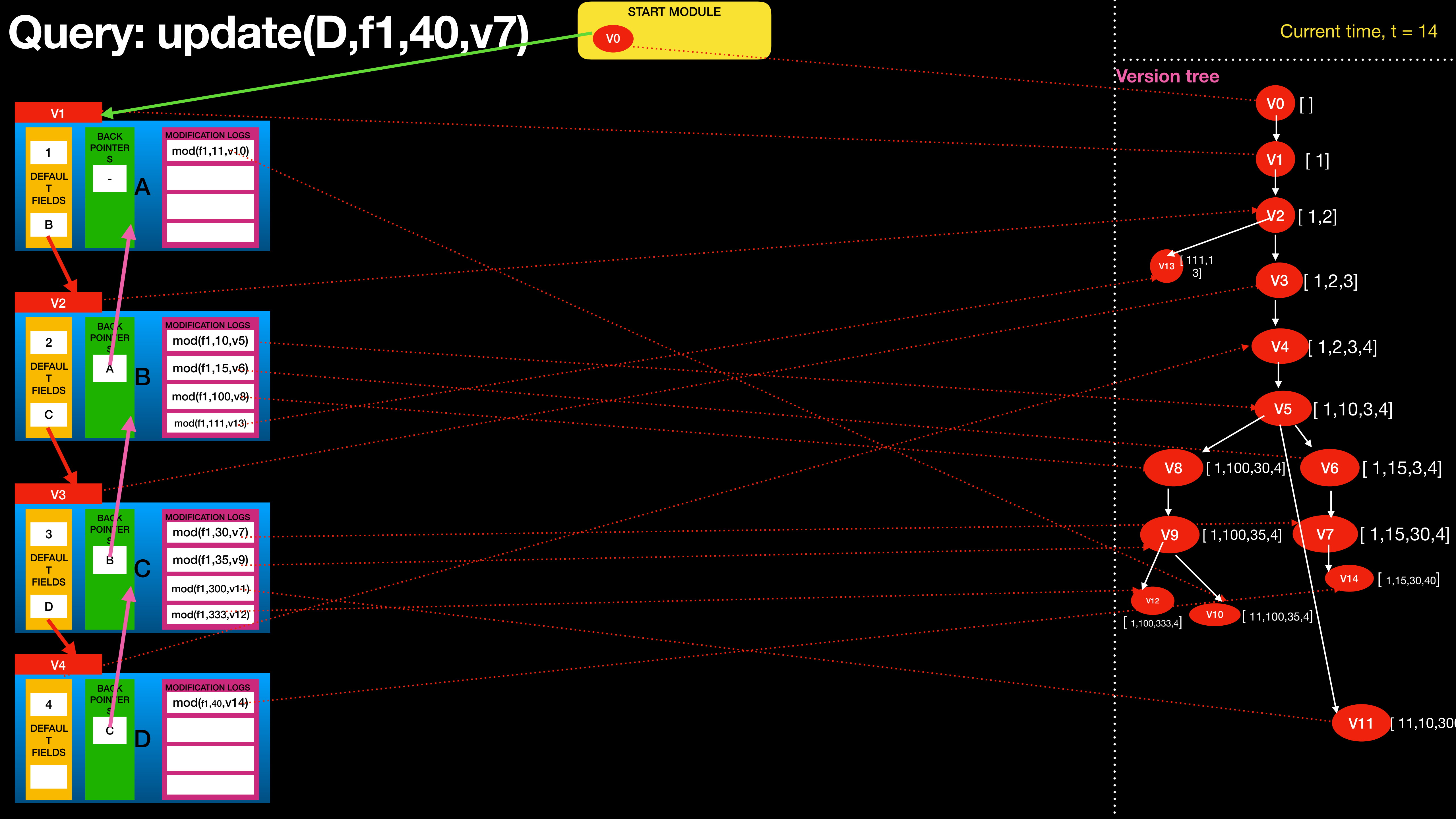
START MODULE

V0

Version tree







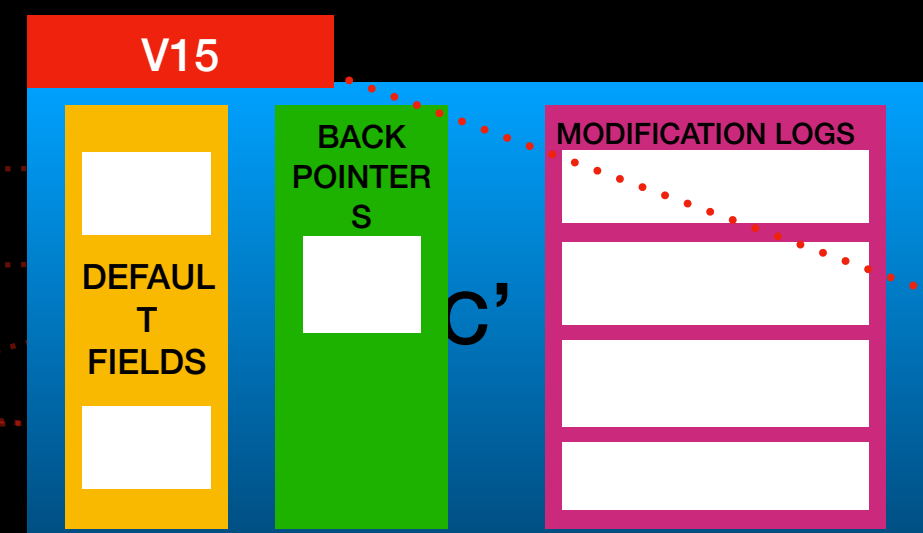
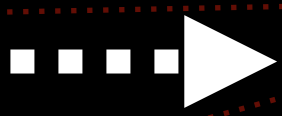
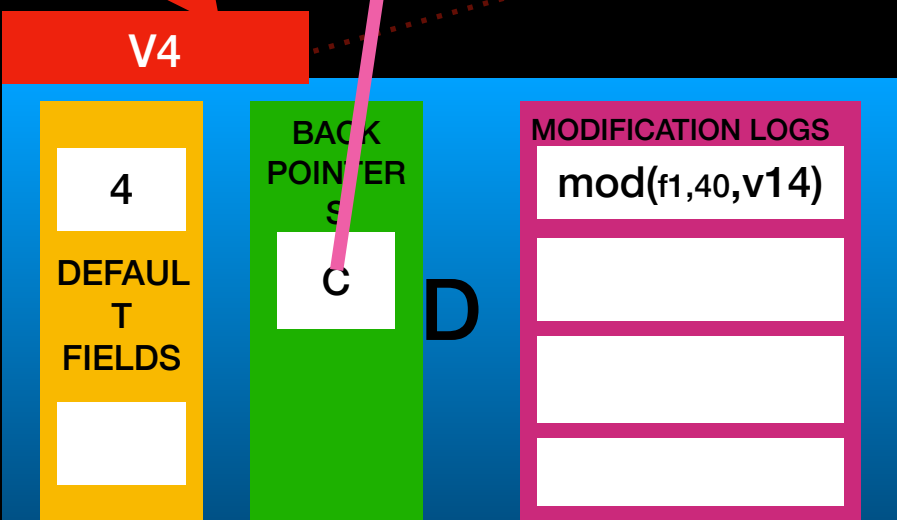
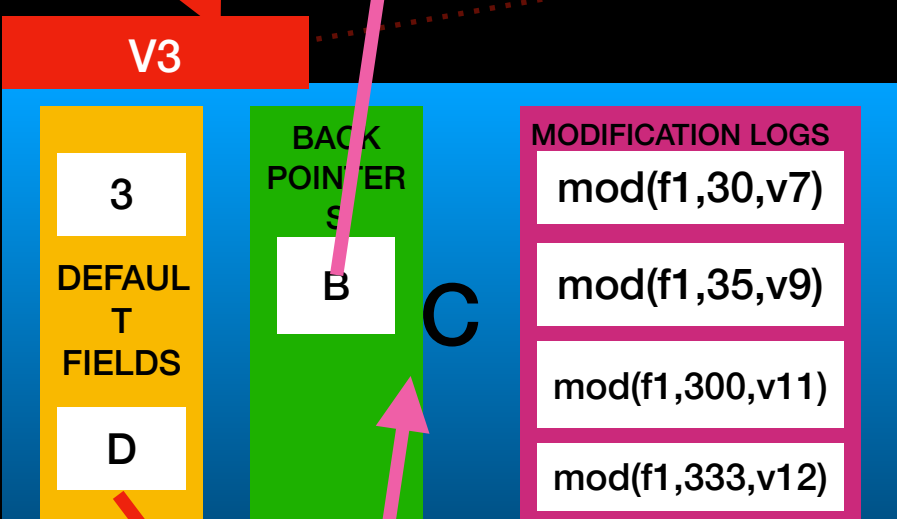
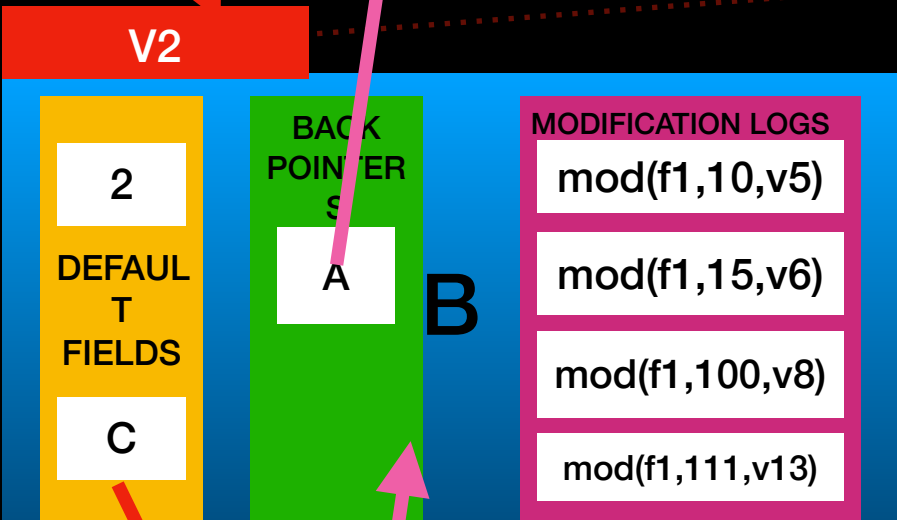
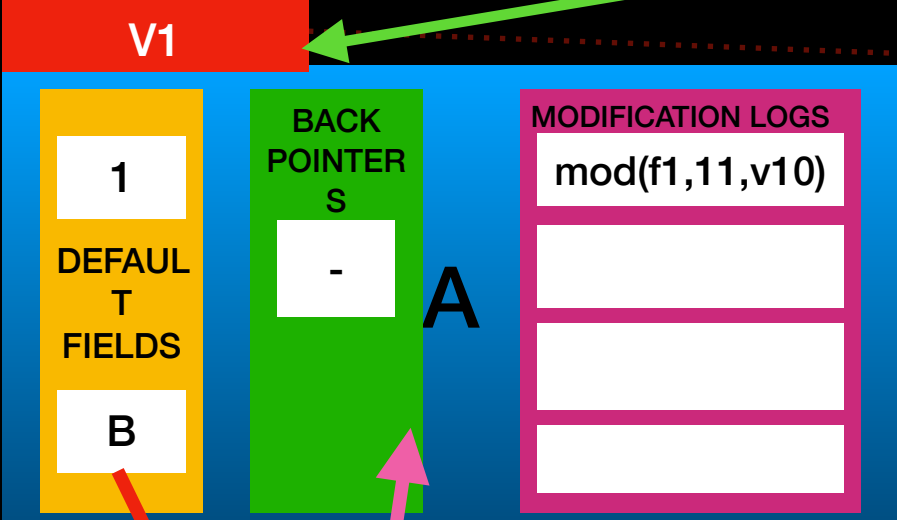
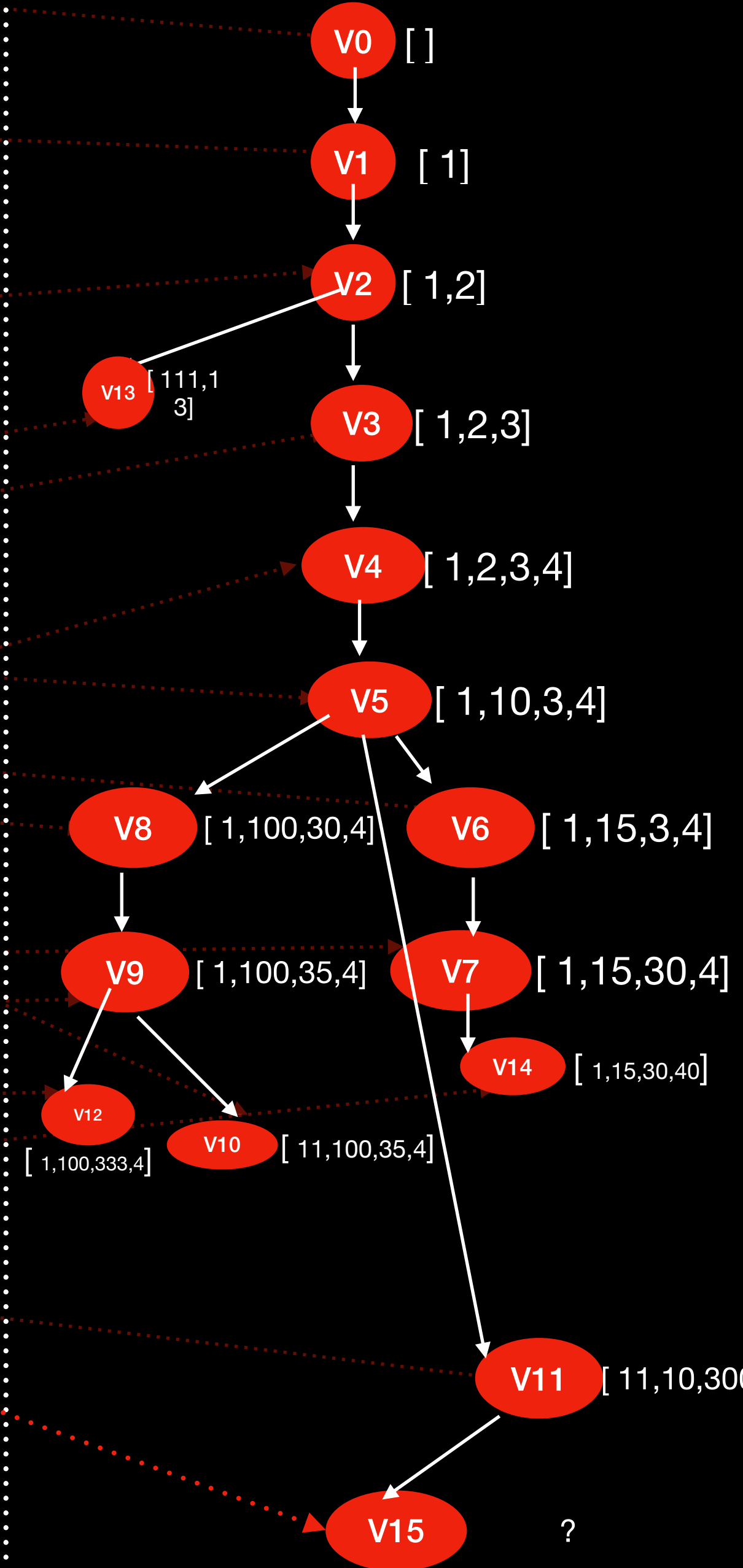
Query: update(C,f1,321,v11)

START MODULE

v0

Current time, t = 15

Version tree



STEP 1

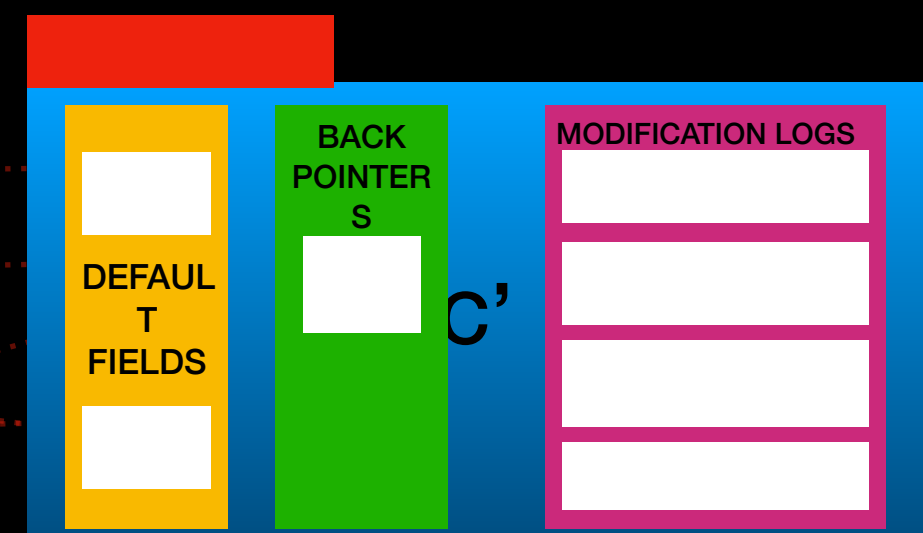
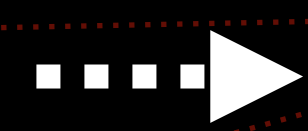
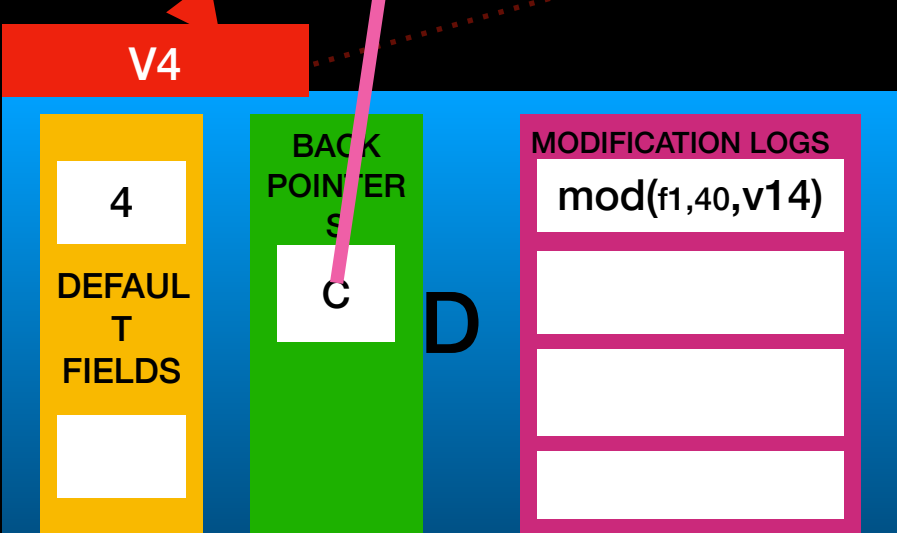
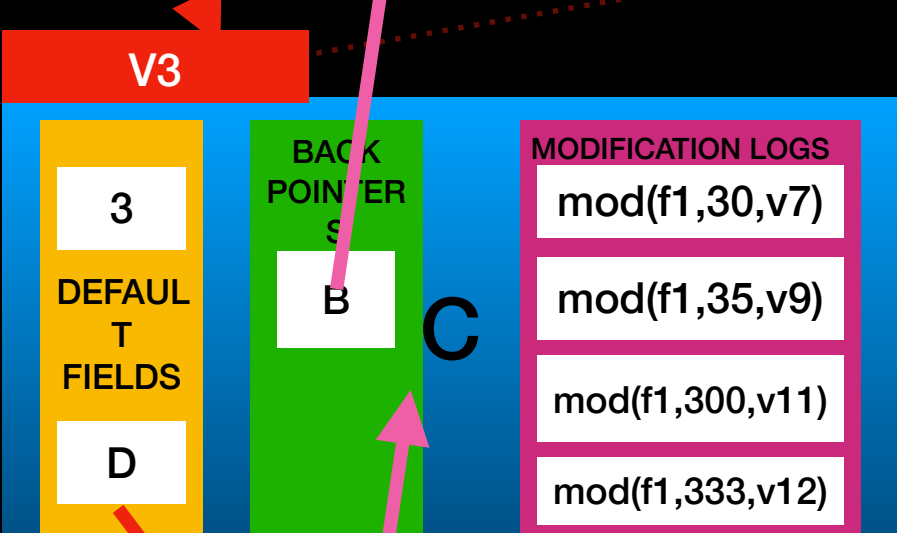
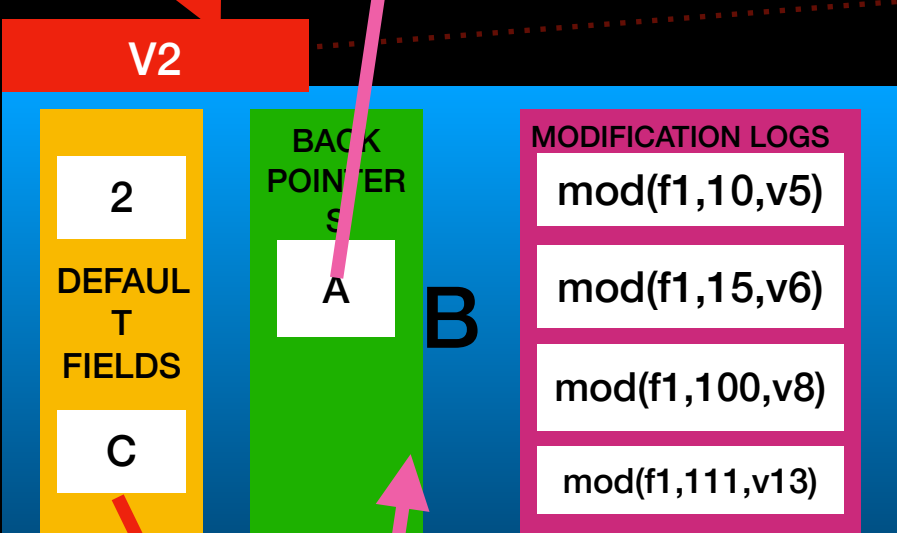
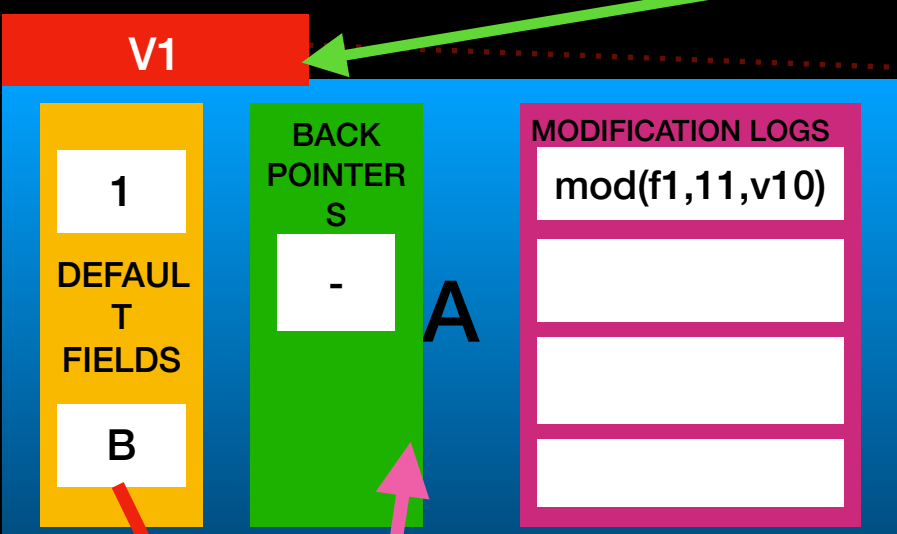
CREATE AN EMPTY NODE
NAMED C'
WITH DEF. VERSION V15

PTO->

Query: update(C,f1,321,v11)

START MODULE
v0

Current time, t = 15

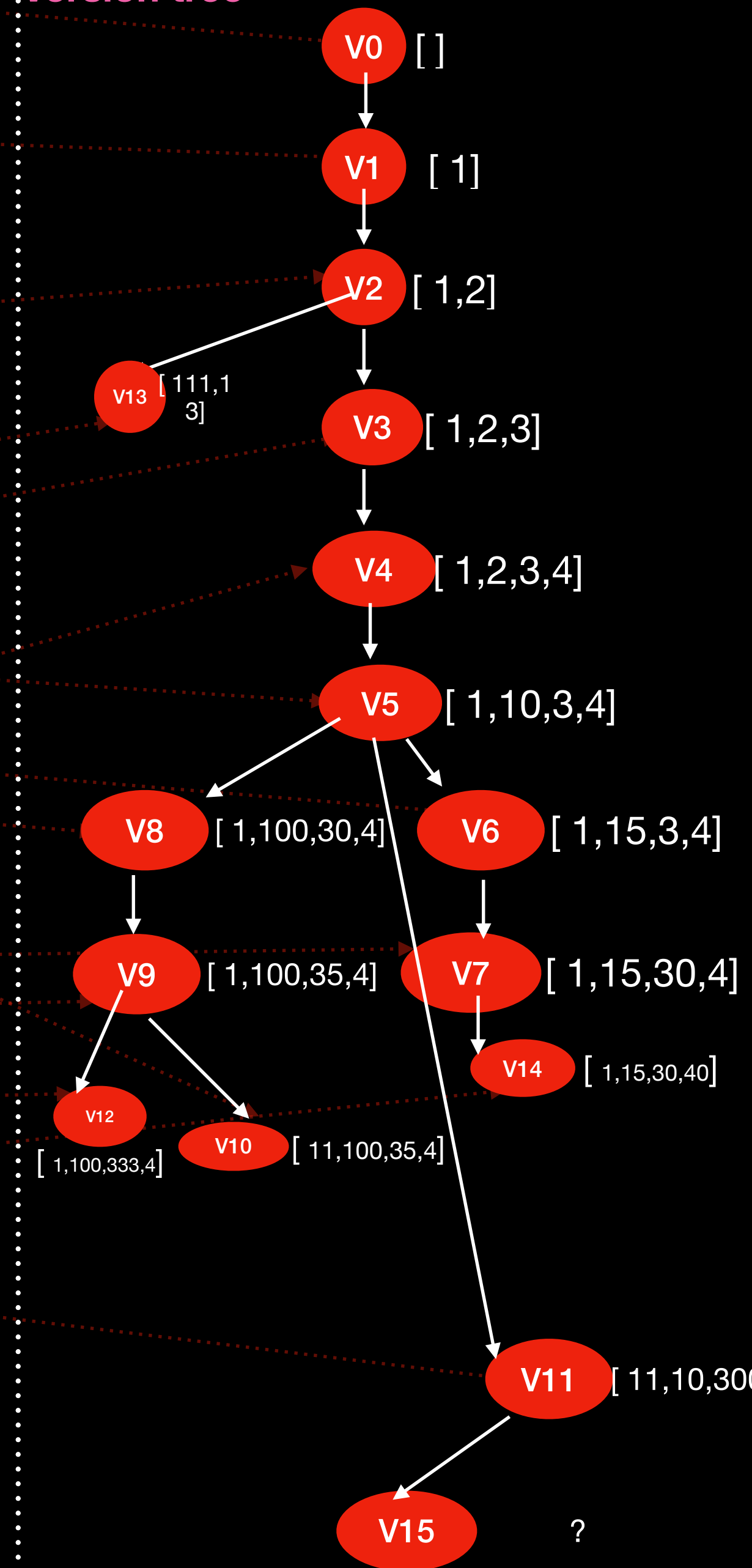


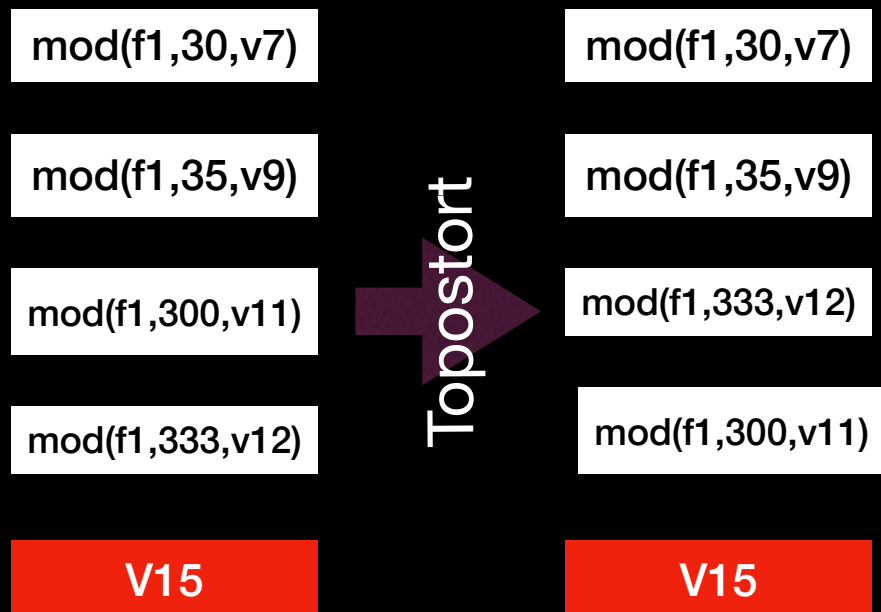
STEP 2
TRANSFER AROUND 50% MODS TO C'
FROM C

**BUT ?? HOW TO DO THE SPLITTING?
ARBITRARILY OR IN A SPECIFIC ORDER?**

PTO->

Version tree





HOW TO DO THE SPLITTING? ARBITRARILY OR IN A SPECIFIC ORDER?

TOPOLOGICALLY SORT THESE MODS ACCORDING TO THE VERSION ORDER
(if V_y is SUCCESOR of V_x , then V_y is considered Greater - hence V_y will got to right subtree)
Thus, you create an Ascending Order

Who will go to New Copy??



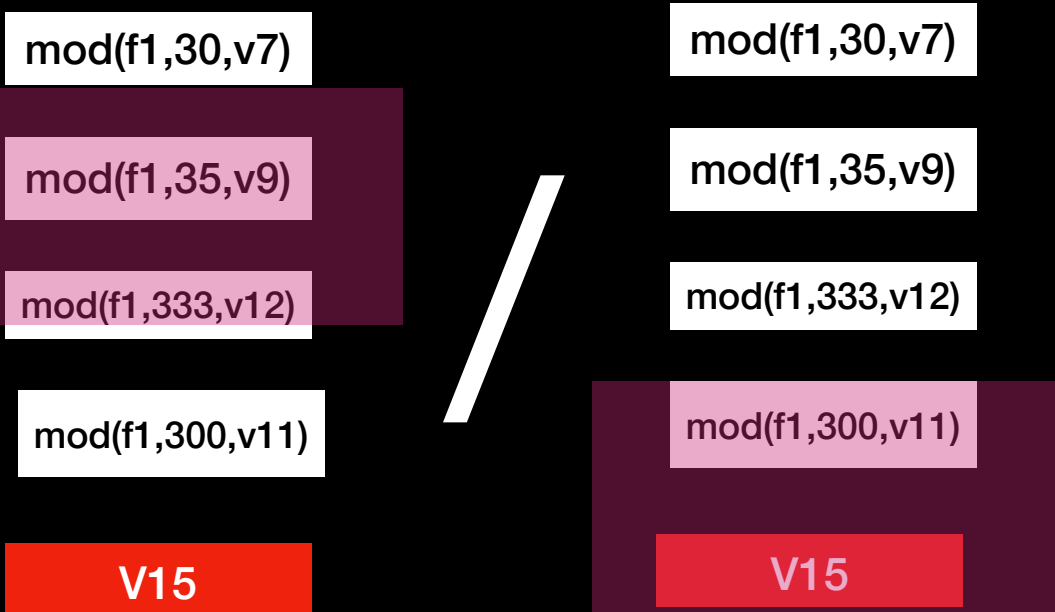
Choose
A Candidate Mod as pivot

Which has most number of mods who are strictly successors of that pivot
In Topological Order

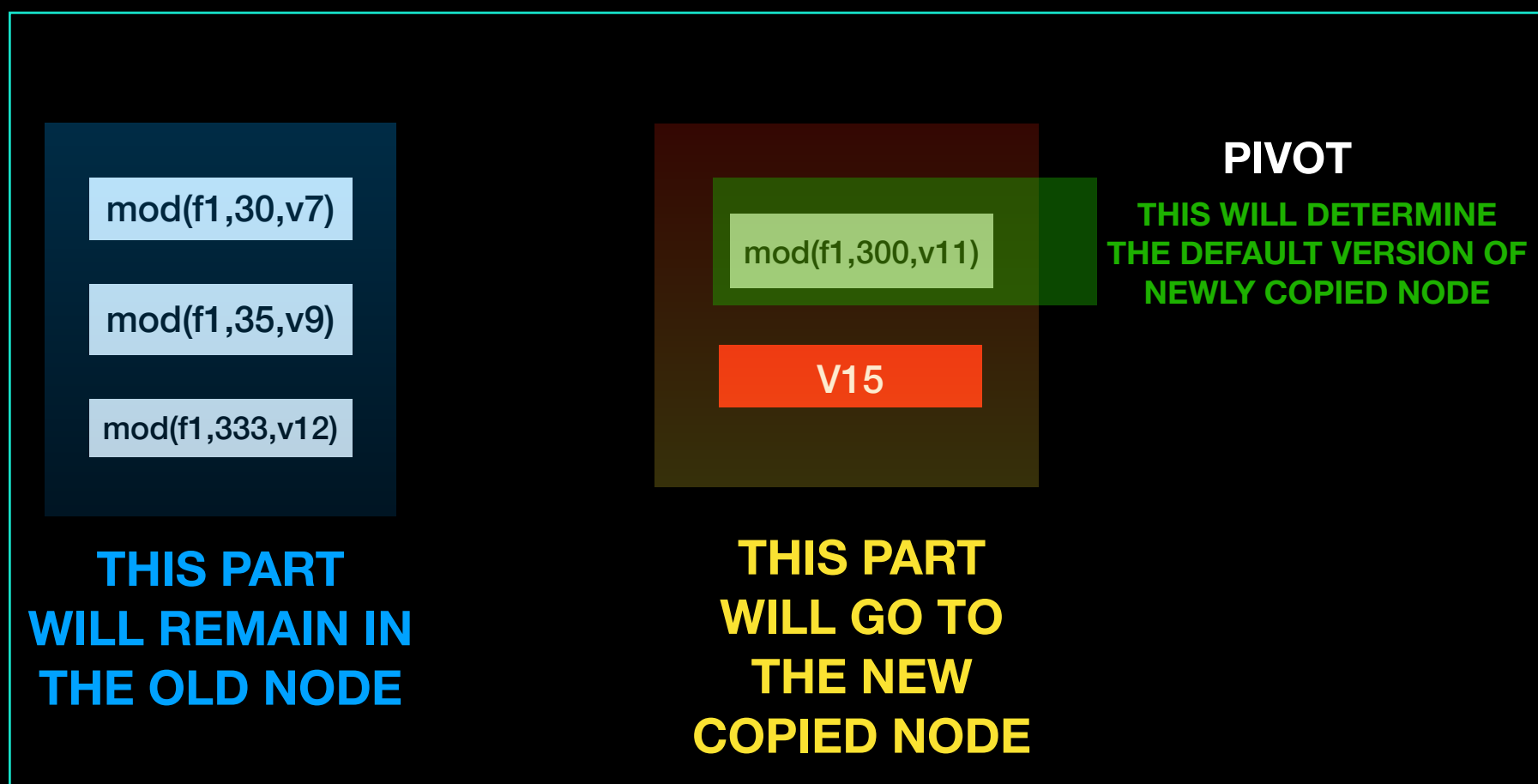
Then transfer that pivot along with its successors to the new copy.

Option 1

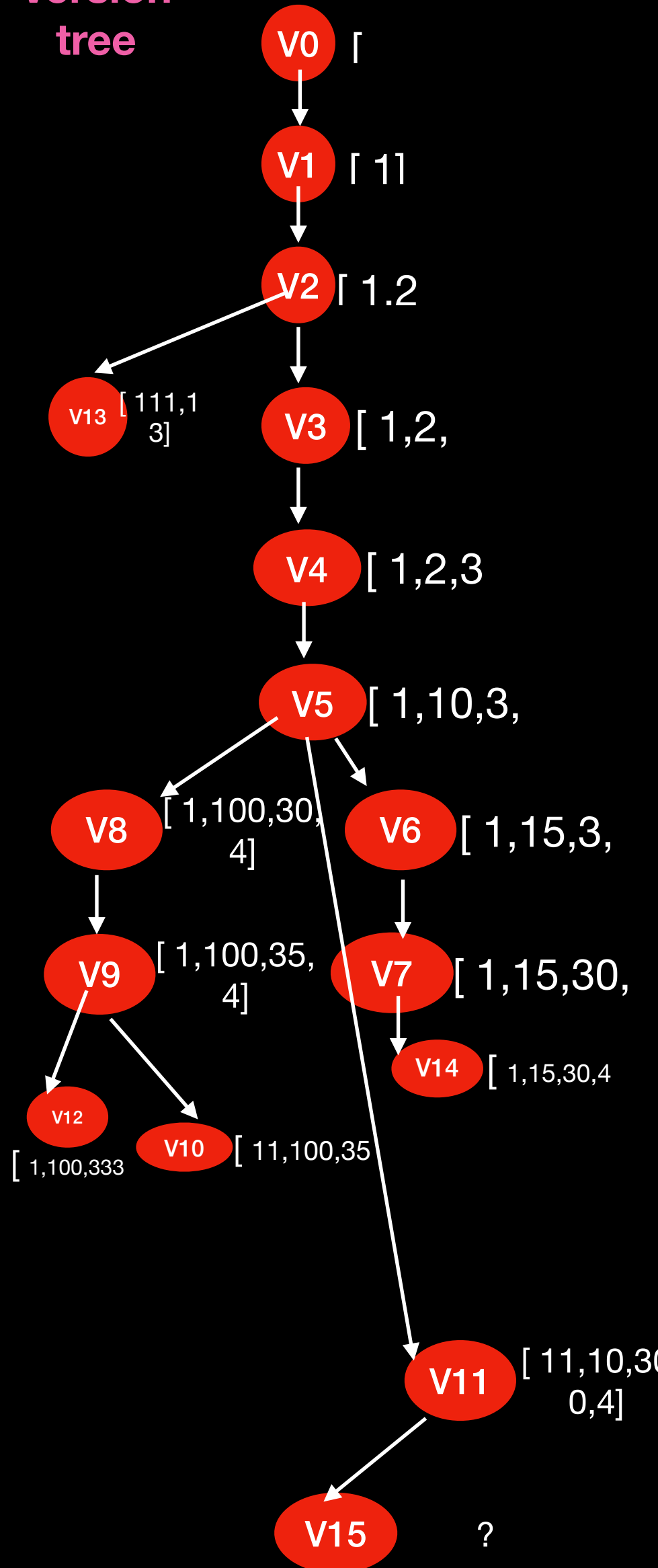
Option 2



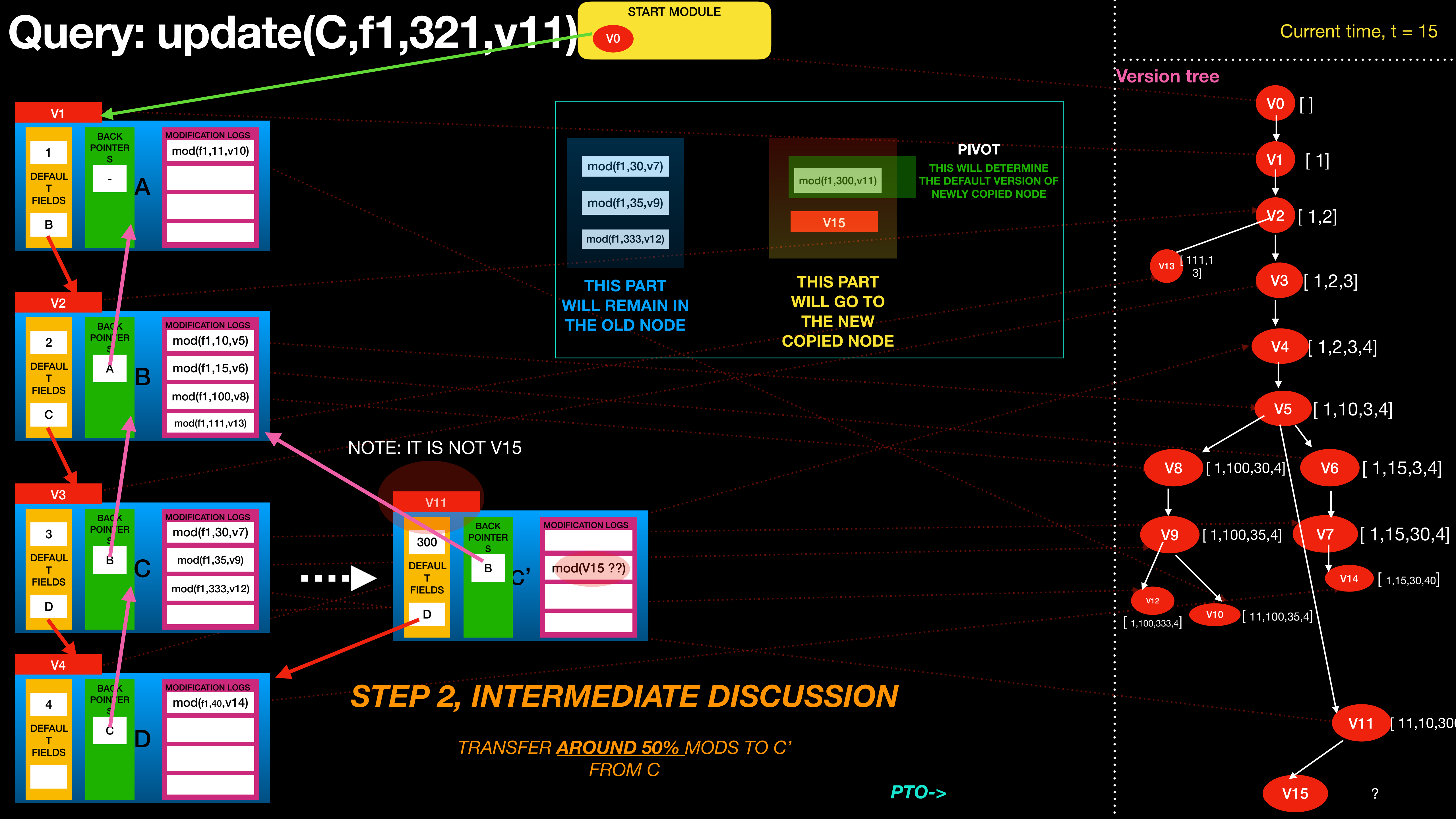
Suppose
Option 2

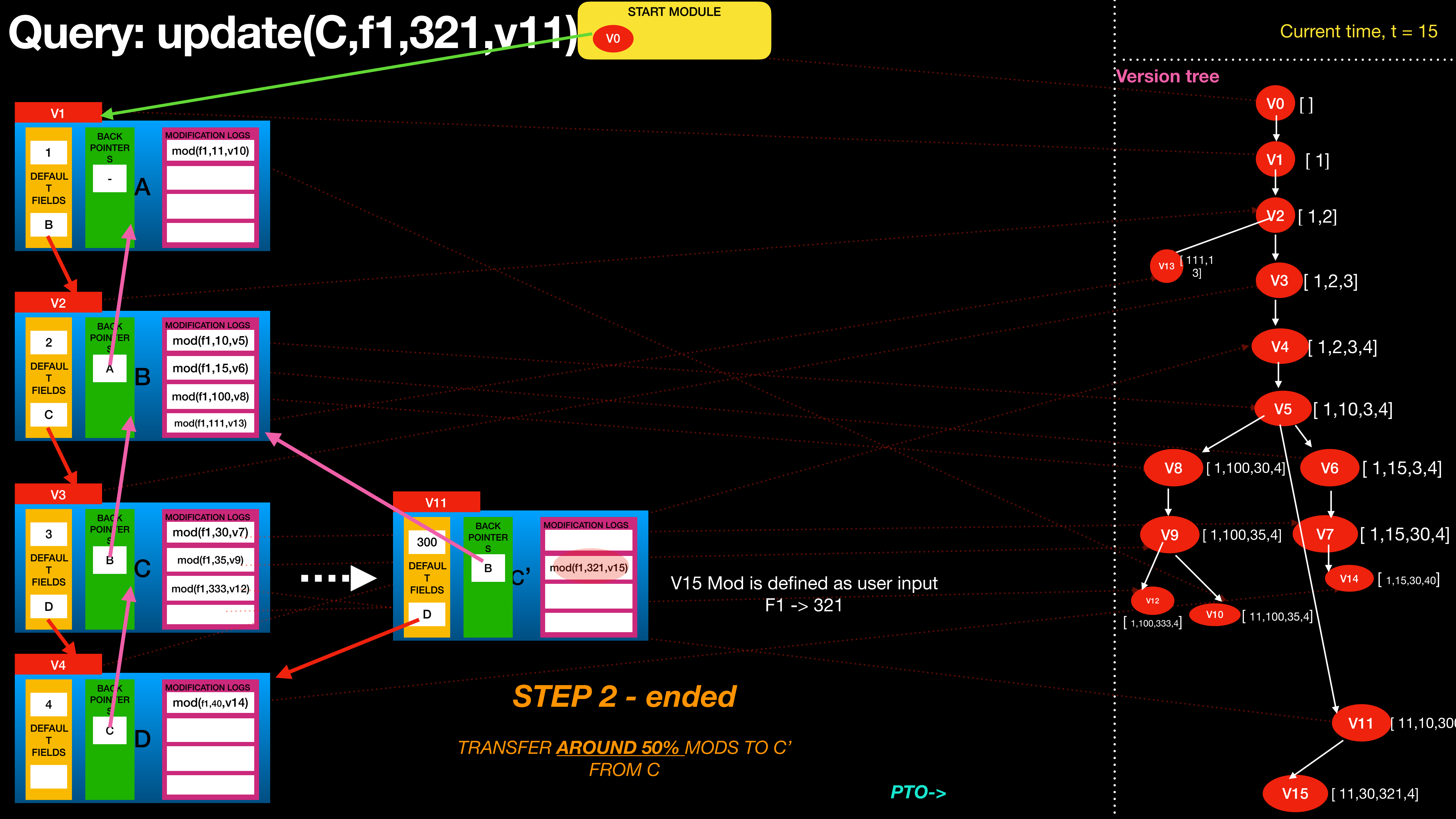


Version
tree



STEP 2, INTERMEDIATE DISCUSSION





300

DEFAULT FIELDS

D

BACK POINTER S

B

MODIFICATION LOGS

mod(f1,321,v15)

START MODULE

v0

Query: update(C,f1,321,v11)

Current time, t = 15

Version tree

v0 []

v1 [1]

v2 [1,2]

v3 [1,2,3]

v4 [1,2,3,4]

v5 [1,10,3,4]

v6 [1,15,3,4]

v7 [1,15,30,4]

v8 [1,100,30,4]

v9 [1,100,35,4]

v10 [11,100,35,4]

v11 [11,10,30,4]

v12 [1,100,333,4]

v13 [111,1,3]

v14 [1,15,30,40]

v15 [11,30,321,4]

STEP 2 - ended

TRANSFER AROUND 50% MODS TO C' FROM C

PTO->

V15 Mod is defined as user input
F1 -> 321

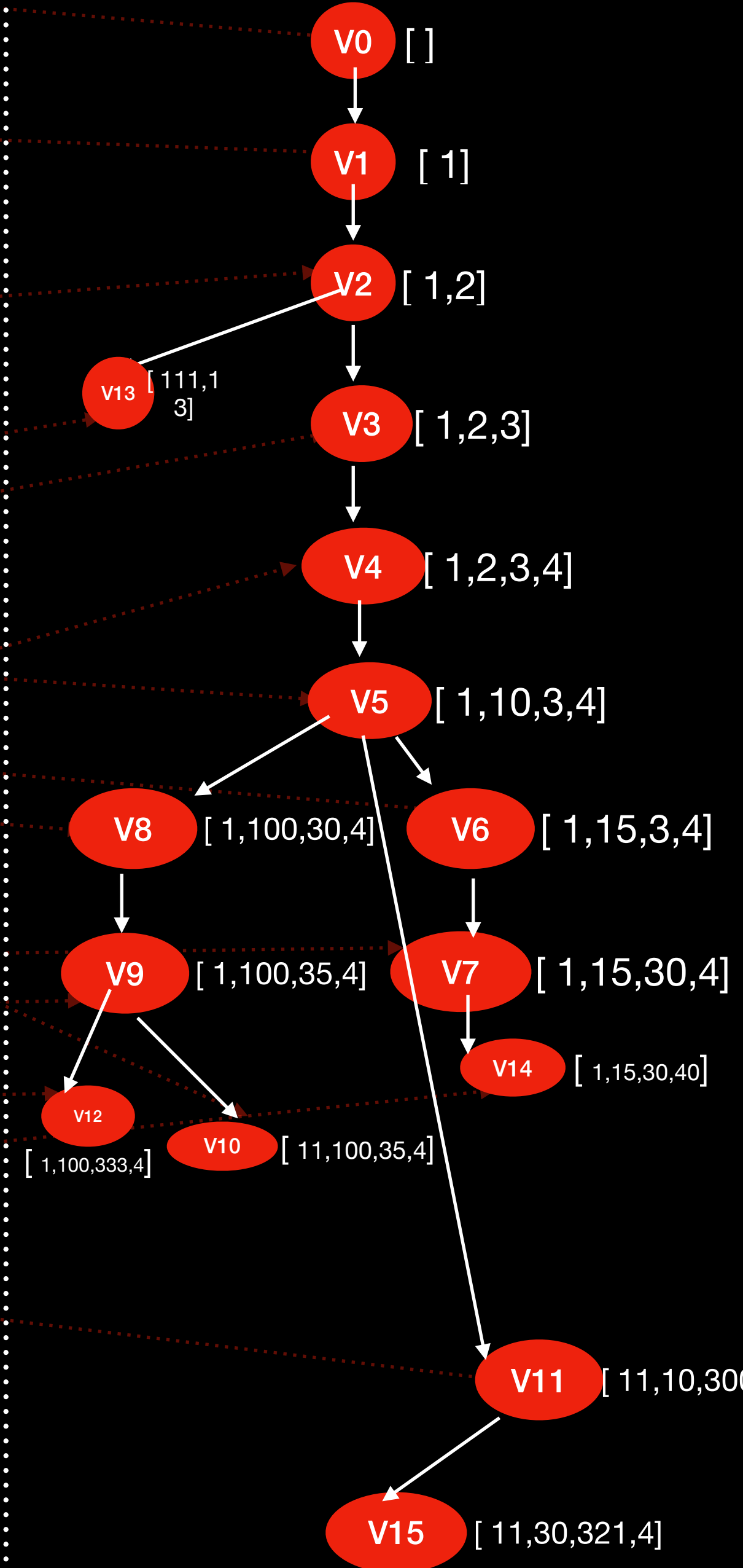
Query: update(C,f1,321,v11)

START MODULE

v0

Current time, t = 15

Version tree



Note:
The argument is v11, because a NEW NODE
Has been sliced off with def. Version v11
So, we send Update Query to B to add a fwd pointer
To a v11.

Here no need to add v11 under v2 as we already know
V11 is topologically at lower poison wrt v2

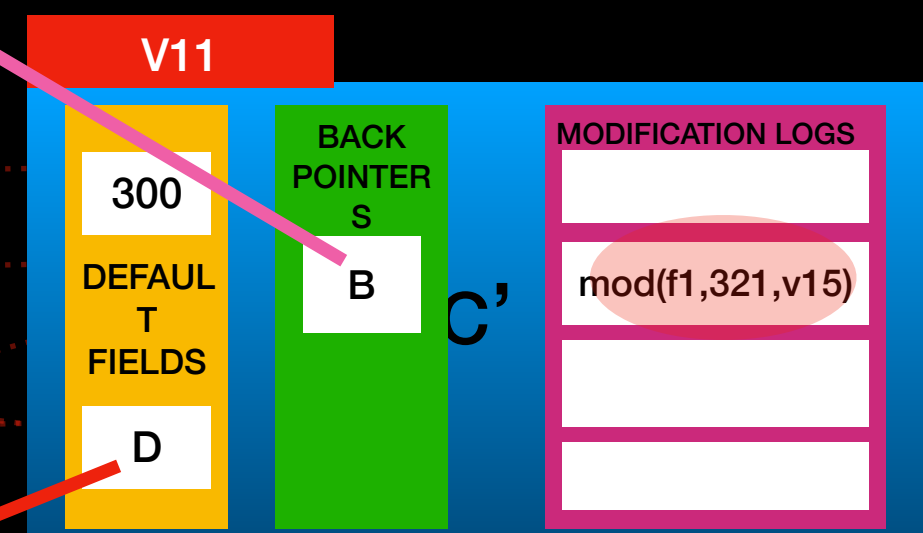
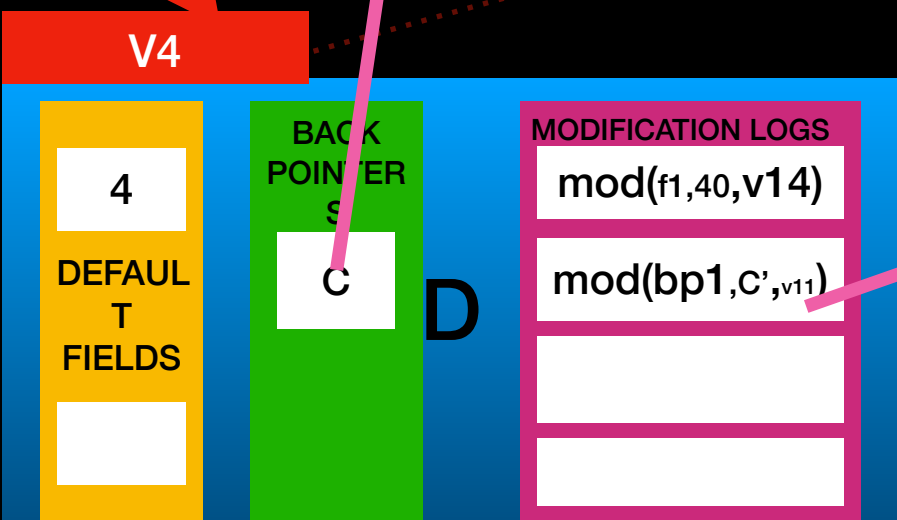
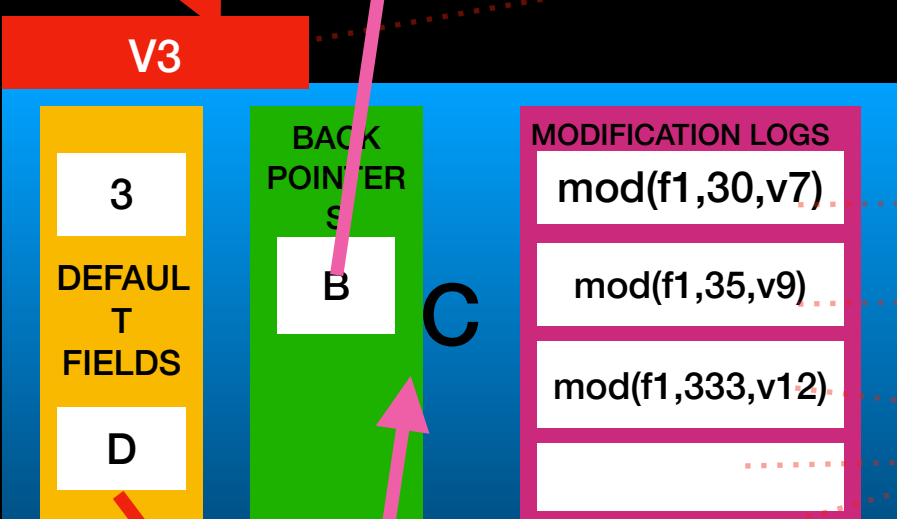
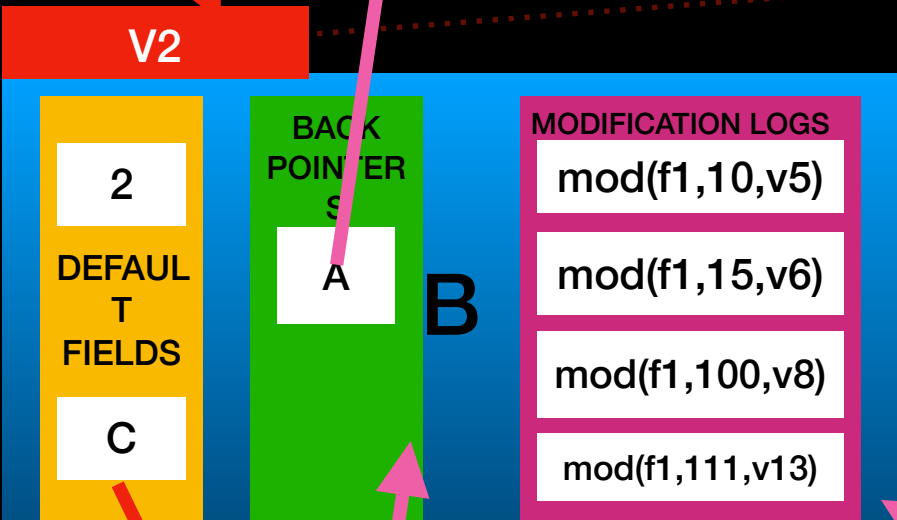
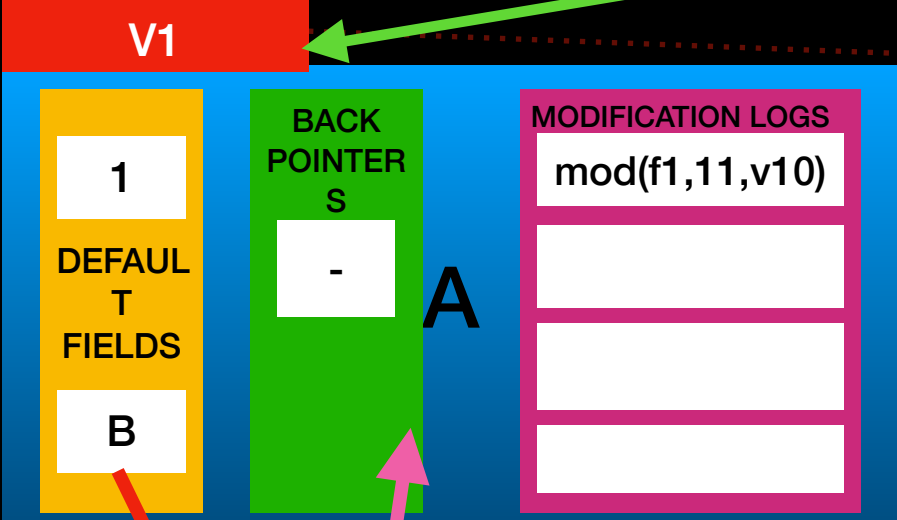
STEP 3

RECURSIVELY
MODIFY THE BACK POINTERS
OF SUCCESSOR

update(D,bp1,C',v11)

Note: Here We Are Not Replacing The Back-pointers
Rather adding mods for bp too (unlike Partial)
This is why, we are strong more number of MODS per
Node

PTO->



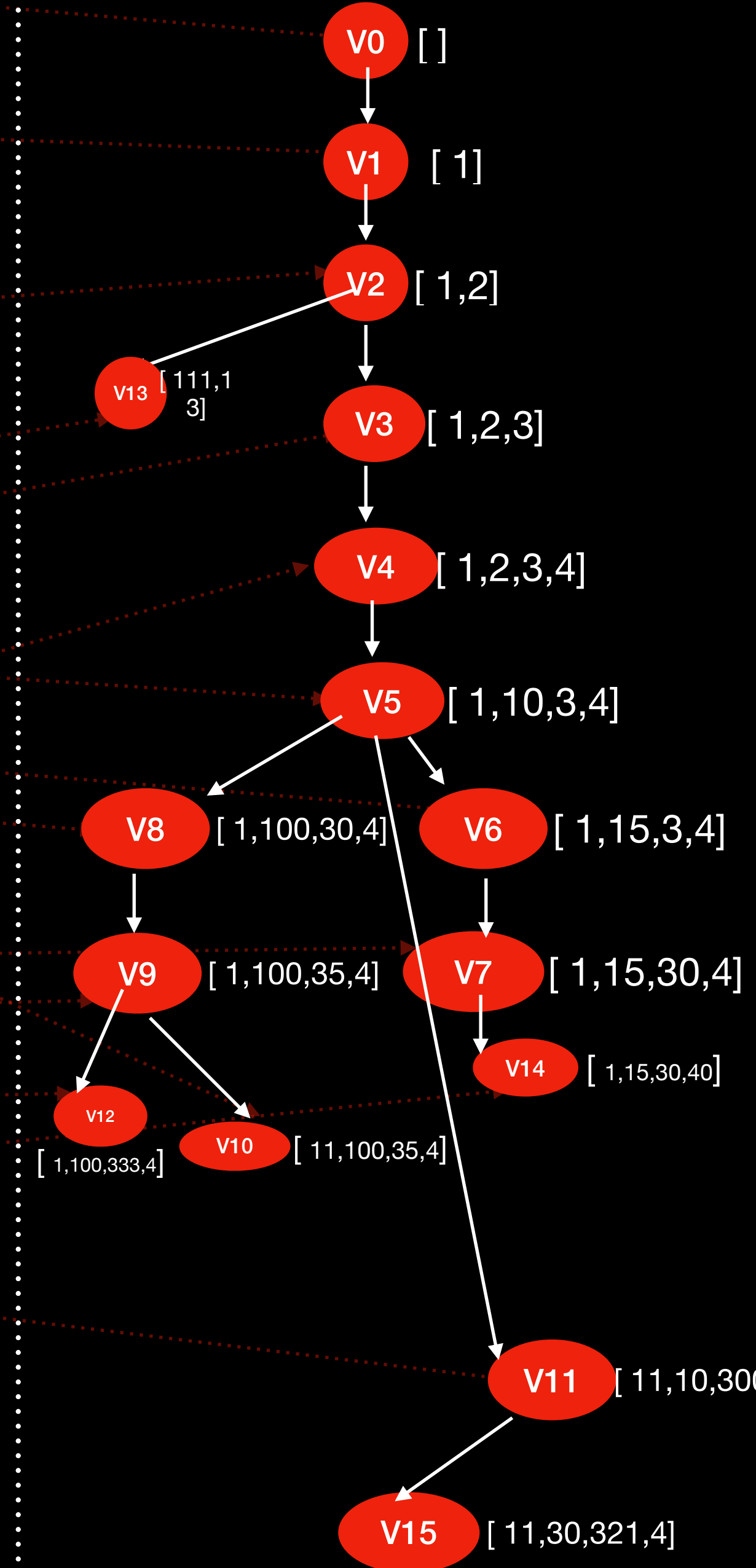
Query: update(C,f1,321,v11)

START MODULE

v0

Current time, t = 15

Version tree



Note:
The argument is v11, because a NEW NODE
Has been sliced off with def. Version v11
So, we send Update Query to B to add a fwd pointer
To a v11.

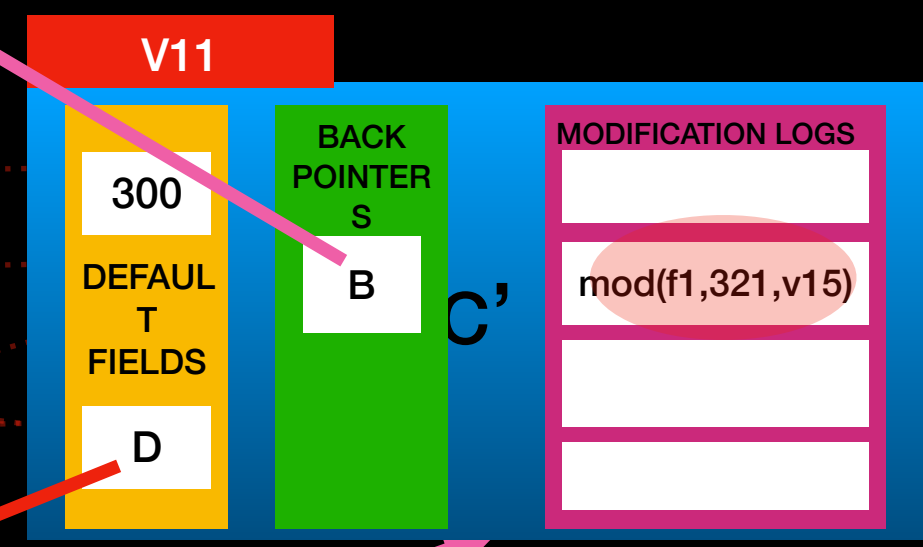
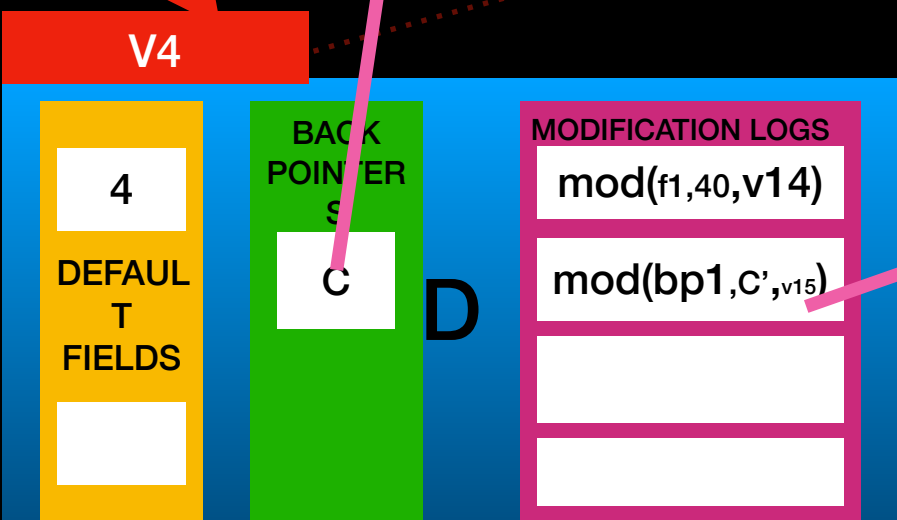
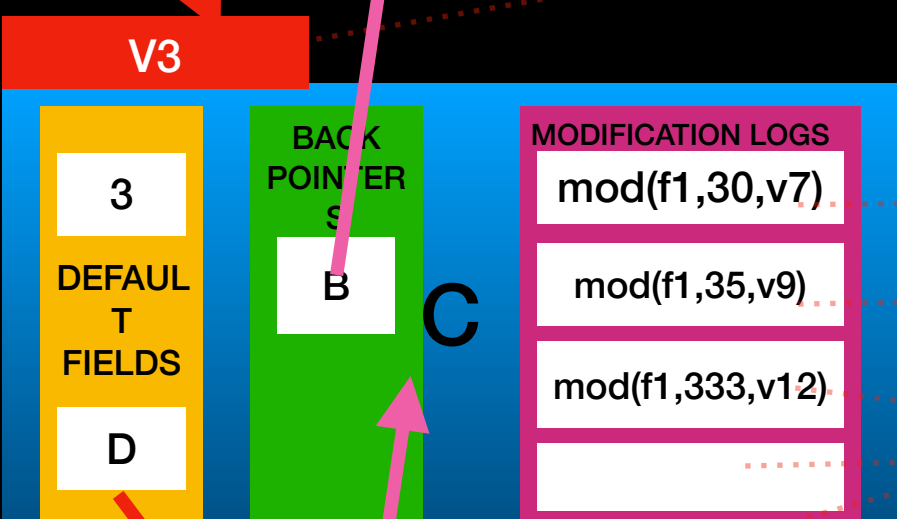
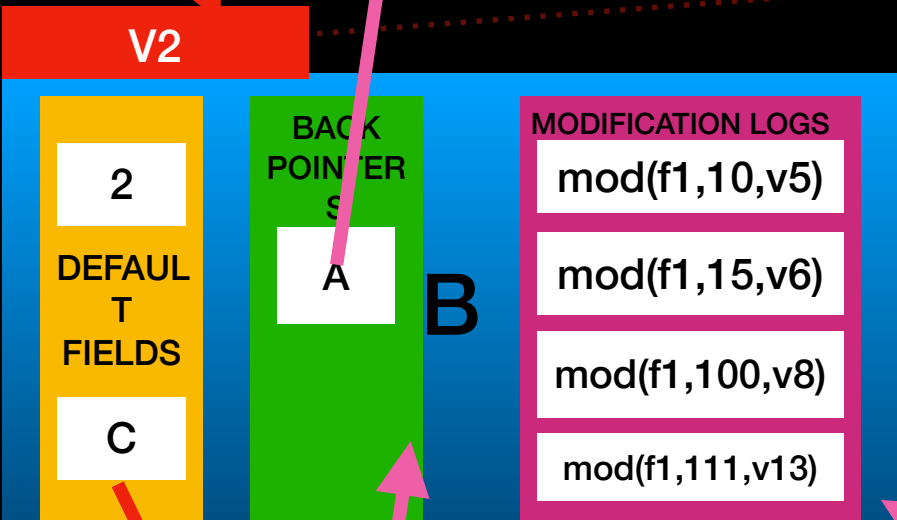
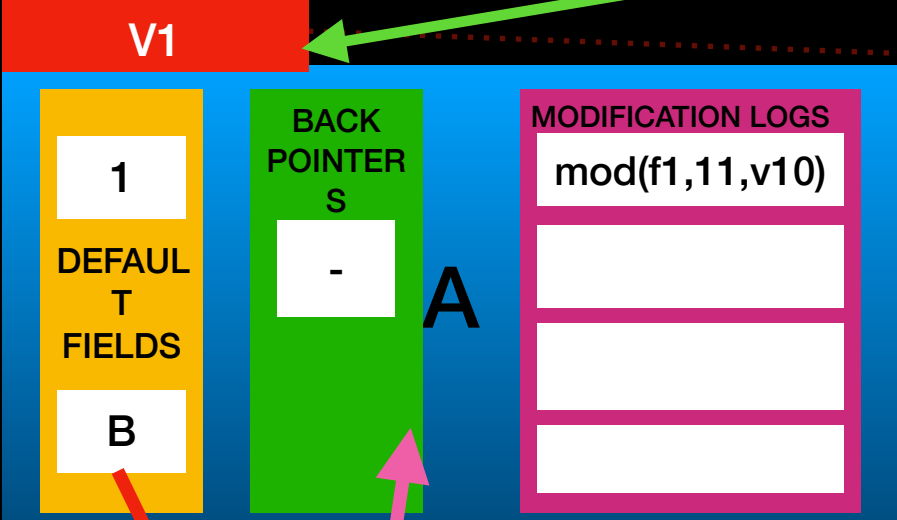
Here no need to add v11 under v2 as we already know
V11 is topologically at lower position wrt v2

STEP 4

RECURSIVELY
MODIFY THE FORWARD POINTERS
OF ANCESTORS

update(B,f2,C',v11)

PTO->



Query: update(C,f1,321,v11)

START MODULE

v0

Current time, t = 15

STEP 4

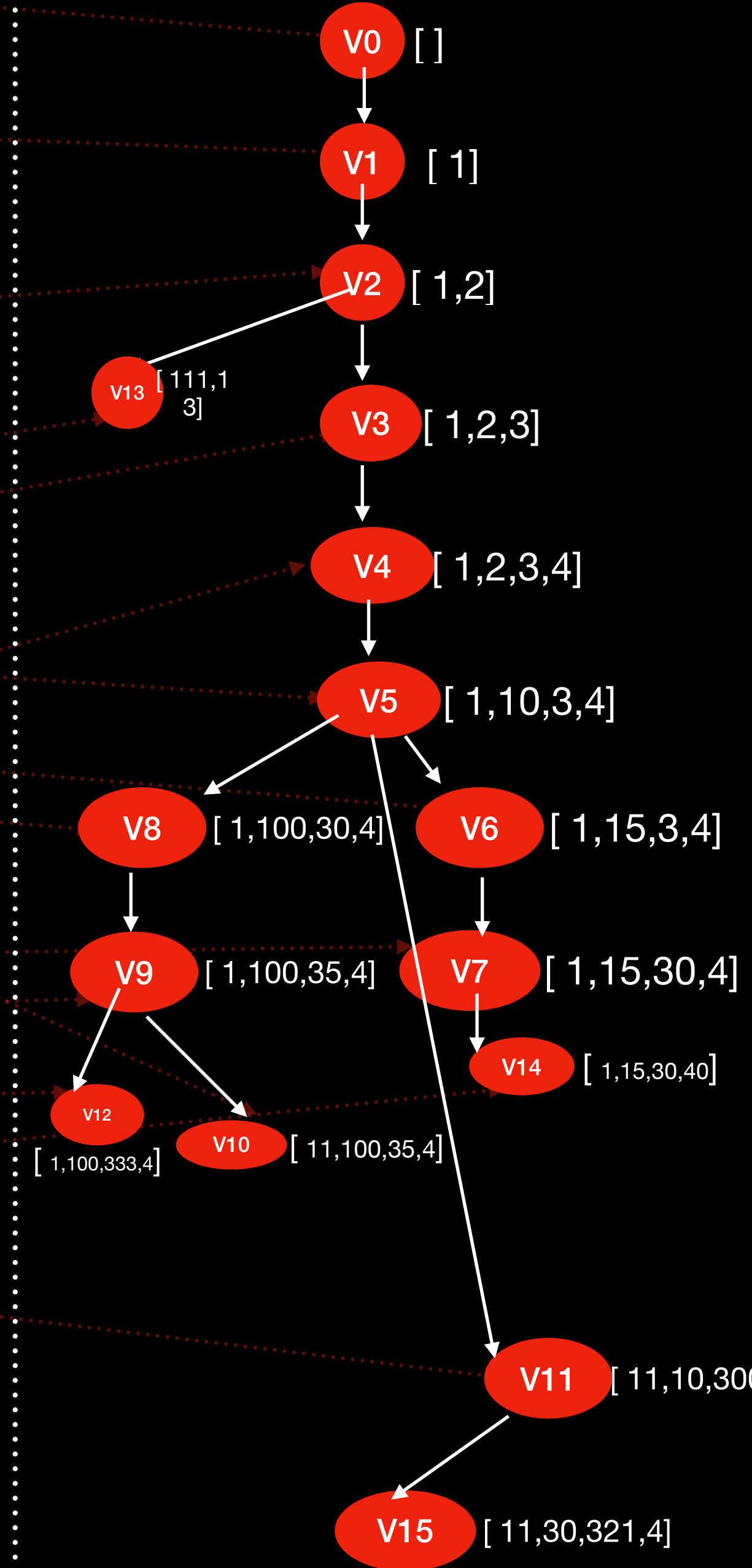
RECURSIVELY
MODIFY THE FORWARD POINTERS
OF ANCESTORS

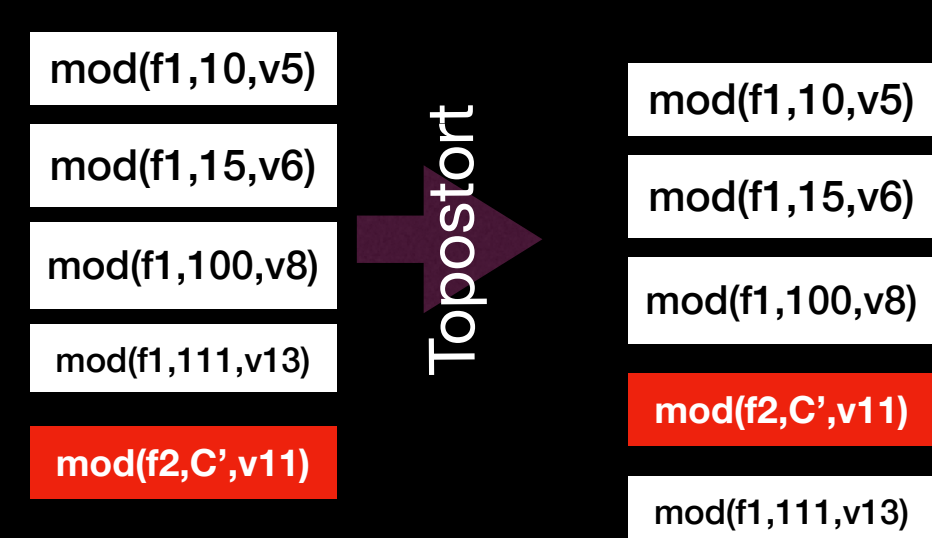
update(B,f2,C',v11)

Mod Logs of B is full!
So create a copy of B at v15 and so on ...

PTO->

Version tree





HOW TO DO THE SPLITTING? ARBITRARILY OR IN A SPECIFIC ORDER?

TOPOLOGICALLY SORT THESE MODS ACCORDING TO THE VERSION ORDER
(if V_y is SUCCESOR of V_x , then V_y is considered Greater - hence V_y will got to right subtree)
Thus, you create an Ascending Order

Who will go to New Copy??

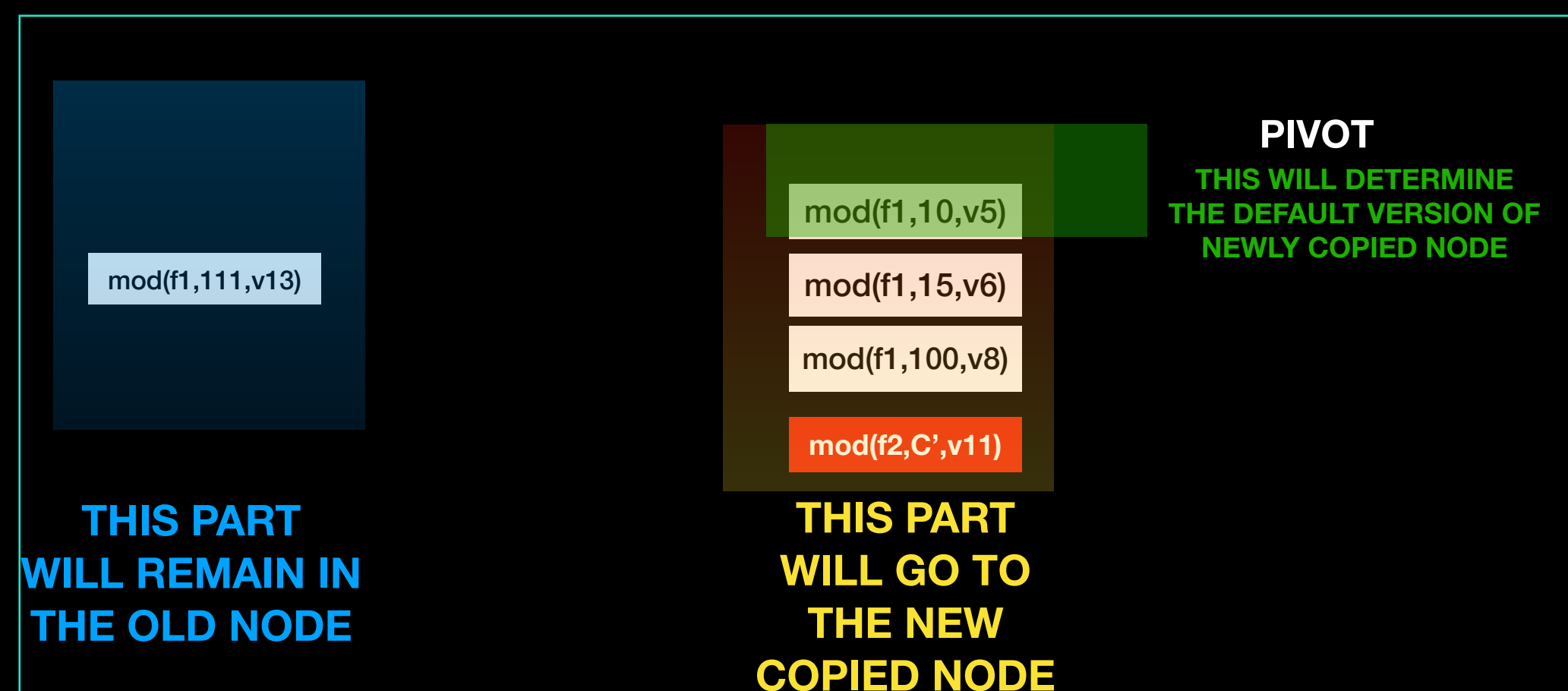
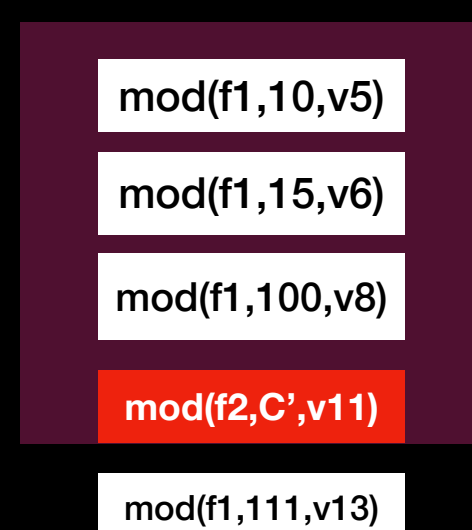


Choose
A Candidate Mod as pivot

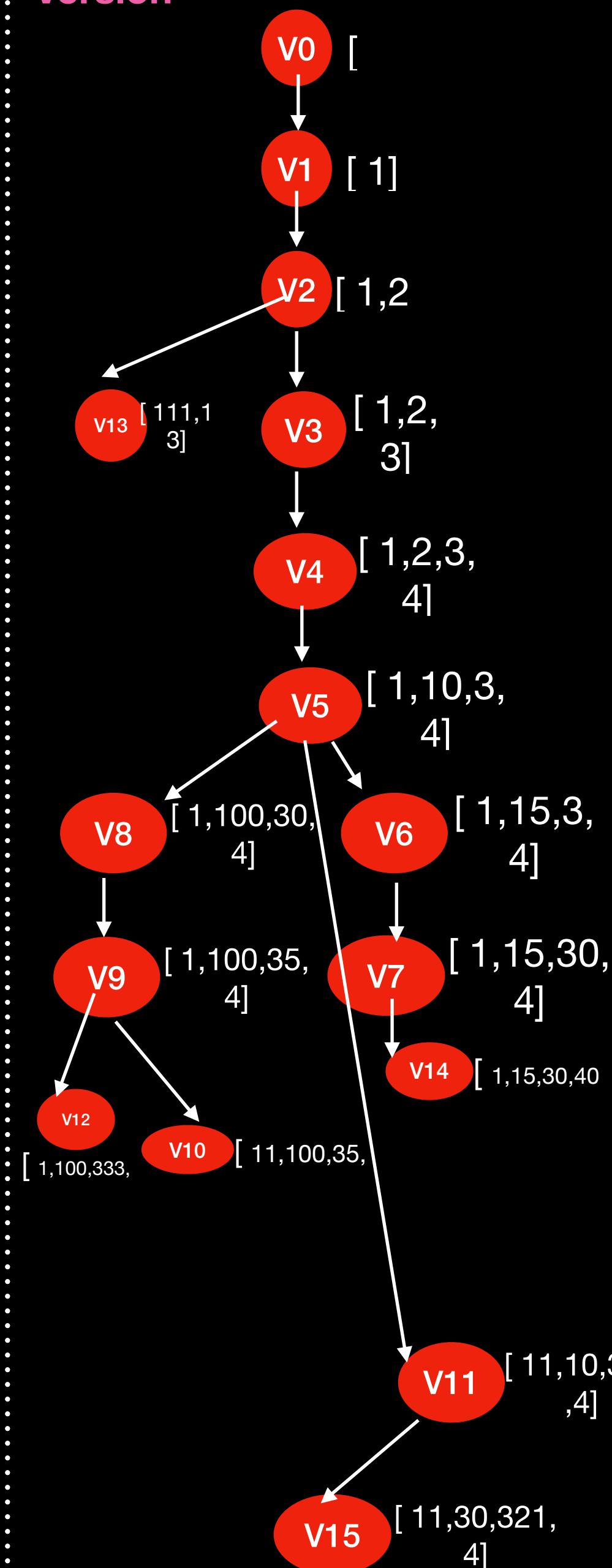
Which has most number of mods who are strictly successors of that pivot
In Topological Order

Then transfer that pivot along with its successors to the new copy.

I am still thinking for an optimisation
Here. Any one help!!



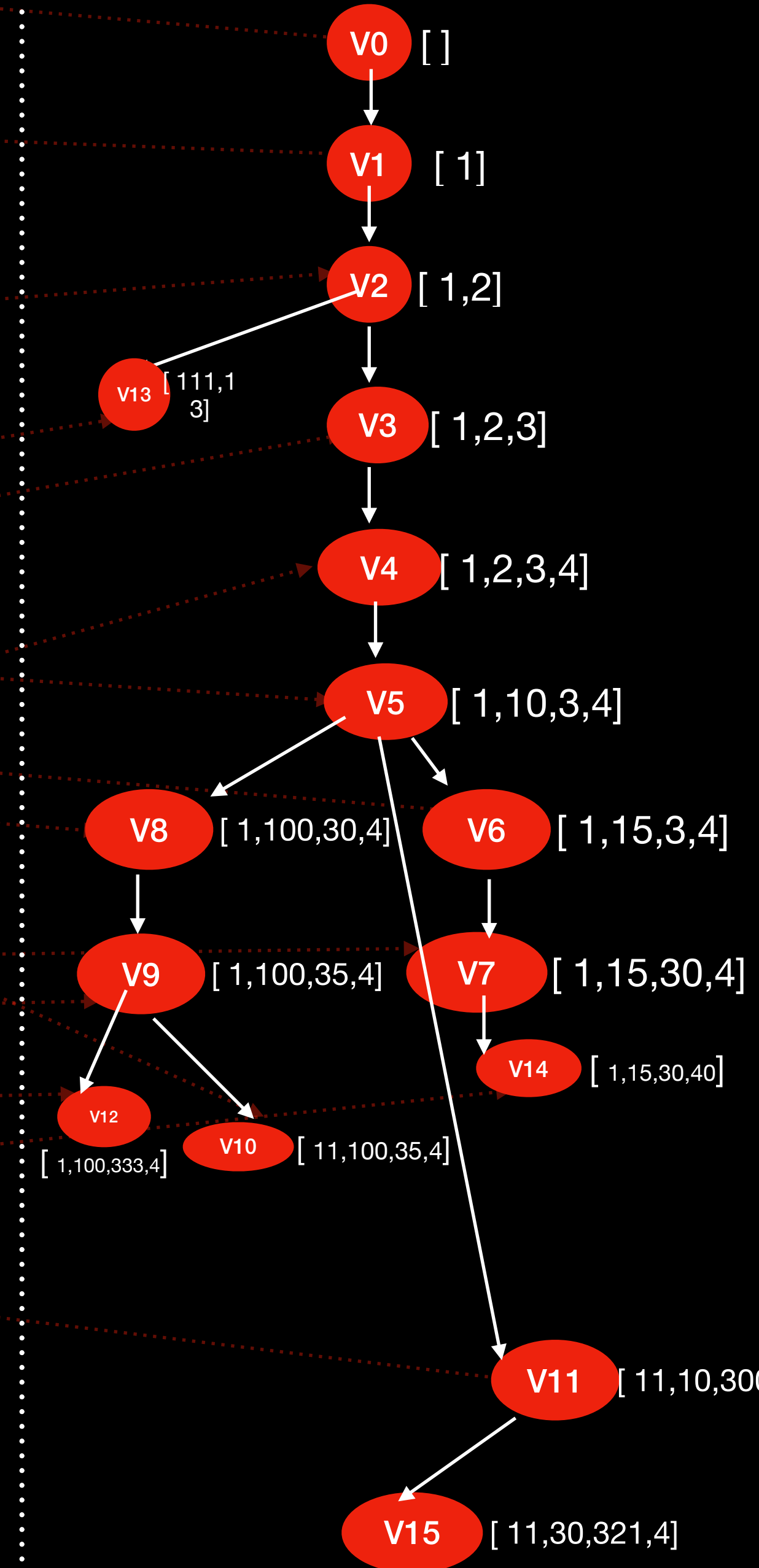
Version

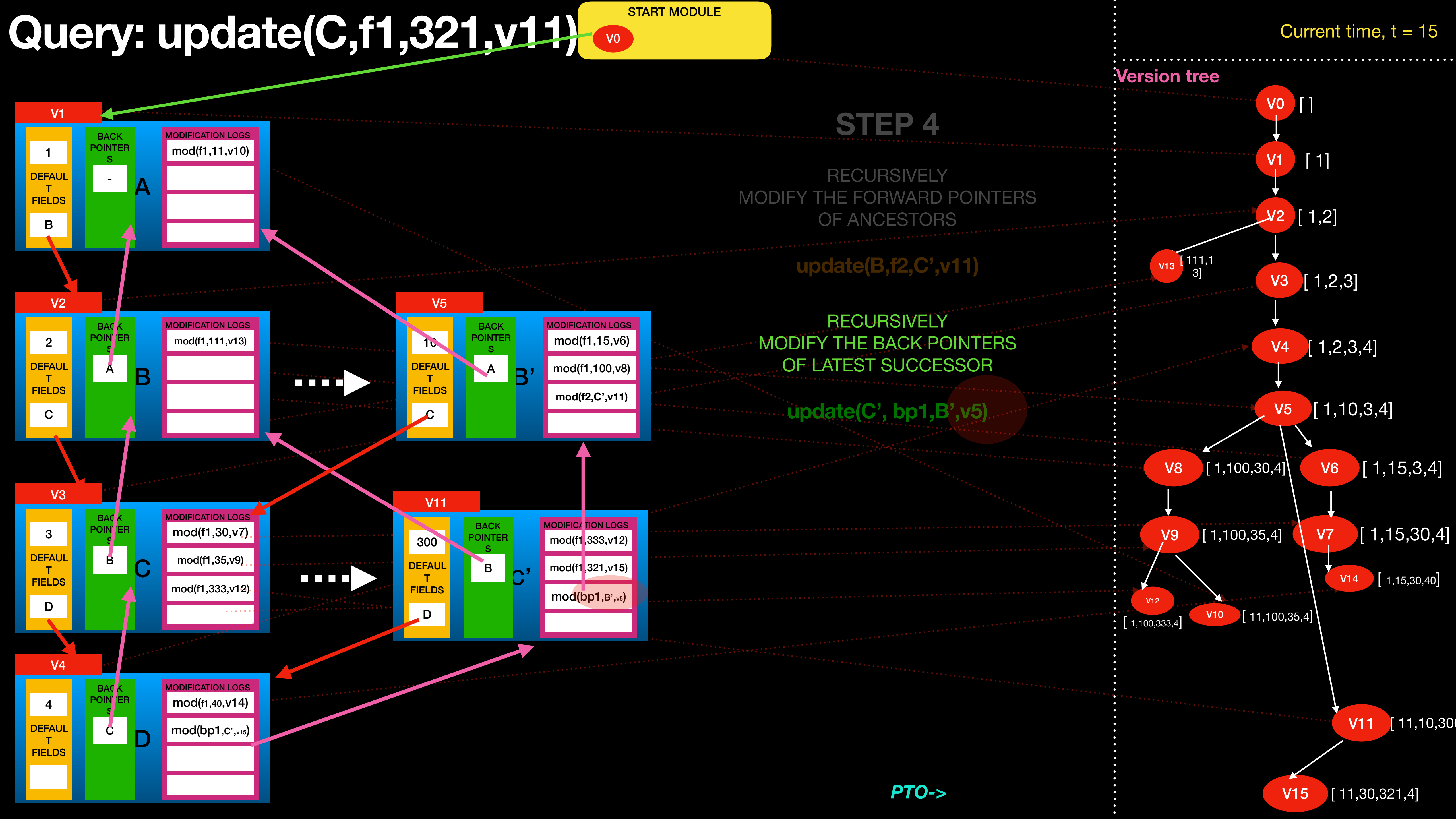


V0

Version tree

PTO->





10

DEFAULT
T
FIELDS

C

BACK
POINTER
S

A

B'

MODIFICATION LOGS

mod(f1,15,v6)

mod(f1,100,v8)

mod(f2,C',v11)

300

DEFAULT
T
FIELDS

D

BACK
POINTER
S

B

C'

MODIFICATION LOGS

mod(f1,333,v12)

mod(f1,321,v15)

mod(bp1,B',v5)

START MODULE

v0

Query: update(C,f1,321,v11)

STEP 4

RECURSIVELY
MODIFY THE FORWARD POINTERS
OF ANCESTORS

update(B,f2,C',v11)

RECURSIVELY
MODIFY THE BACK POINTERS
OF LATEST SUCCESSOR

update(C', bp1,B',v5)

Version tree

v0 []

v1 [1]

v2 [1,2]

v3 [1,2,3]

v4 [1,2,3,4]

v5 [1,10,3,4]

v6 [1,15,3,4]

v7 [1,15,30,4]

v8 [1,100,30,4]

v9 [1,100,35,4]

v10 [11,100,35,4]

v11 [11,10,30,4]

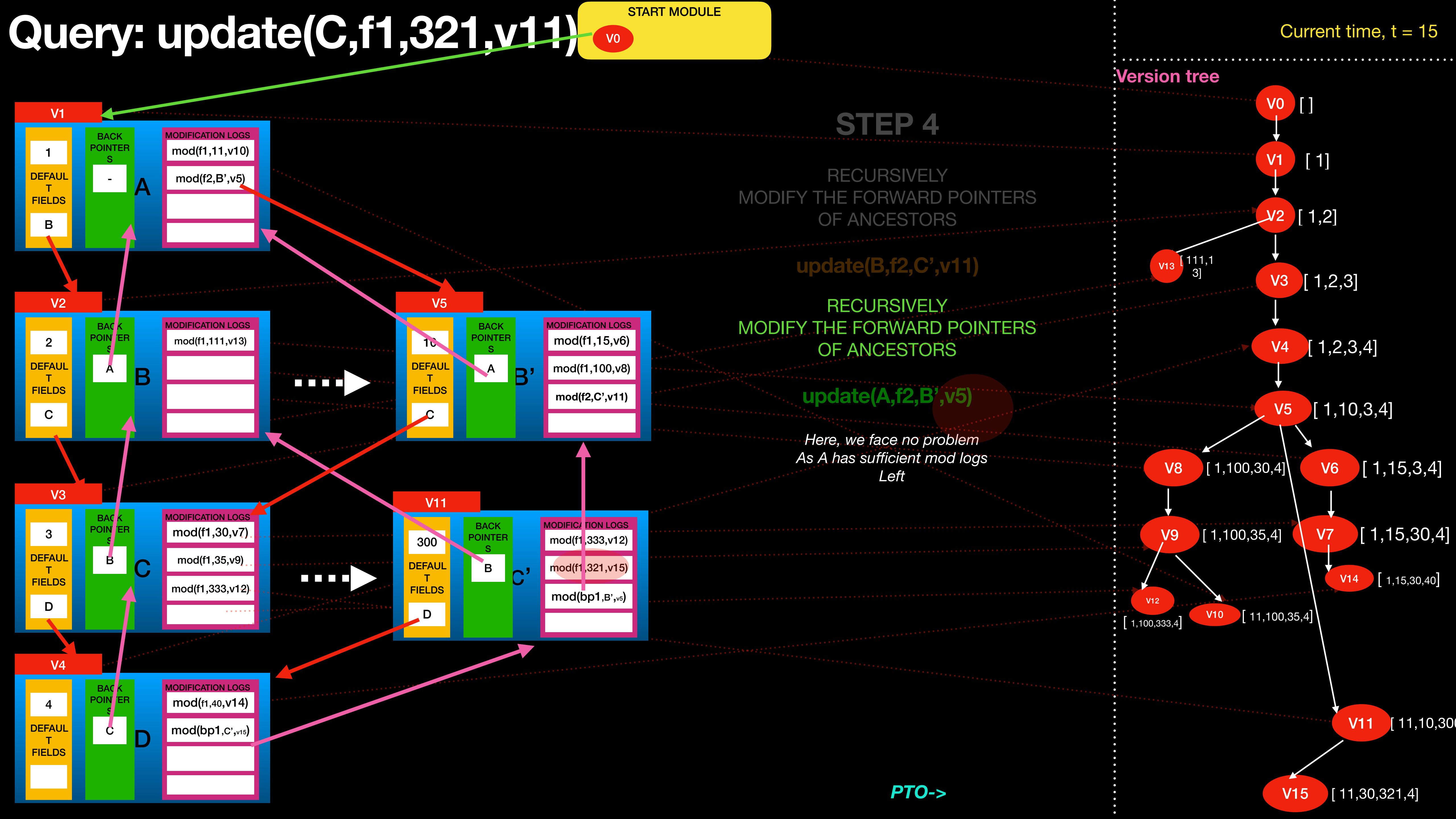
v12 [1,100,333,4]

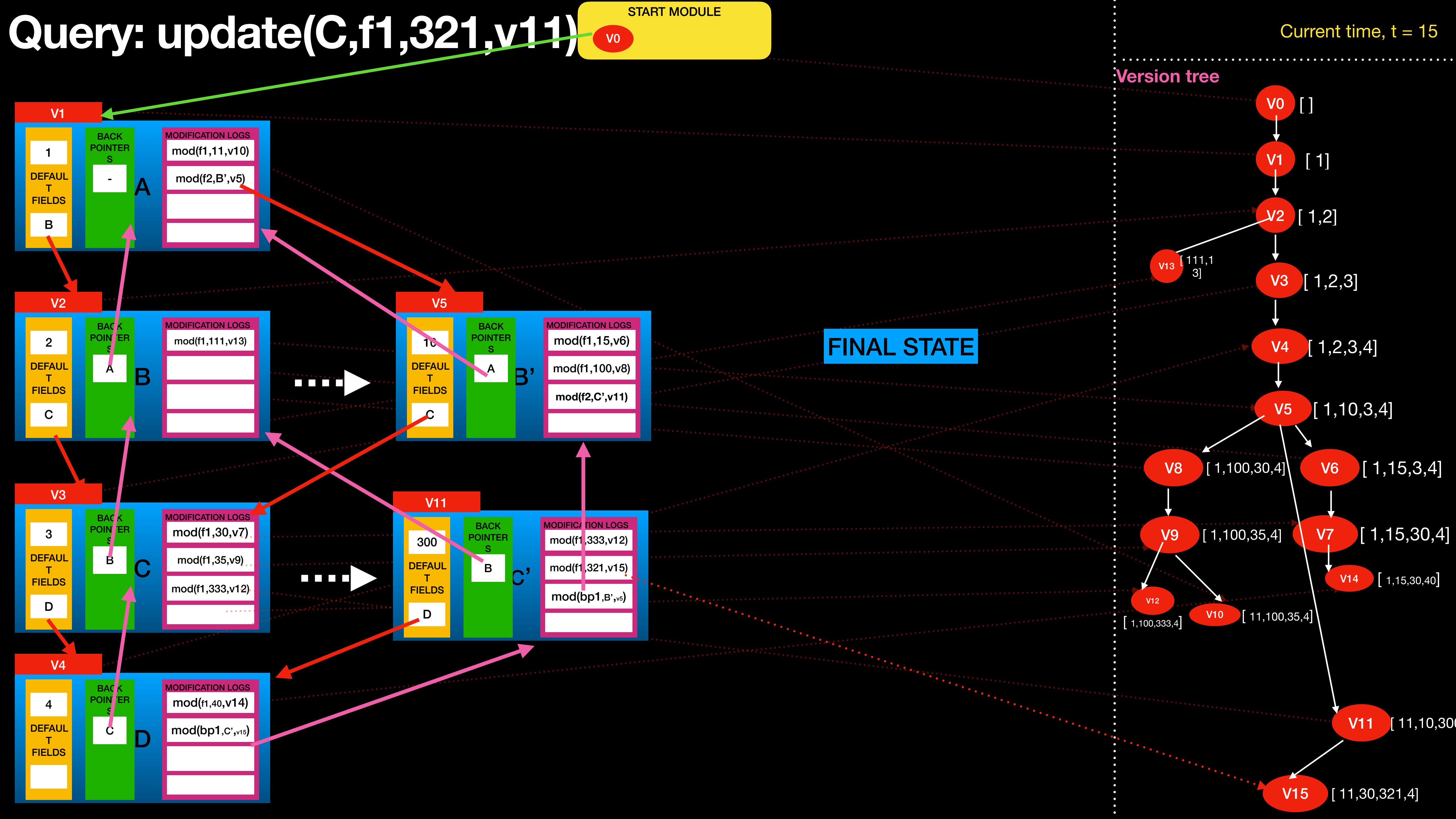
v13 [111,1,3]

v14 [1,15,30,40]

v15 [11,30,321,4]

PTO->





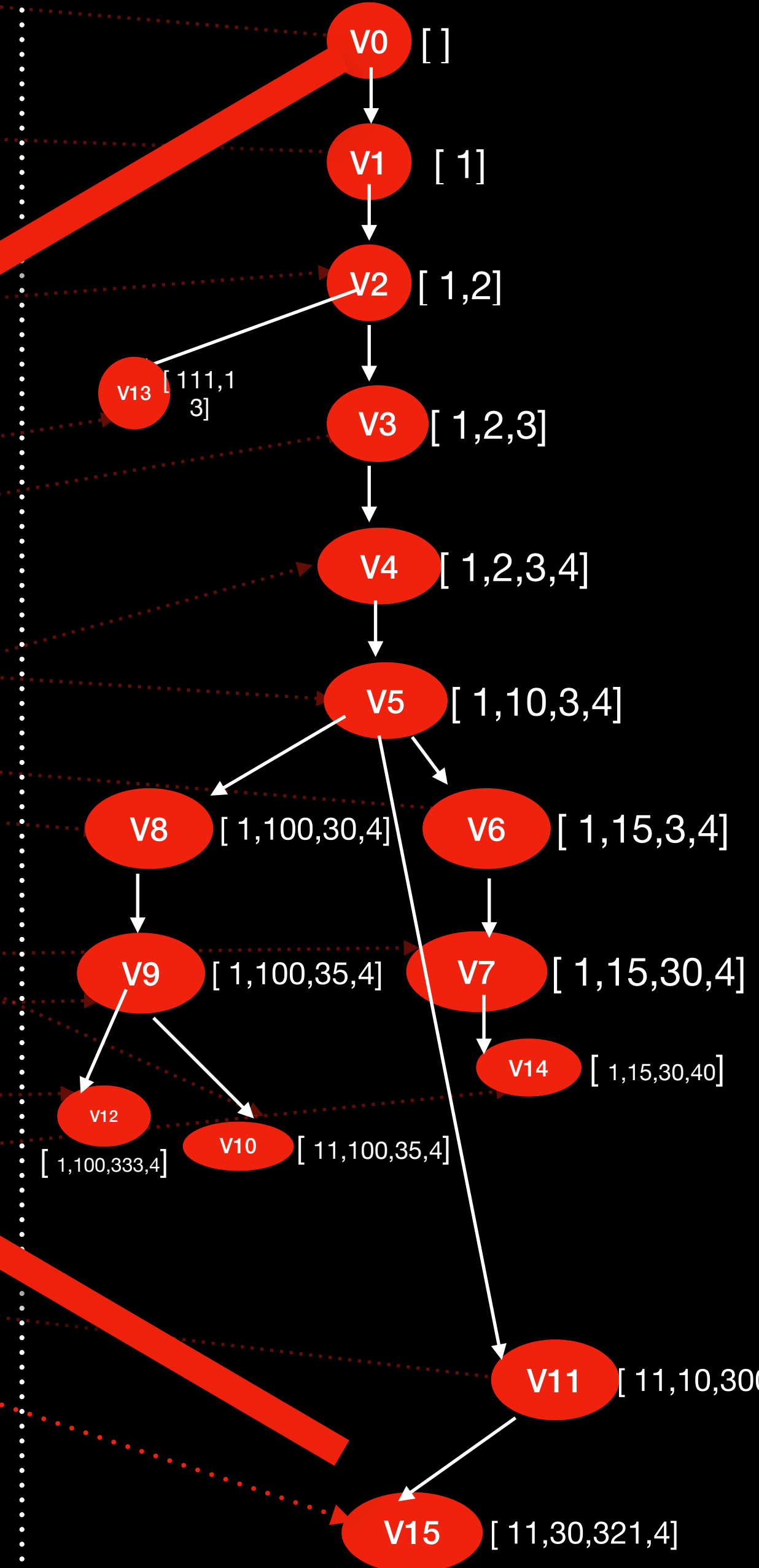
Current time, $t = 15$

C Node was absent In Version 13

We have to check it in implementation

~~FINAL STATE~~

Version tree



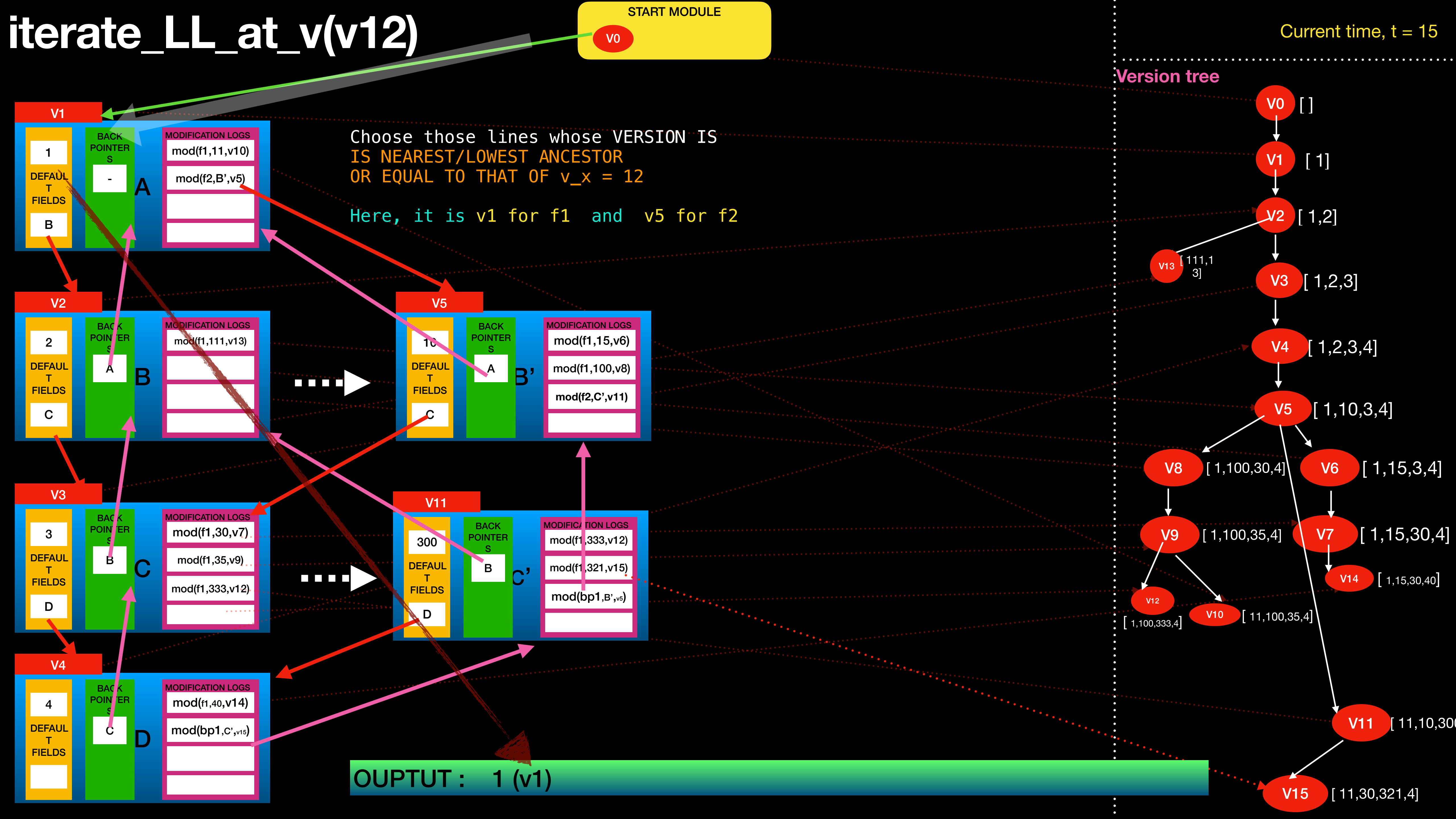
Same Way we can show

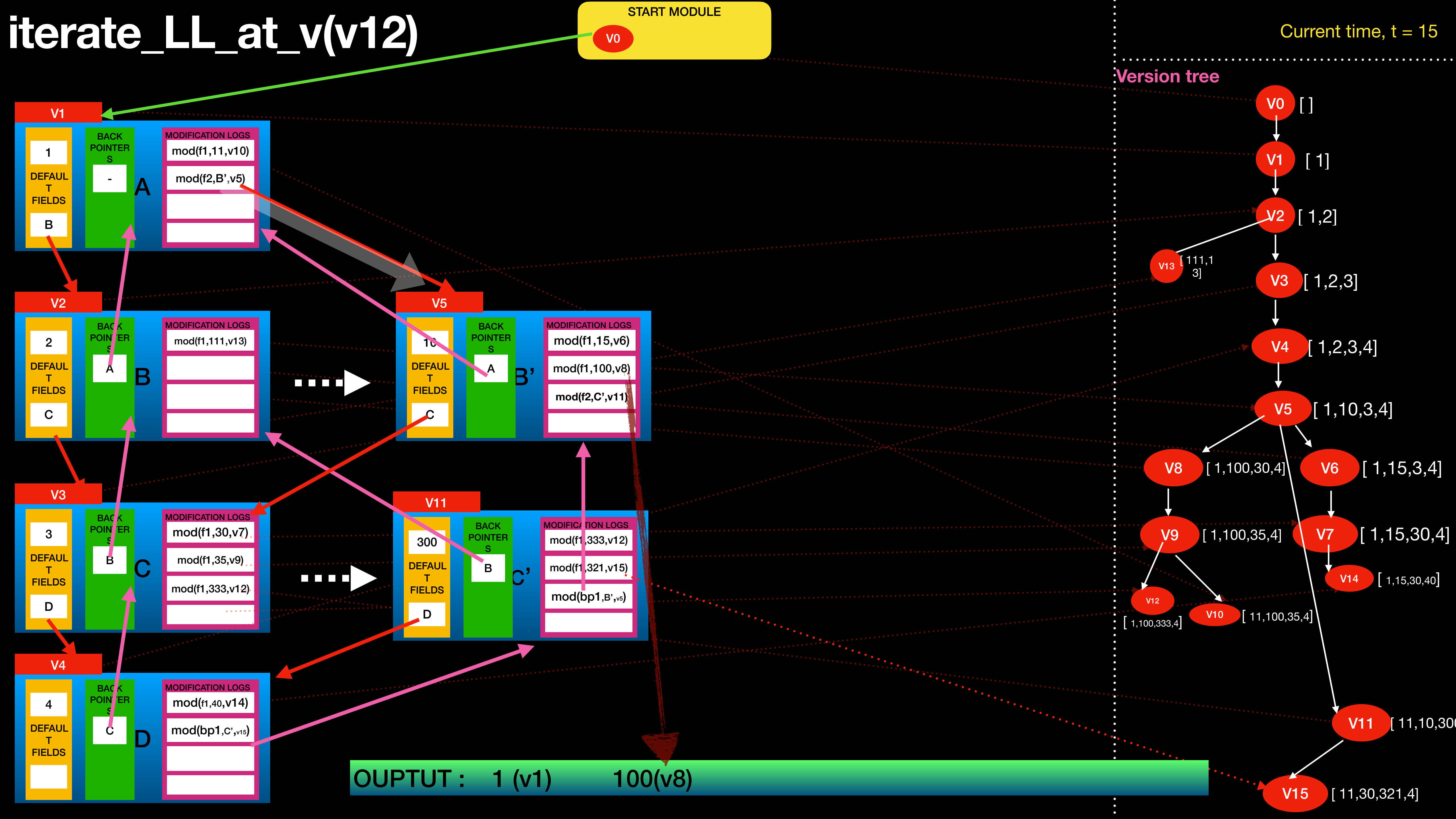
- > Deletion of Node
- > Insertion of Node

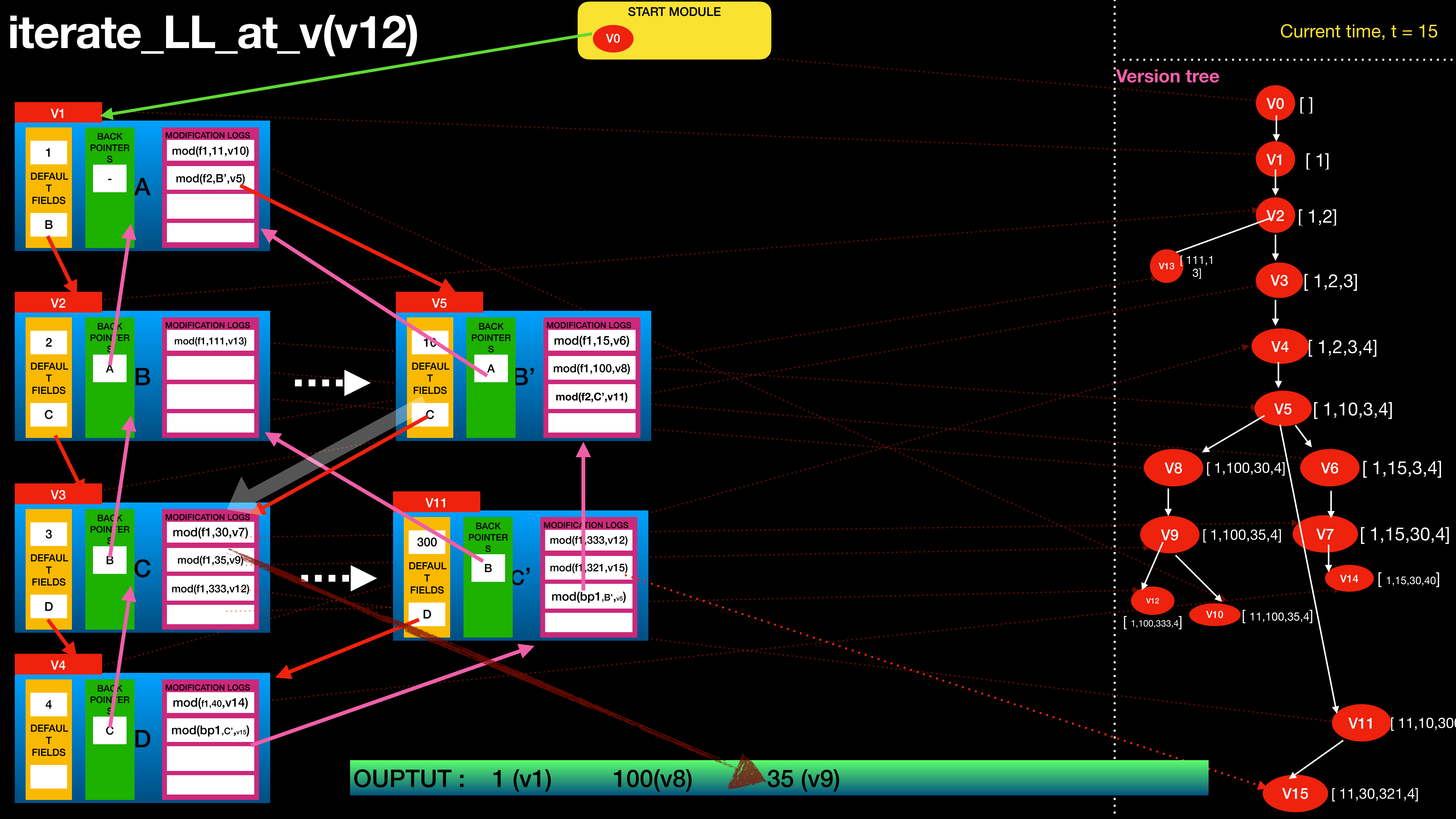
remove(x,v)
add(x, y, v)

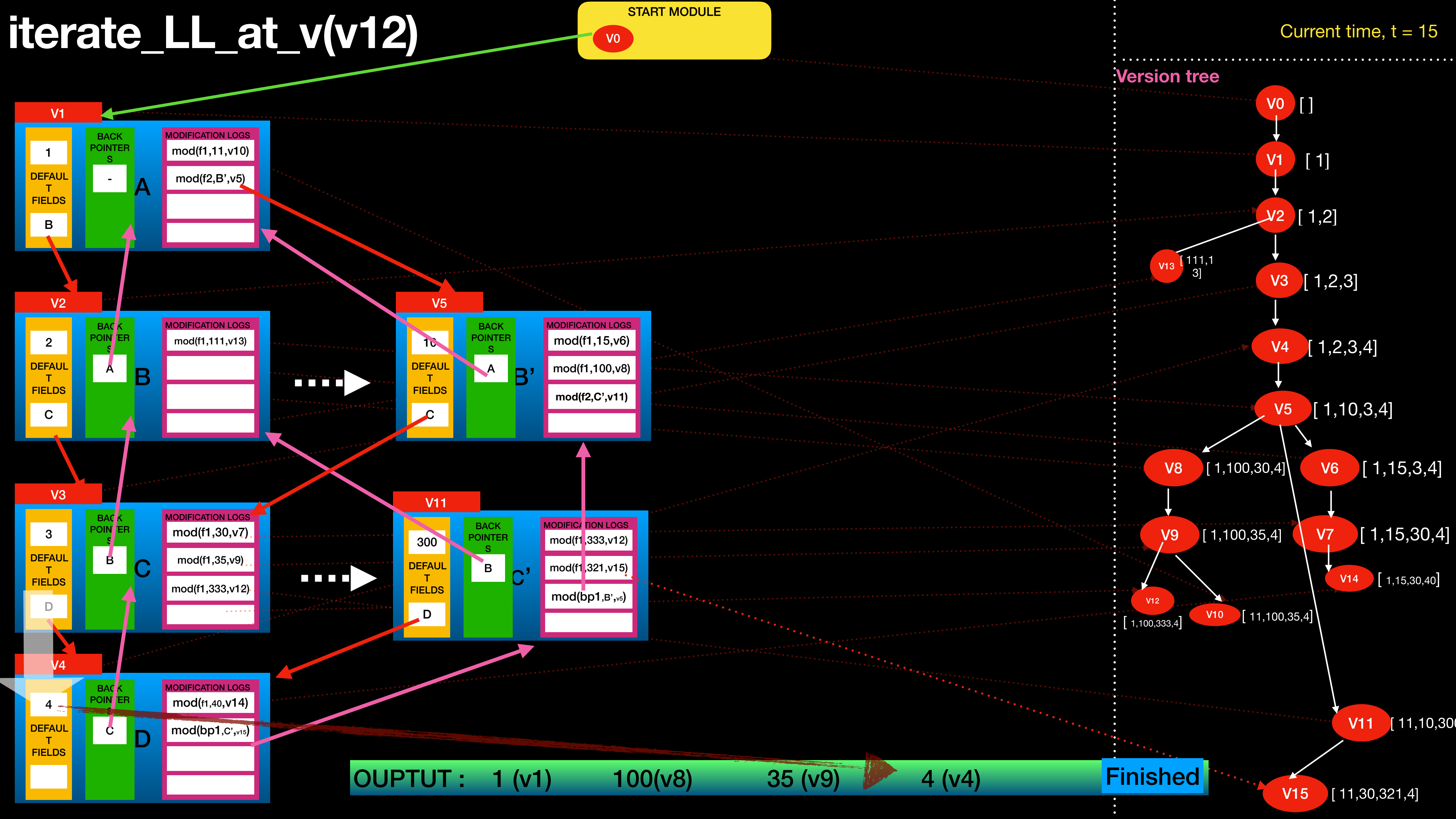
Shown in partial persistent mode

iterate_LL_at_v(v12)









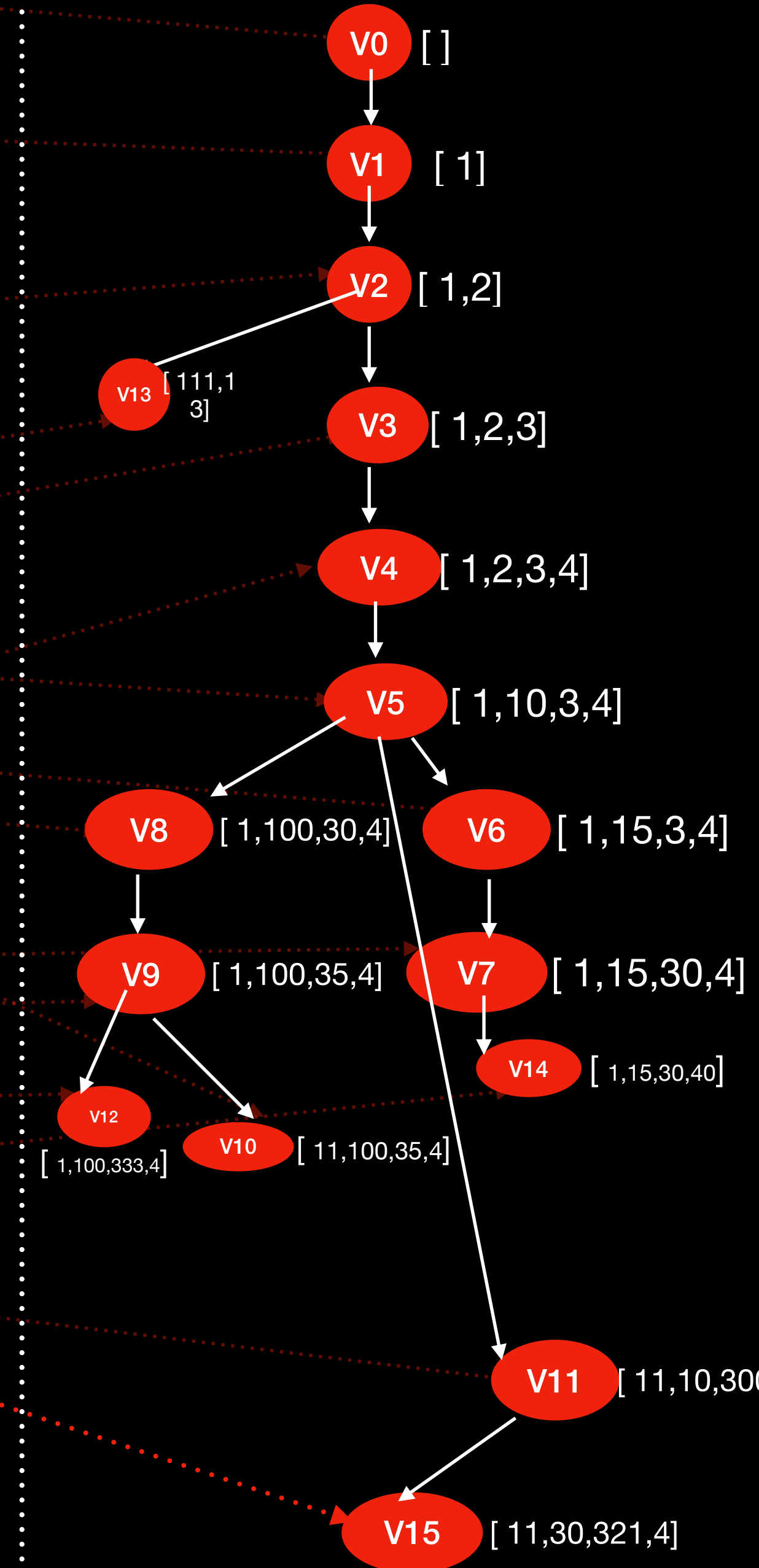
Final Note

START MODULE

V0

Current time, $t = 15$

Version tree



Nothing At BP Means pointing to address of Start Module

**NOTE: HERE ARE NO
READ-ONLY NODE**