Partial Persistent Data-structure

Pointer Machine Model

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

Ephemeral Linked List

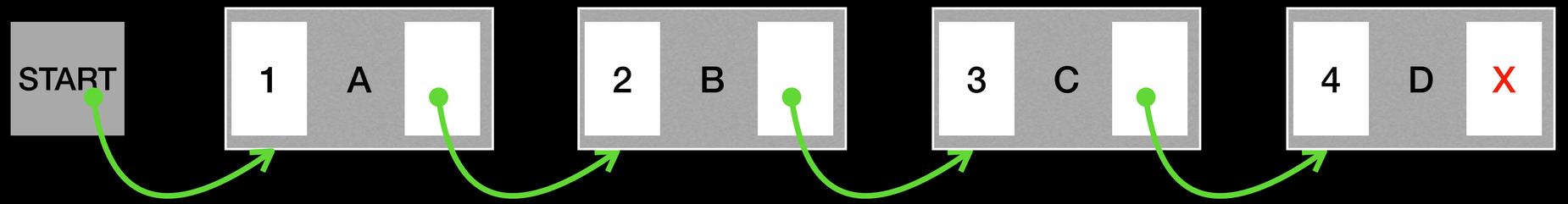
Operations:

- start = init() : To initiate linked list and “start” pointer holds the starting position
- add(x, y) : Add new node x after y
- remove(x) : Remove node x
- iterate_over_LL() : Iterate over the whole linked list
- update(f_i,x,val) : Update the i-th field in node x to new value ‘val’

Ephemeral Linkedlist

Node structure

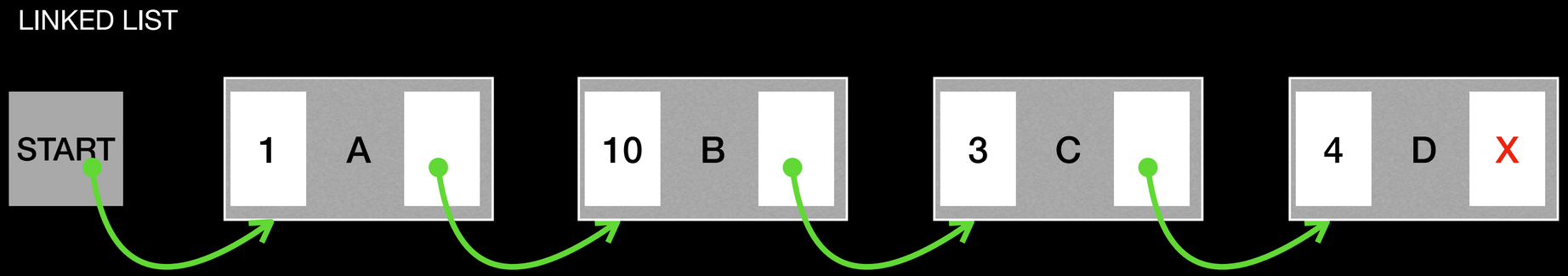
LINKED LIST



HERE IN-DEGREE OF A NODE IS <= 1 AND OUT-DEGREE OF A NODE IS <=1

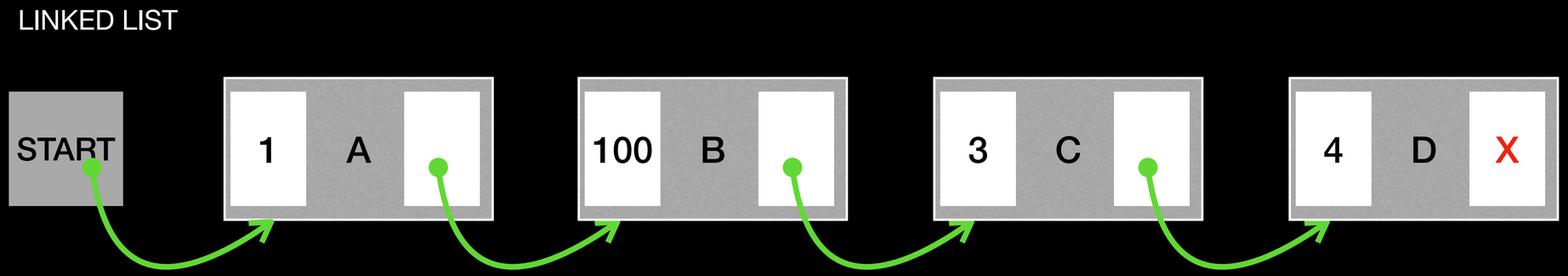
SOME OPERATIONS

update(f1,B,10)



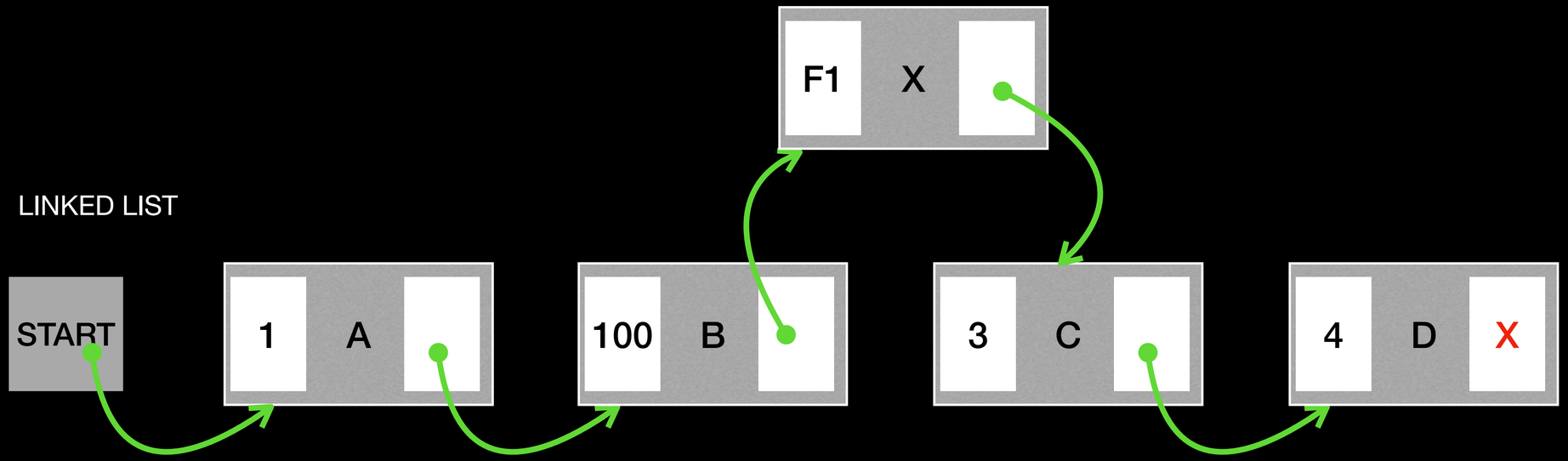
SOME OPERATIONS

update(f1,B,100)



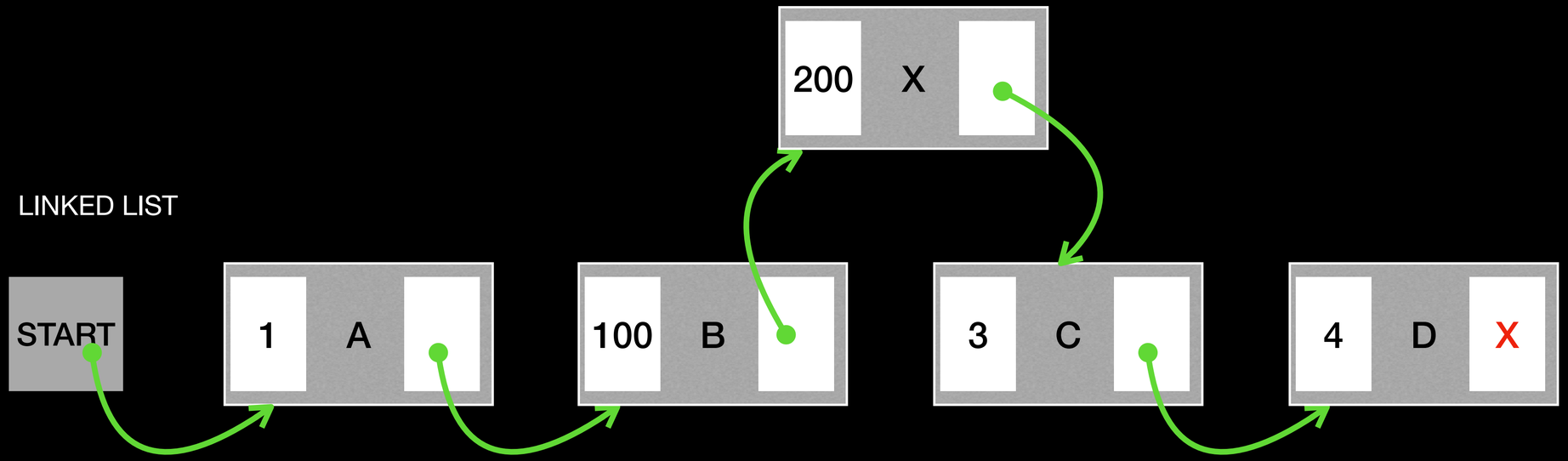
SOME OPERATIONS

add(X,B)



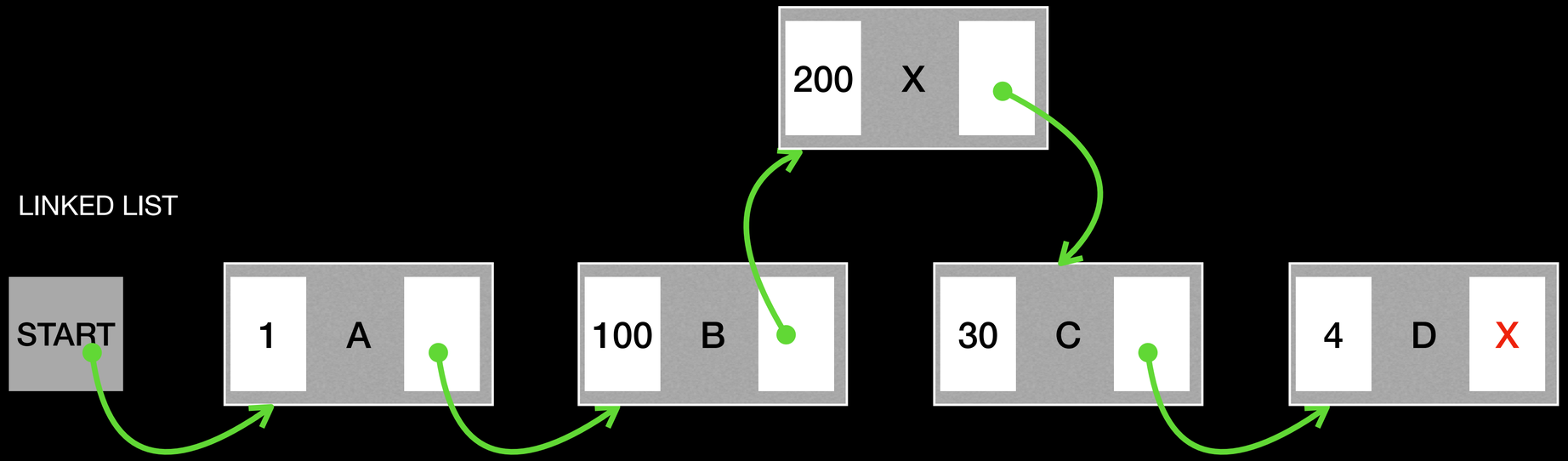
SOME OPERATIONS

update(f1,X,200)



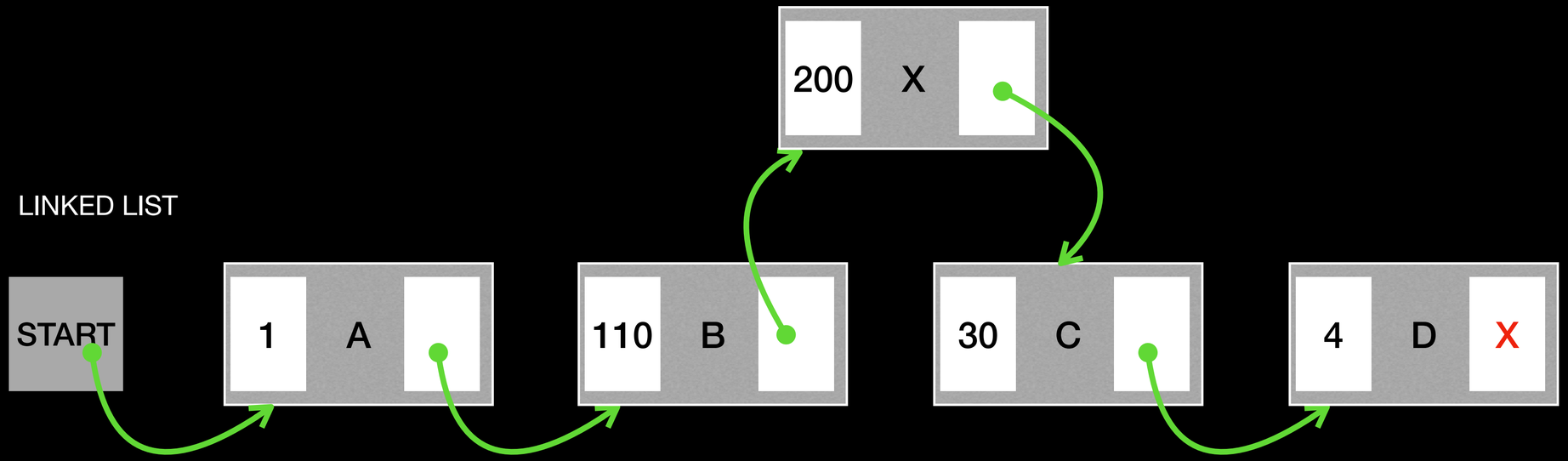
SOME OPERATIONS

update(f1,C,30)



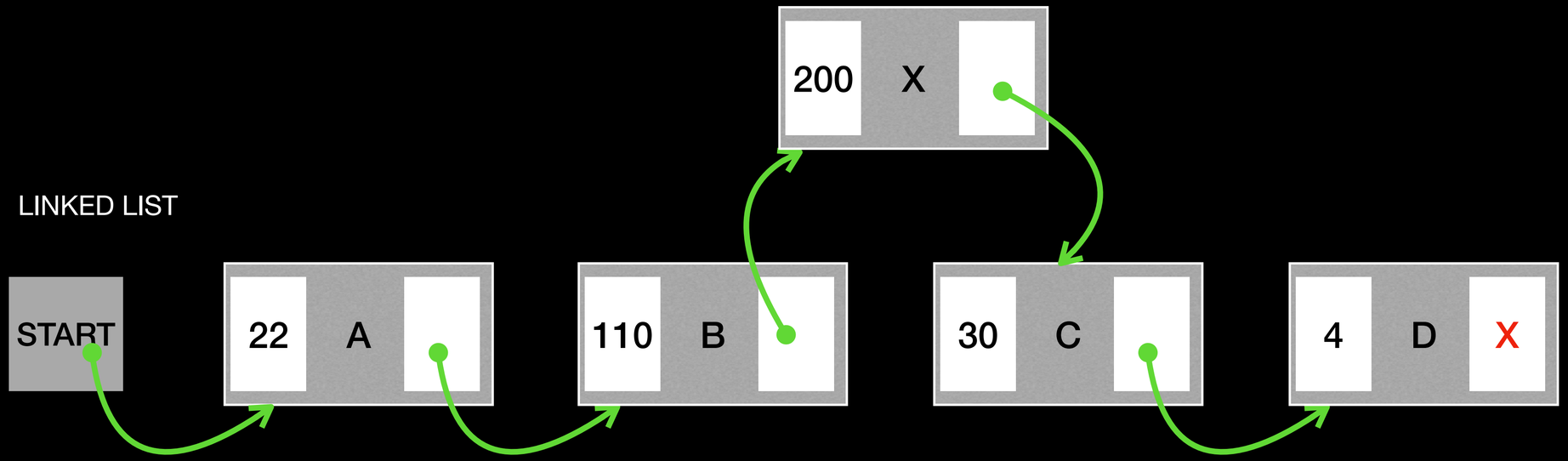
SOME OPERATIONS

update(f1,B,110)



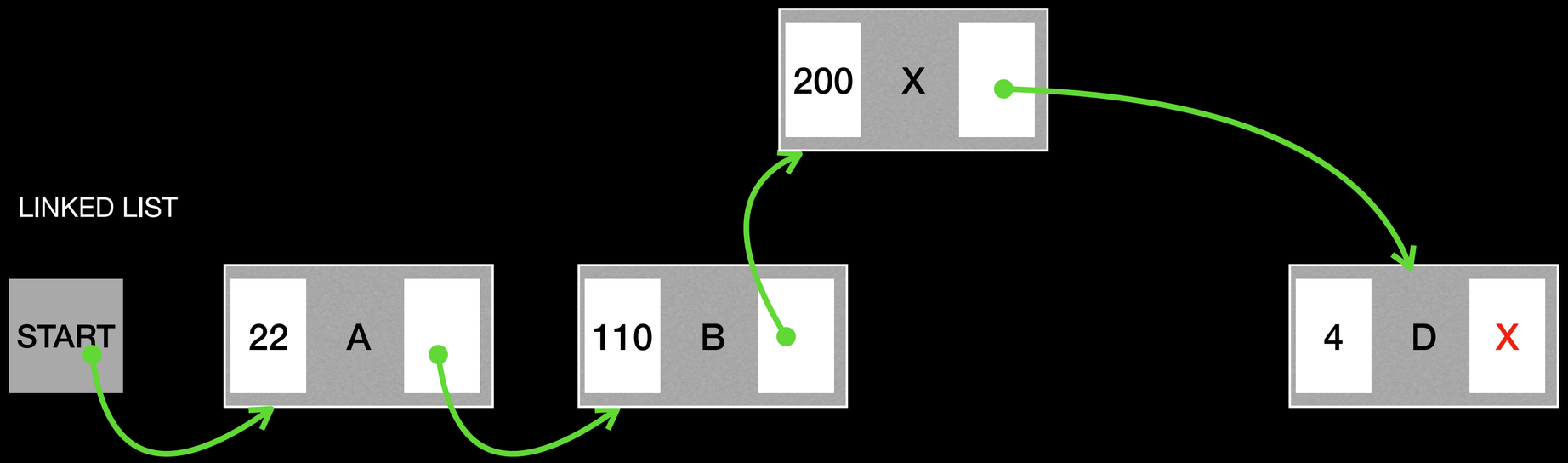
SOME OPERATIONS

update(f1,A,22)



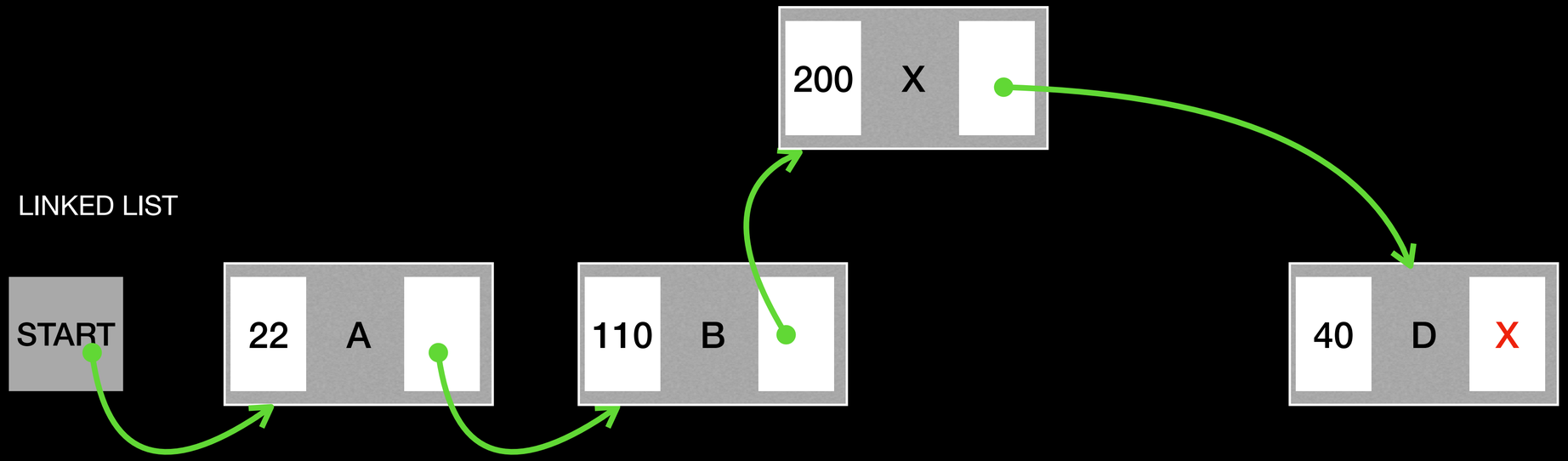
SOME OPERATIONS

remove(C)



SOME OPERATIONS

update(f1,D,40)



Partial Persistent Linked List

Operations:

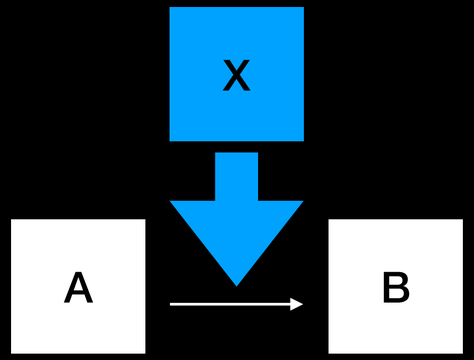
- start = init() : To initiate linked list and “start” pointer holds the starting position in v0
- create_node(x, a): Allocate a new node x with f1 = a and f2 = NULL, and set its default version to current version
- add(x, y, a) : Add new node x with value = a after y and update the version
- remove(x)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

Interesting thing!

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

add(X,C,123) consists of

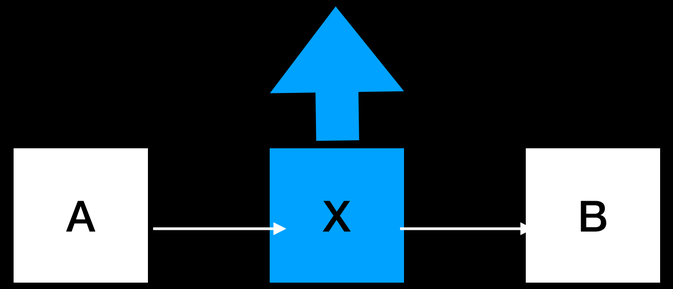
Create node X with value 123
Modify f2 of A to X : update(f2, A, X)
Modify f2 of X to C : update(f2, X, C)
Set BP of X to A: X.bp = A
Set BP of B to X: B.bp = X
~~Set f1 of X(optional)~~



remove(x) consists of

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

update(f2, A, B)
Set BP of B to A: B.bp = A
If all shared reference of X is removed Then free up the memory associated with X

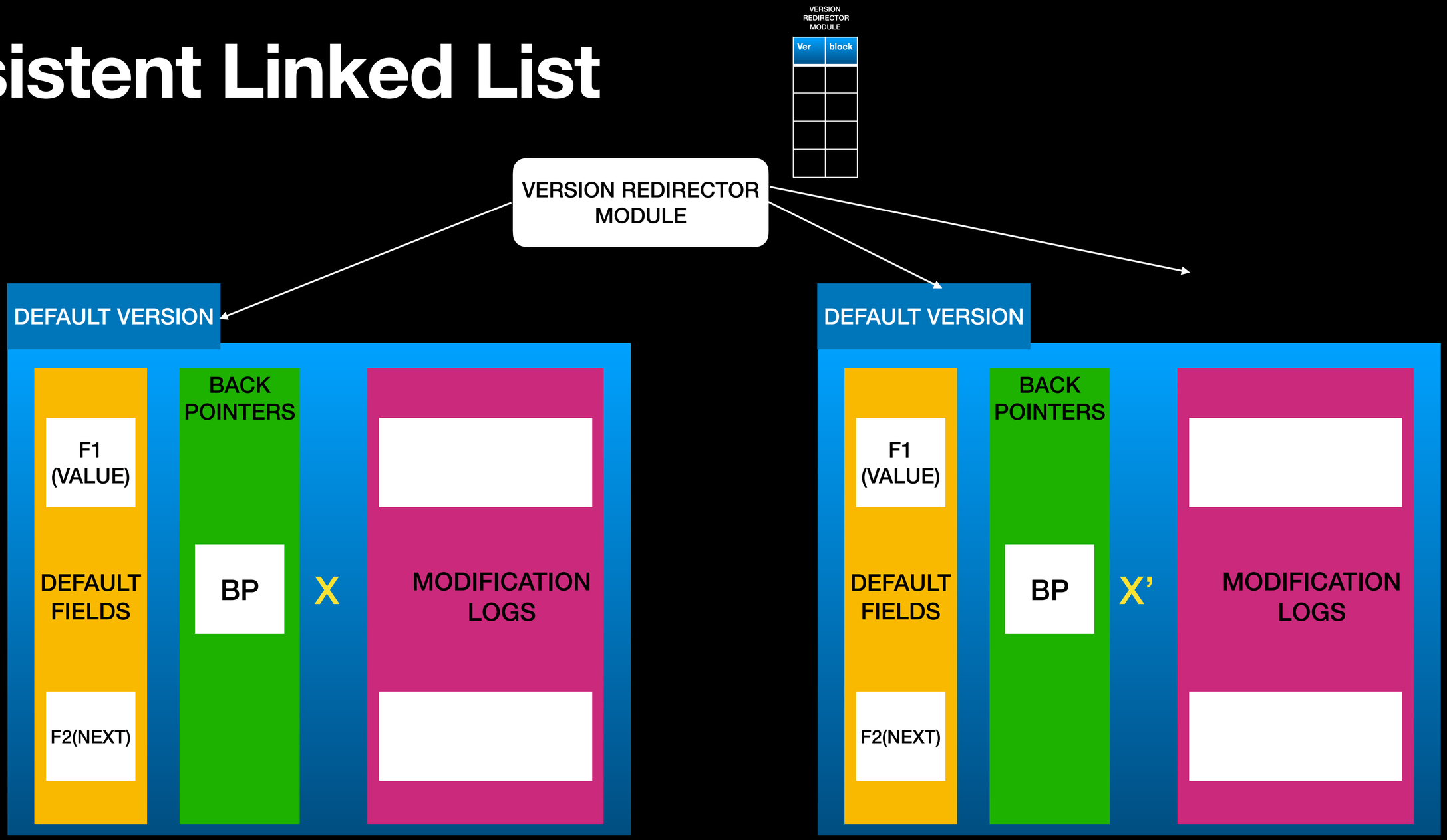


Elementary Operations:

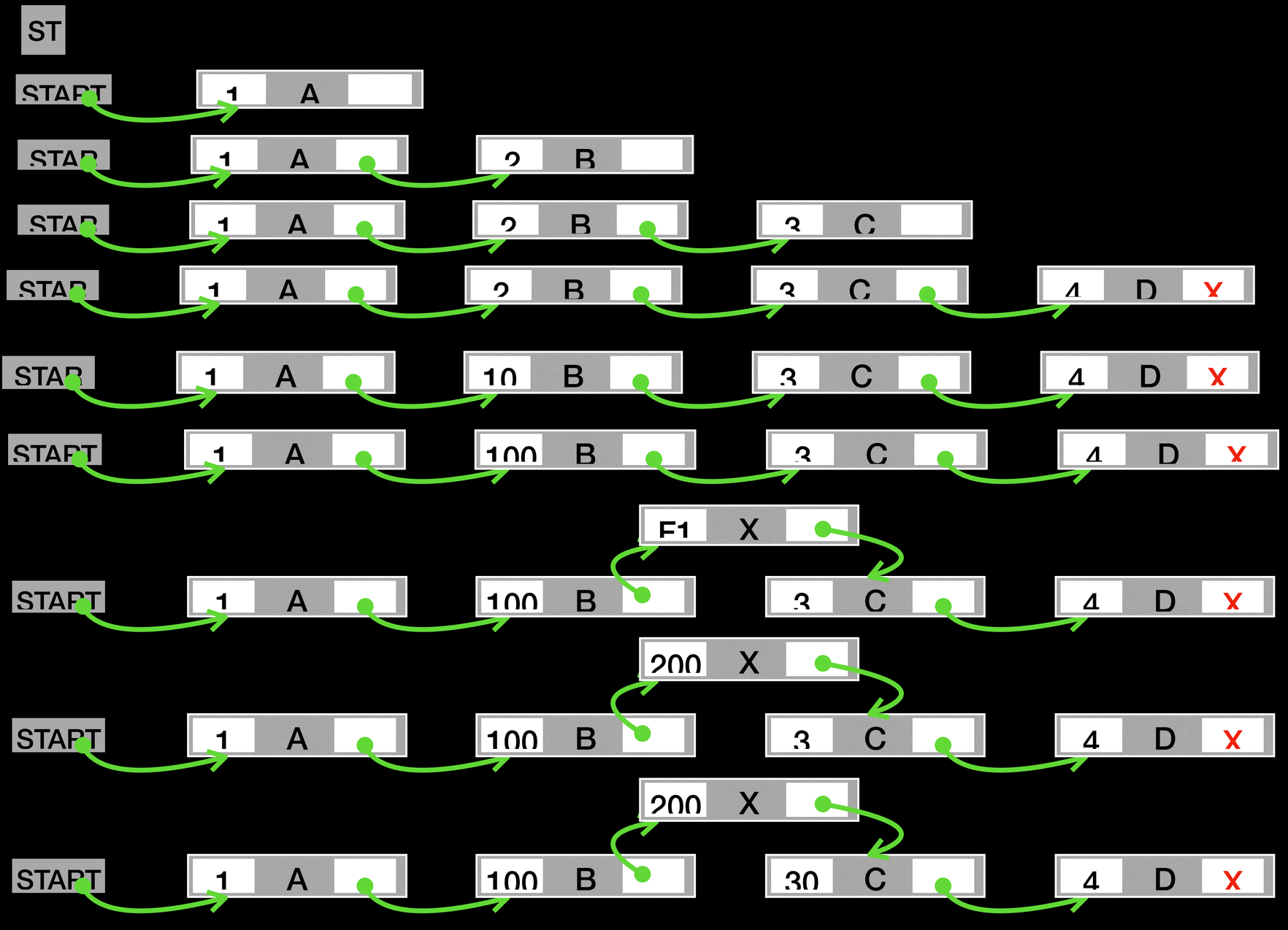
- start = init()
- create_node(x, a)
- iterate_over_LL(v)
- update(f_i,x,val)

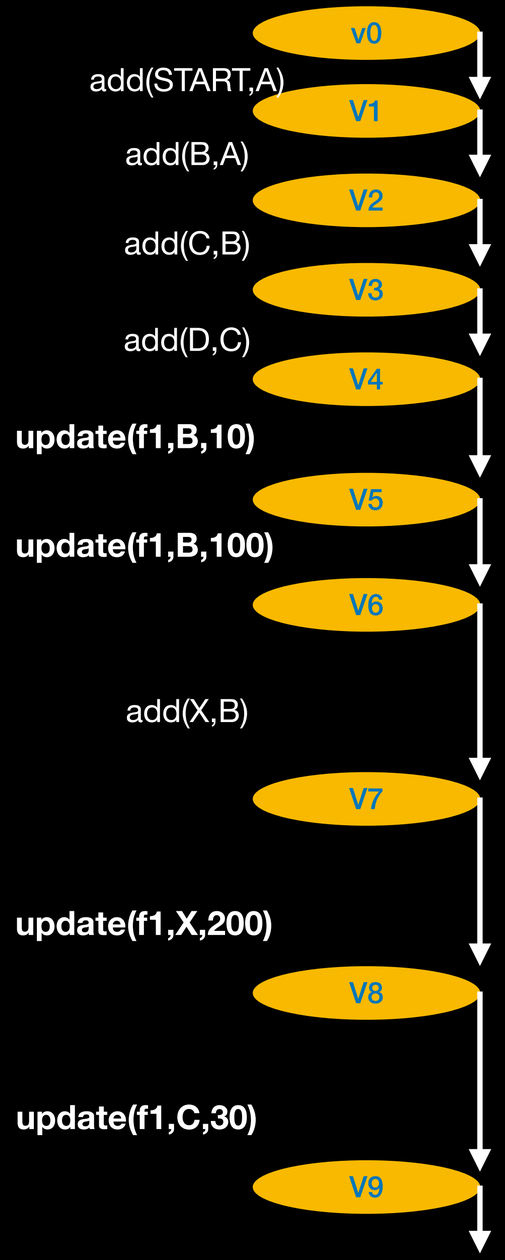
Persistent Linked List

Node

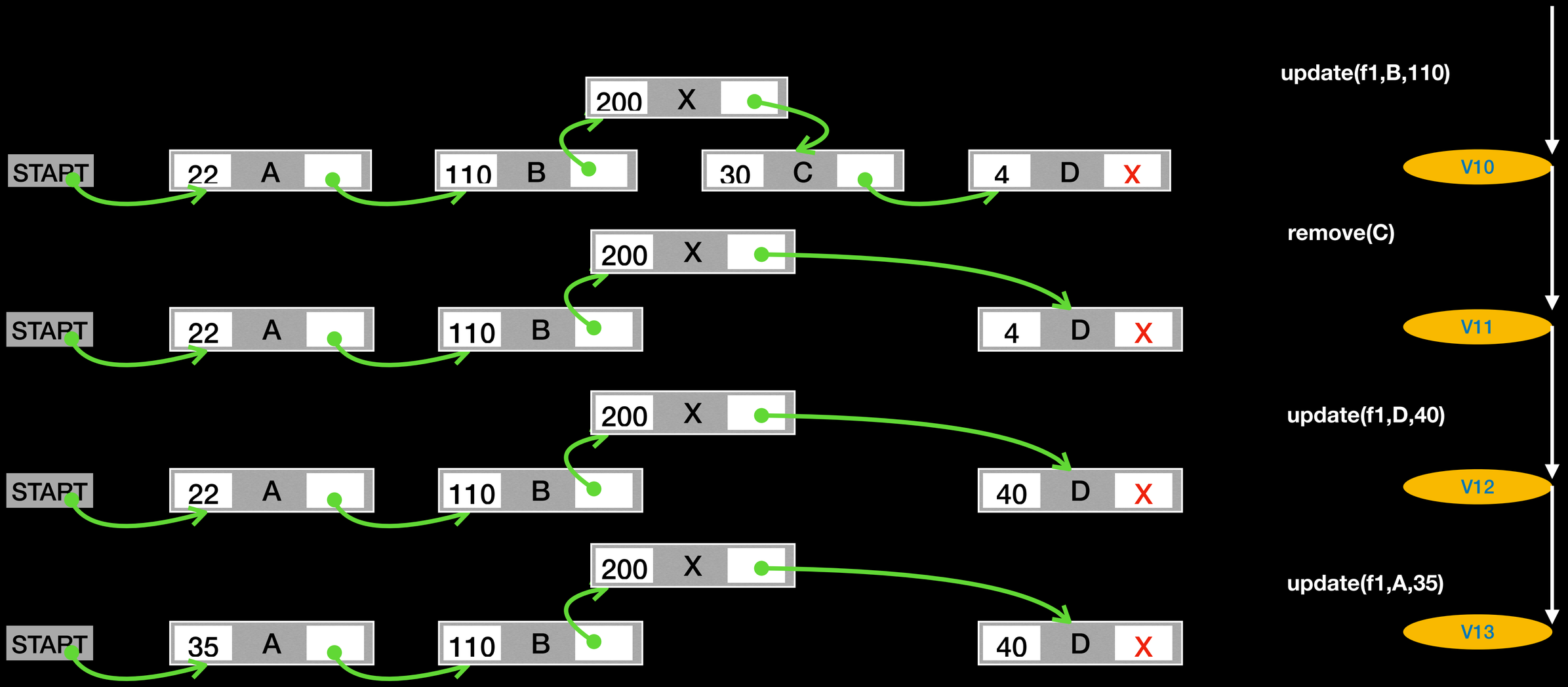


Ephemeral List and Corresponding Versions





Ephemeral List and Corresponding Versions (contd.)



Current Version: v0

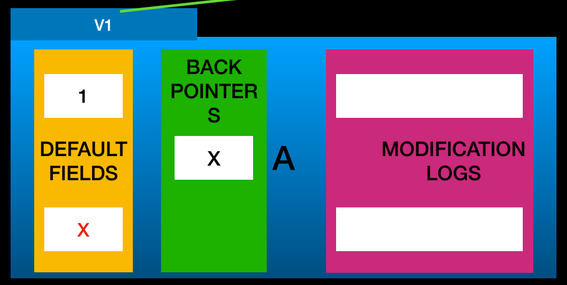
IMPLEMENTATION IN PERSISTENT POINTER MACHINE MODEL

Current Version: v1

START MODULE

V0

add(A,START,1)



VERSION REDIRECTOR MODULE

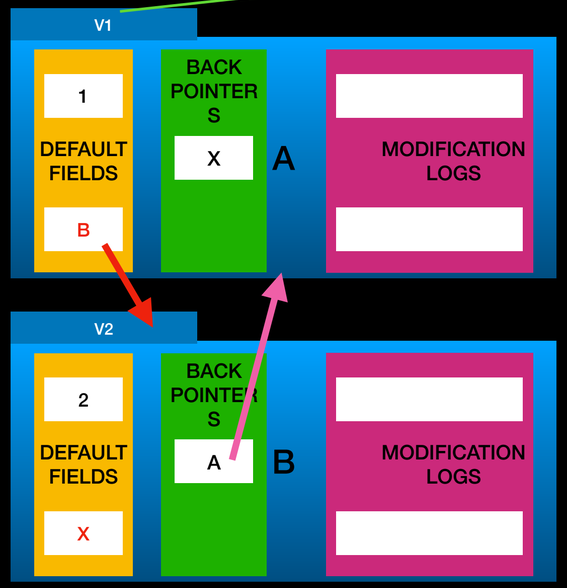
Ver	block
1	A

Current Version: v2

START MODULE

V0

add(B,A,2)



VERSION REDIRECTOR MODULE

Ver	block
1	A

VERSION REDIRECTOR MODULE

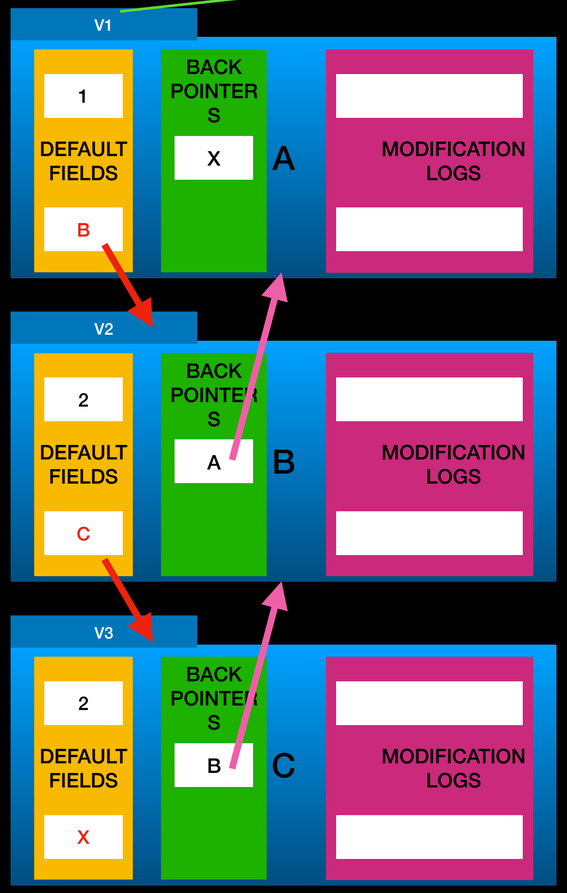
Ver	block
2	B

Current Version: v3

START MODULE

V0

add(C,B,3)



VERSION REDIRECTOR MODULE

Ver	block
1	A

VERSION REDIRECTOR MODULE

Ver	block
2	B

VERSION REDIRECTOR MODULE

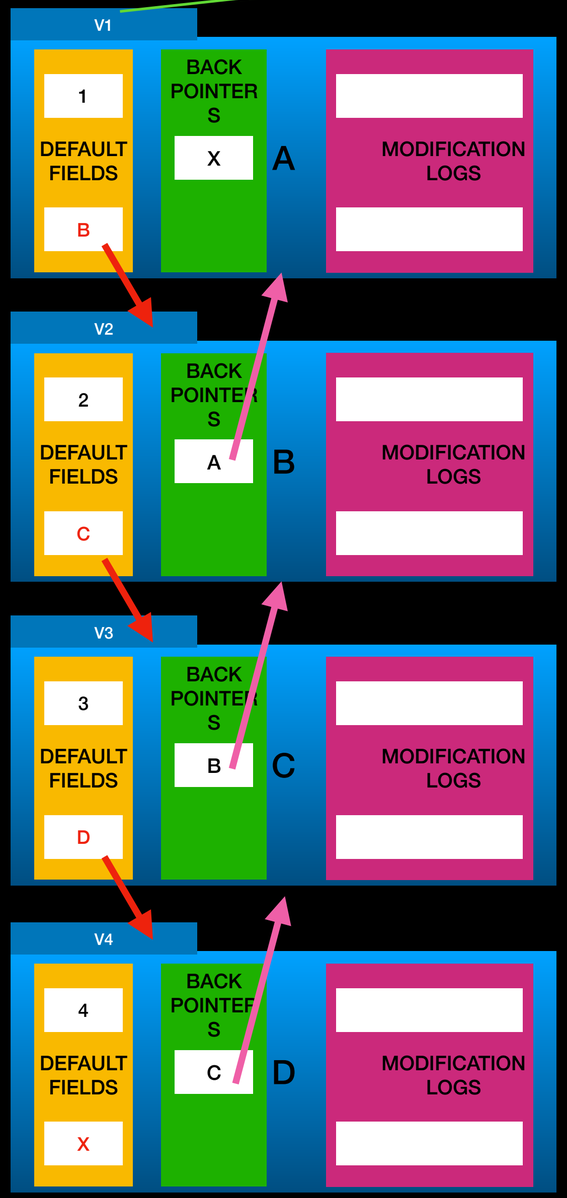
Ver	block
3	C

Current Version: v4

START MODULE

V0

add(D,C,4)



VERSION REDIRECTOR MODULE

Ver	block
1	A

VERSION REDIRECTOR MODULE

Ver	block
2	B

VERSION REDIRECTOR MODULE

Ver	block
3	C

VERSION REDIRECTOR MODULE

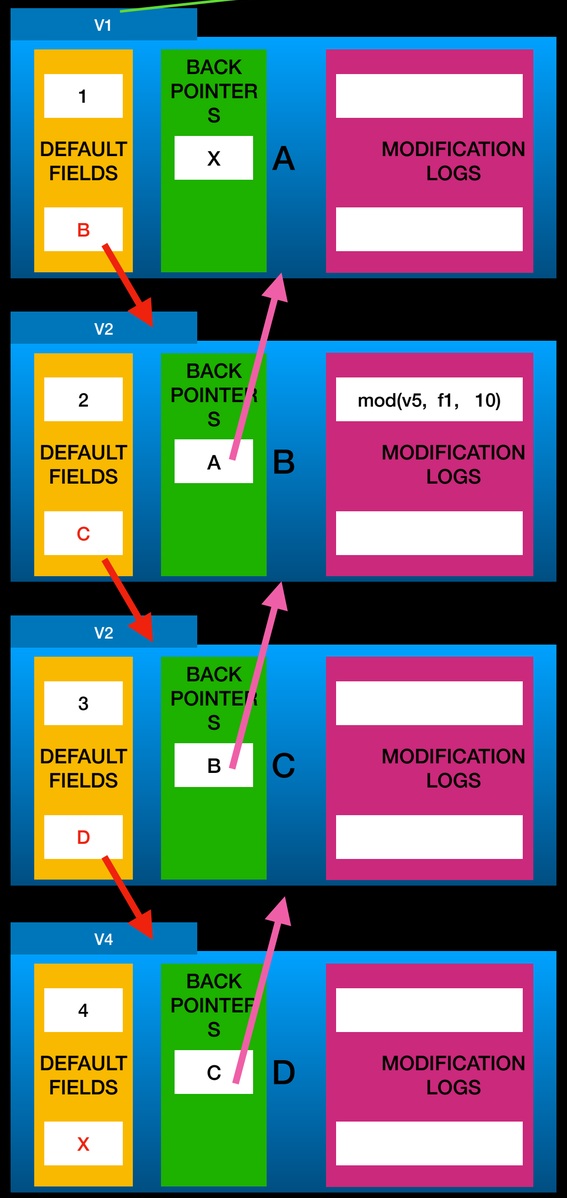
Ver	block

Current Version: v5

START MODULE

V0

update(f1,B,10)



VERSION REDIRECTOR MODULE

Ver	block
1	A

VERSION REDIRECTOR MODULE

Ver	block
2	B

VERSION REDIRECTOR MODULE

Ver	block
3	C

VERSION REDIRECTOR MODULE

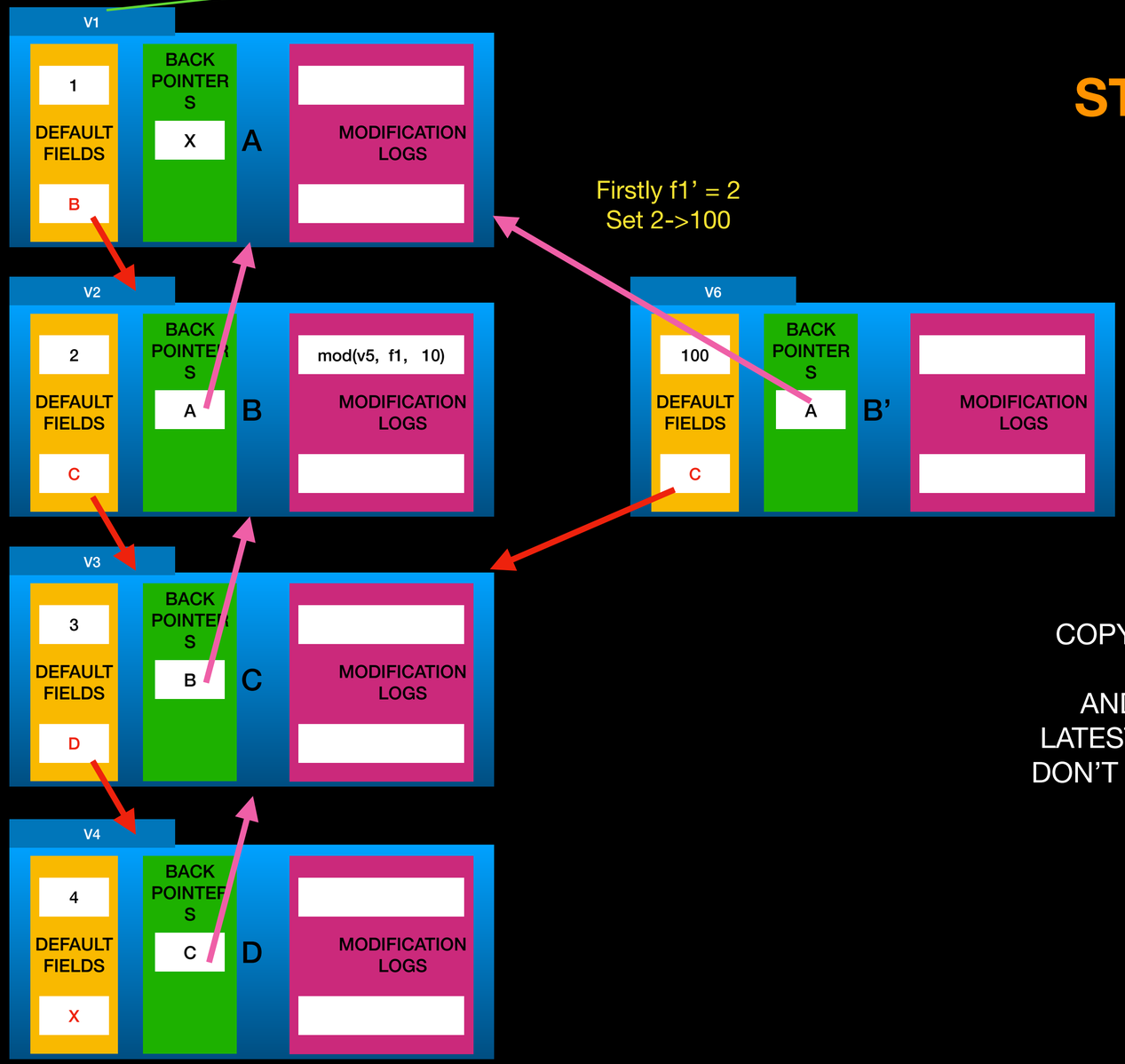
Ver	block
4	D

Current Version: v6

START MODULE

V0

update(f1,B,100)



Firstly f1' = 2
Set 2->100

STEP 1

COPY THE WHOLE NODE
AND KEEP THE LATEST VALUES AND
DON'T COPY MODLOG

VERSION REDIRECTOR MODULE

Ver	block
1	A

VERSION REDIRECTOR MODULE

Ver	block
2	B
6	B'

VERSION REDIRECTOR MODULE

Ver	block
3	C

VERSION REDIRECTOR MODULE

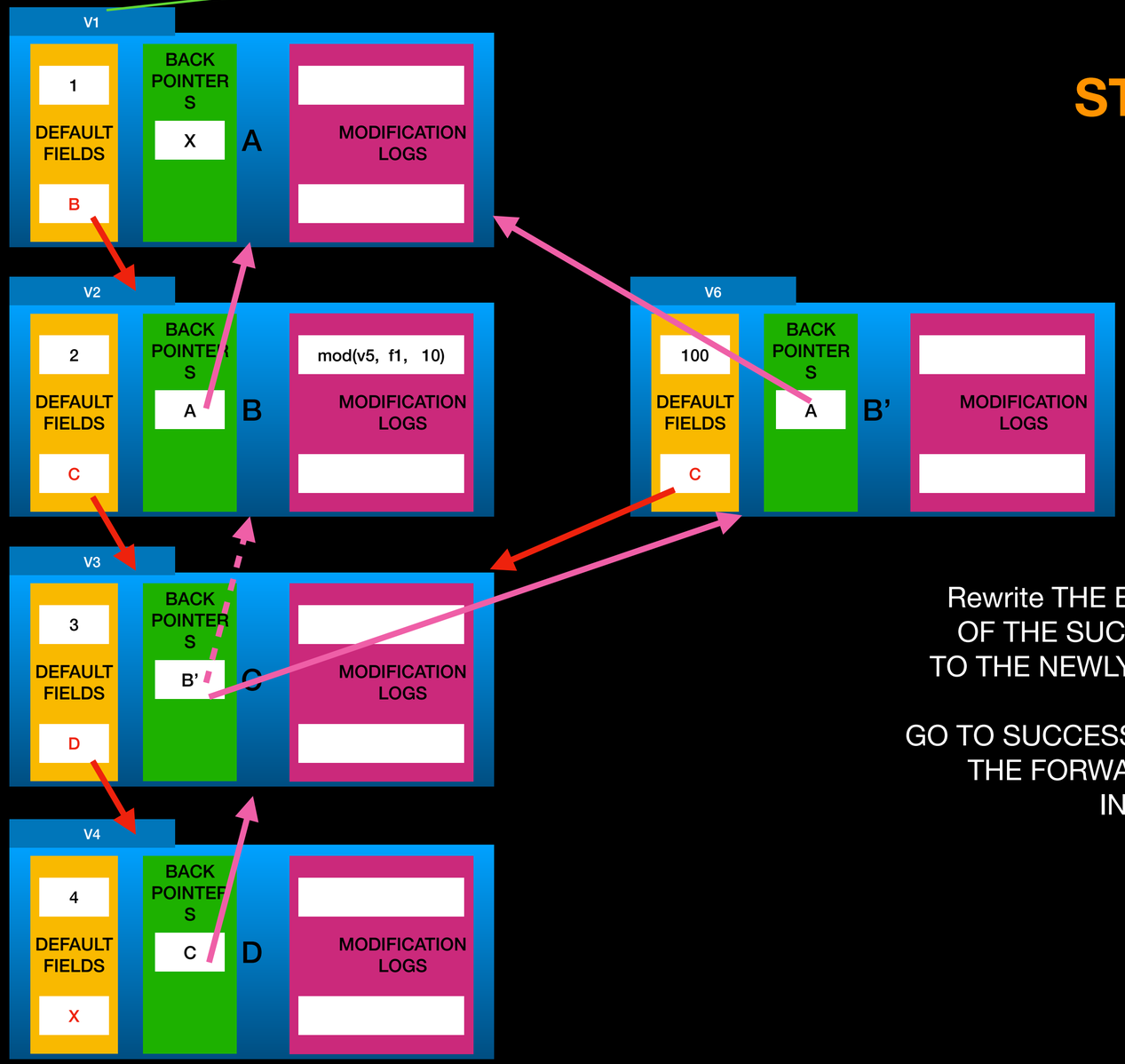
Ver	block
4	D

Current Version: v6

START MODULE

V0

update(f1,B,100)



STEP 2

Rewrite THE BACK POINTERS
OF THE SUCCESSOR NODE
TO THE NEWLY CREATED NODE

GO TO SUCCESSOR NODES USING
THE FORWARD POINTERS
IN O(1)

VERSION REDIRECTOR MODULE

Ver	block
1	A

VERSION REDIRECTOR MODULE

Ver	block
2	B
6	B'

VERSION REDIRECTOR MODULE

Ver	block
3	C

VERSION REDIRECTOR MODULE

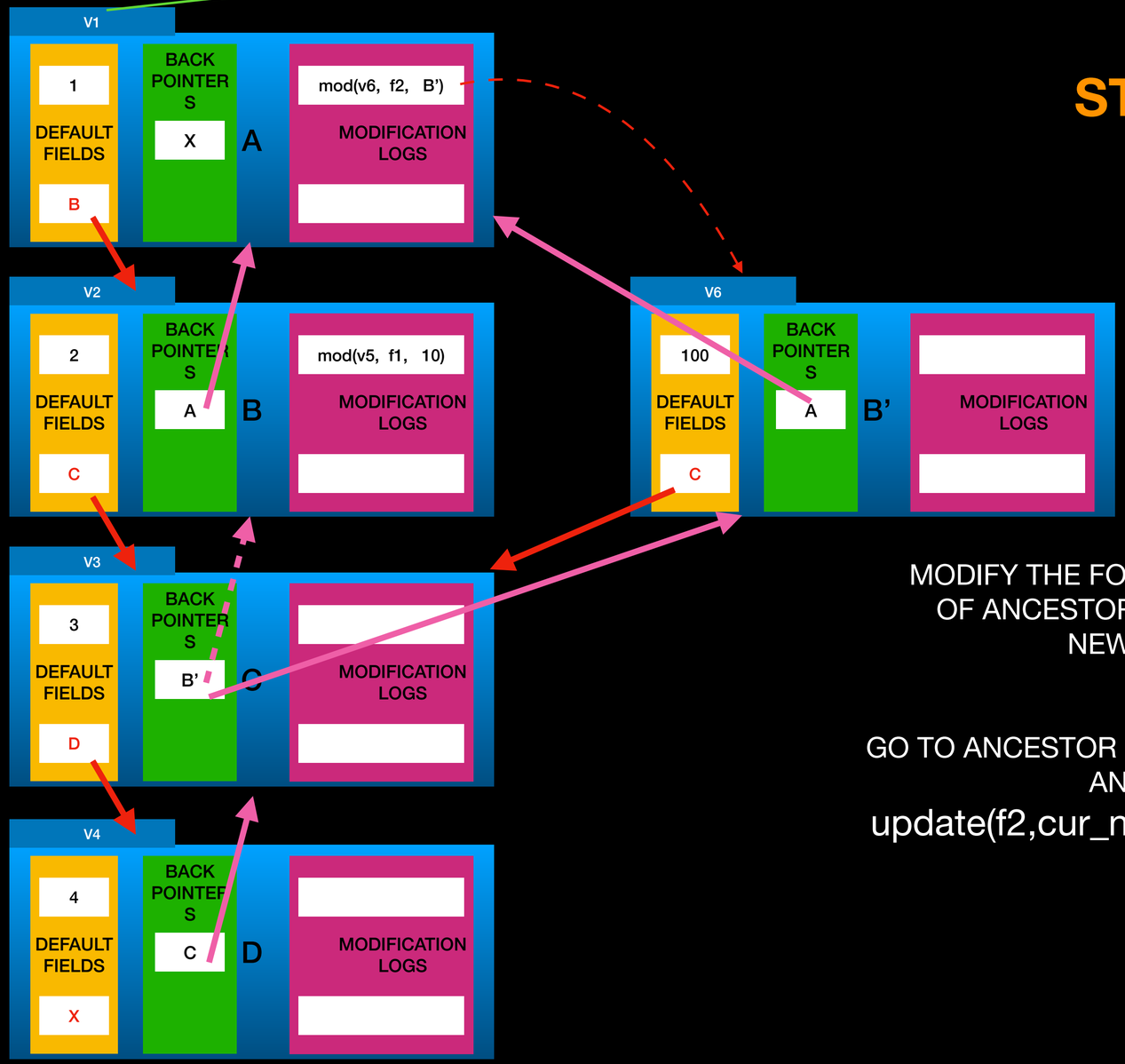
Ver	block
4	D

Current Version: v6

START MODULE

V0

update(f1,B,100)



STEP 3

MODIFY THE FORWARD POINTERS OF ANCESTOR NODES TO THE NEW NODE

GO TO ANCESTOR NODES RECURSIVELY AND DO

update(f2,cur_node,NEW_NODE)

VERSION REDIRECTOR MODULE

Ver	block
1	A

VERSION REDIRECTOR MODULE

Ver	block
2	B
6	B'

VERSION REDIRECTOR MODULE

Ver	block
3	C

VERSION REDIRECTOR MODULE

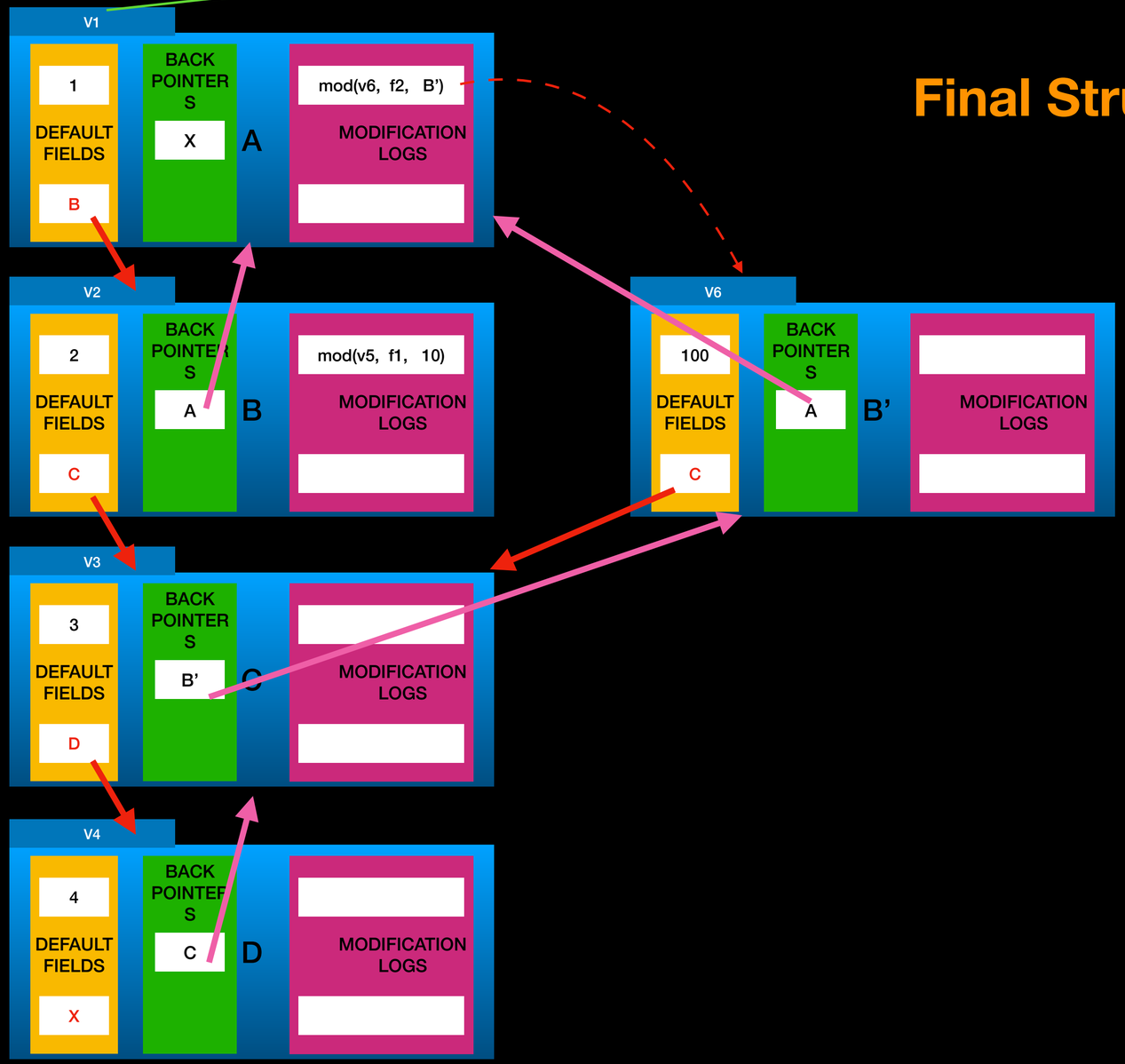
Ver	block
4	D

Current Version: v6

START MODULE

V0

update(f1,B,100)



Final Structure of v6

VERSION REDIRECTOR MODULE

Ver	block
1	A

VERSION REDIRECTOR MODULE

Ver	block
2	B
6	B'

VERSION REDIRECTOR MODULE

Ver	block
3	C

VERSION REDIRECTOR MODULE

Ver	block
4	D

Current Version: v7

START MODULE

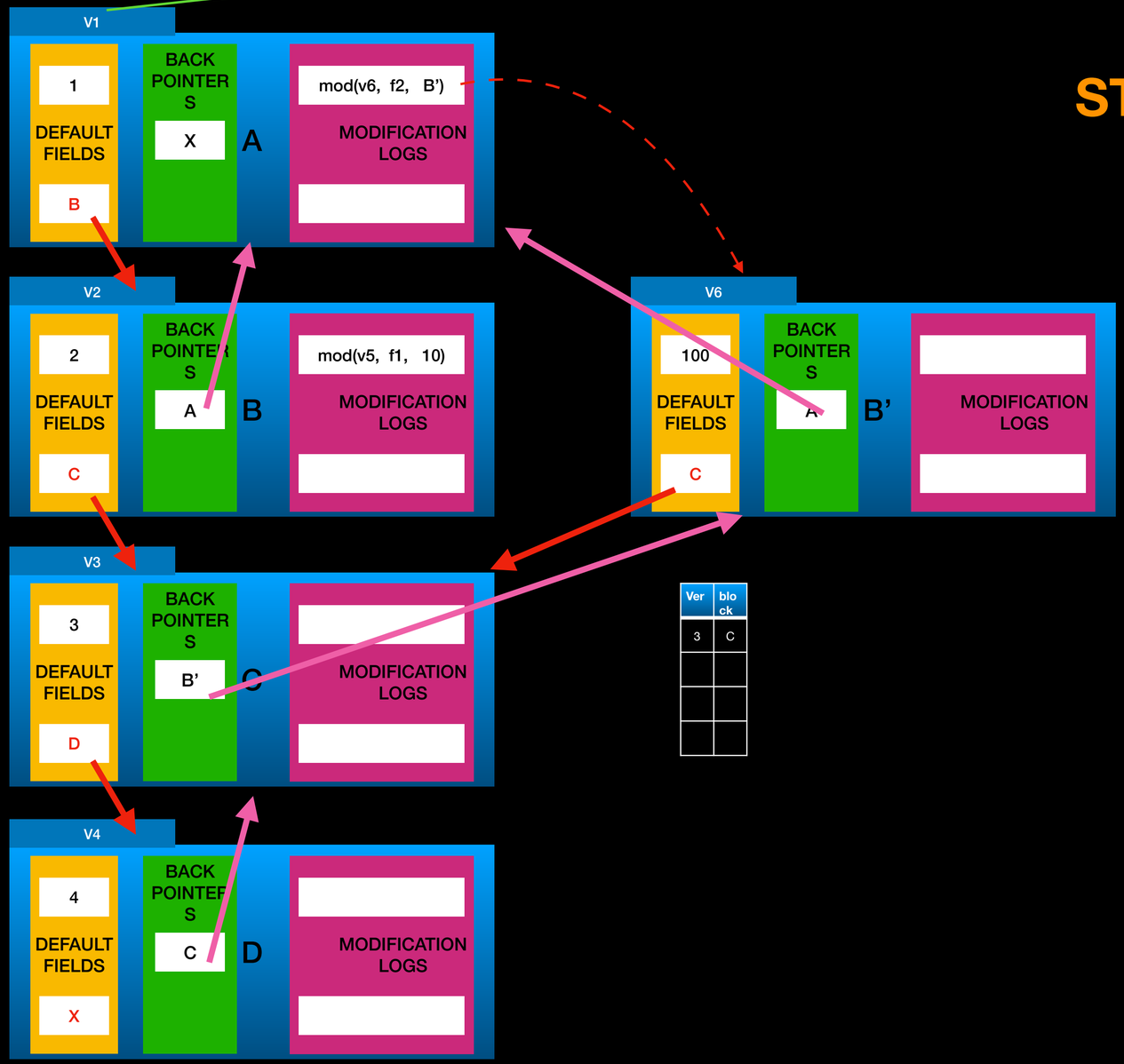
V0

add(X,B,_)

add(X,B,_) consists of

Create node X
Modify f2 of B’ to X
Modify f2 of X to C + Set BP of X to B’
Update BP of C to X
Set f1 of X

STEP 1



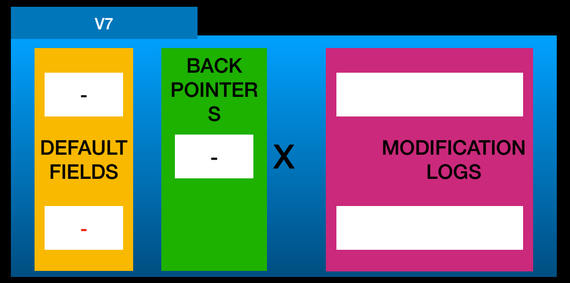
Ver	block
3	C

VERSION REDIRECTOR MODULE

Ver	block
1	A

VERSION REDIRECTOR MODULE

Ver	block
2	B
6	B’



VERSION REDIRECTOR MODULE

Ver	block
7	X

VERSION REDIRECTOR MODULE

Ver	block
4	D

Current Version: v7

START MODULE

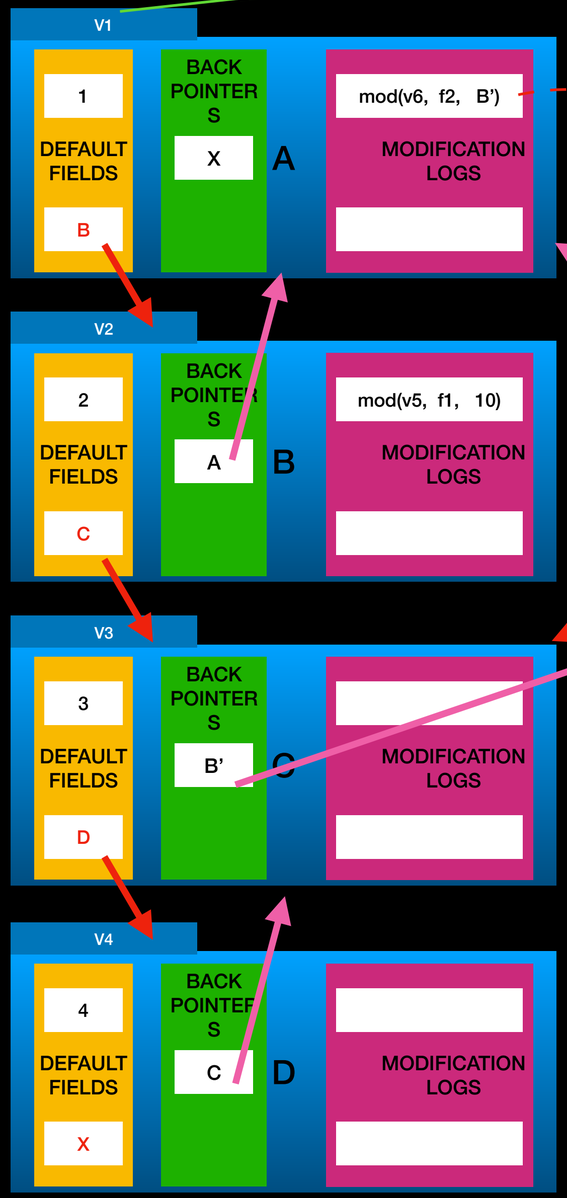
V0

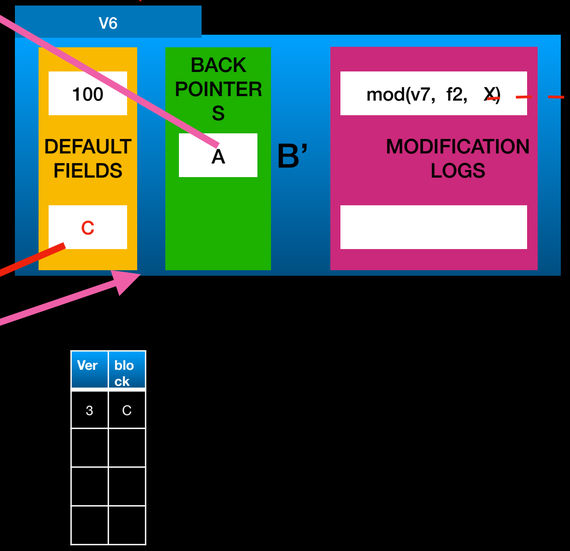
add(X,B,_)

add(X,B,_) consists of

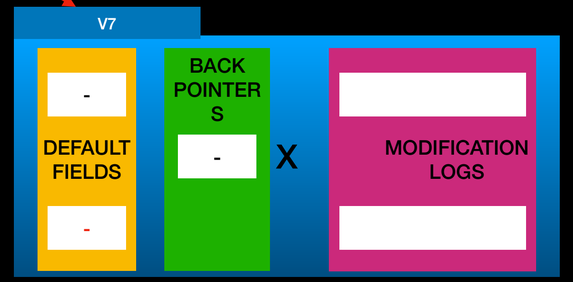
Create node X
Modify f2 of B’ to X
Modify f2 of X to C + Set BP of X to B’
Update BP of C to X
Set f1 of X

STEP 2





Ver	block
3	C



VERSION REDIRECTOR MODULE

Ver	block
1	A

VERSION REDIRECTOR MODULE

Ver	block
2	B
6	B’

VERSION REDIRECTOR MODULE

Ver	block
7	X

VERSION REDIRECTOR MODULE

Ver	block
4	D

Current Version: v7

START MODULE

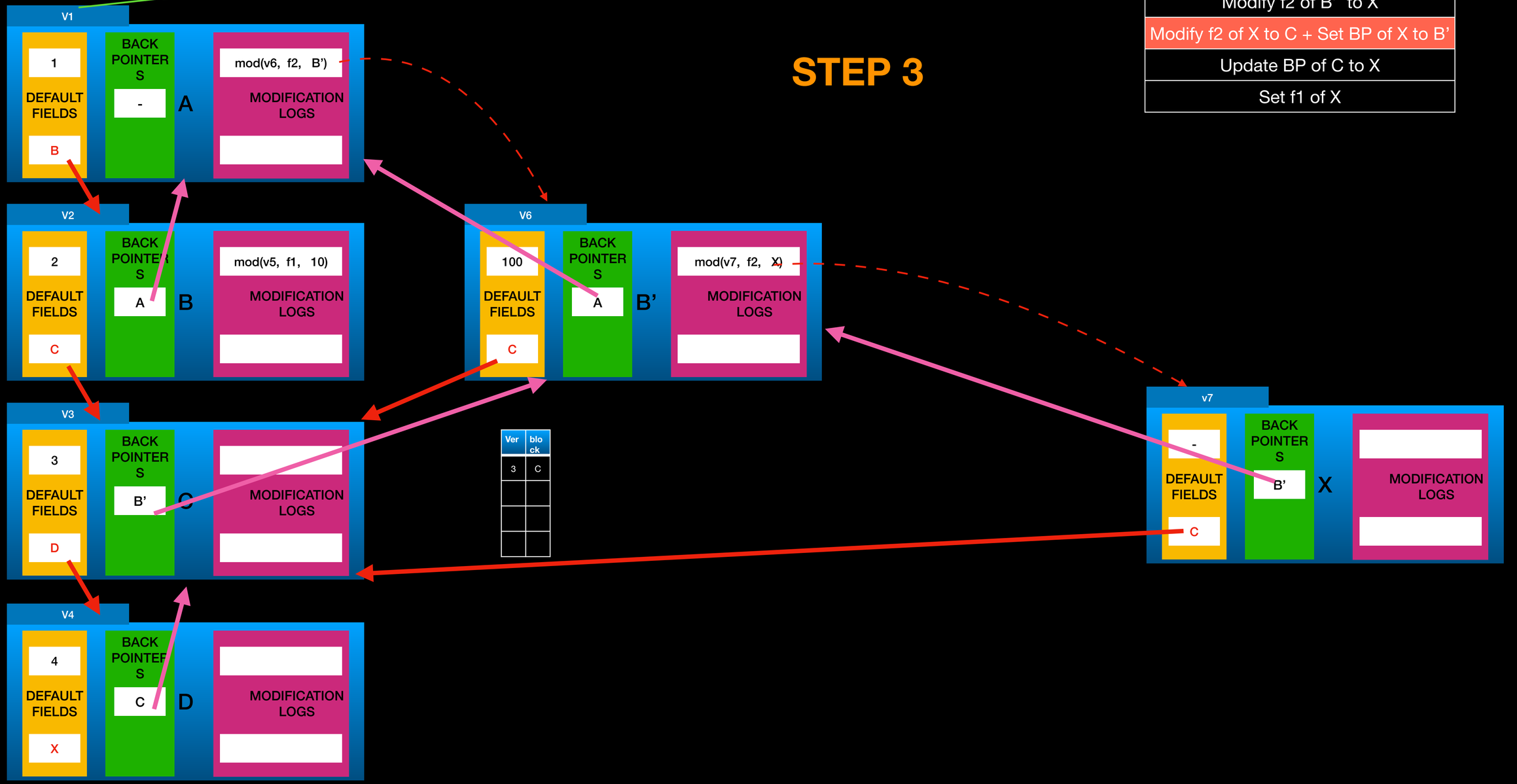
V0

add(X,B,_) consists of

Create node X
Modify f2 of B’ to X
Modify f2 of X to C + Set BP of X to B’
Update BP of C to X
Set f1 of X

add(X,B,_)

STEP 3



Ver	block
3	C

VERSION REDIRECTOR MODULE

Ver	block
1	A

VERSION REDIRECTOR MODULE

Ver	block
2	B
6	B’

VERSION REDIRECTOR MODULE

Ver	block
7	X

VERSION REDIRECTOR MODULE

Ver	block
4	D

Current Version: v7

START MODULE

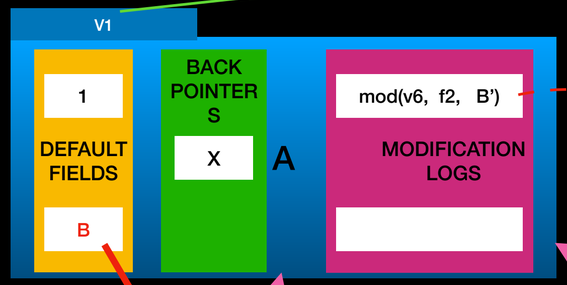
V0

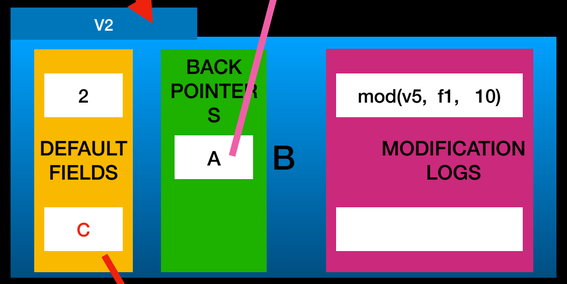
add(X,B,_)

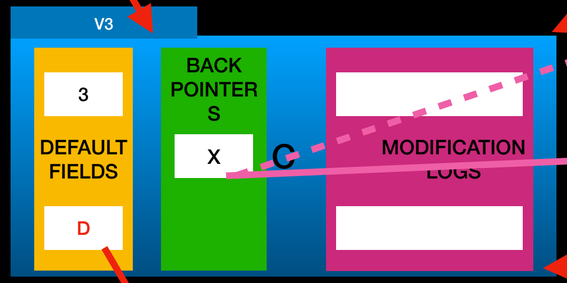
add(X,B,_) consists of

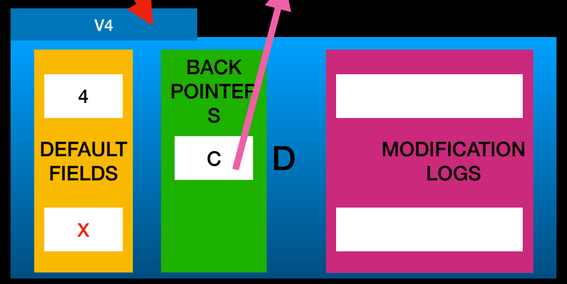
Create node X
Modify f2 of B’ to X
Modify f2 of X to C + Set BP of X to B’
Update BP of C to X
Set f1 of X

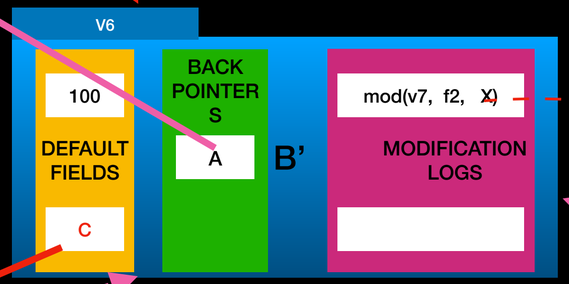
STEP 4



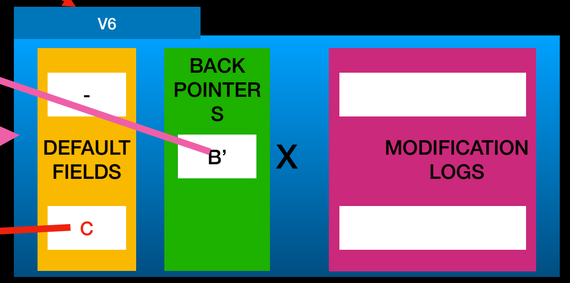








Ver	block
3	C



VERSION REDIRECTOR MODULE

Ver	block
1	A

VERSION REDIRECTOR MODULE

Ver	block
2	B
6	B’

VERSION REDIRECTOR MODULE

Ver	block
7	X

VERSION REDIRECTOR MODULE

Ver	block
4	D

Current Version: v7

START MODULE

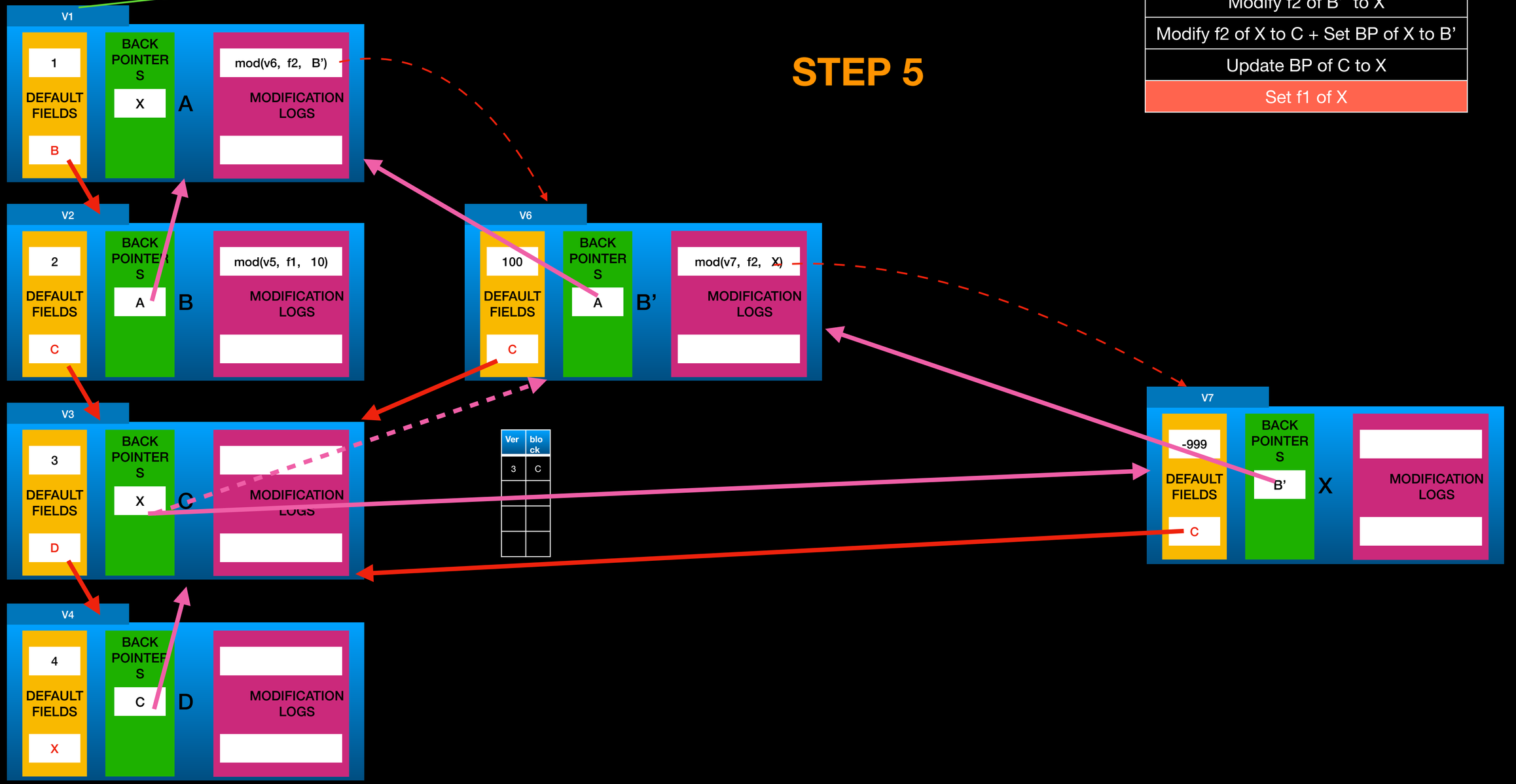
V0

add(X,B,_) consists of

Create node X
Modify f2 of B’ to X
Modify f2 of X to C + Set BP of X to B’
Update BP of C to X
Set f1 of X

add(X,B,_)

STEP 5



Ver	block
3	C

VERSION REDIRECTOR MODULE

Ver	block
1	A

VERSION REDIRECTOR MODULE

Ver	block
2	B
6	B’

VERSION REDIRECTOR MODULE

Ver	block
7	X

VERSION REDIRECTOR MODULE

Ver	block
4	D

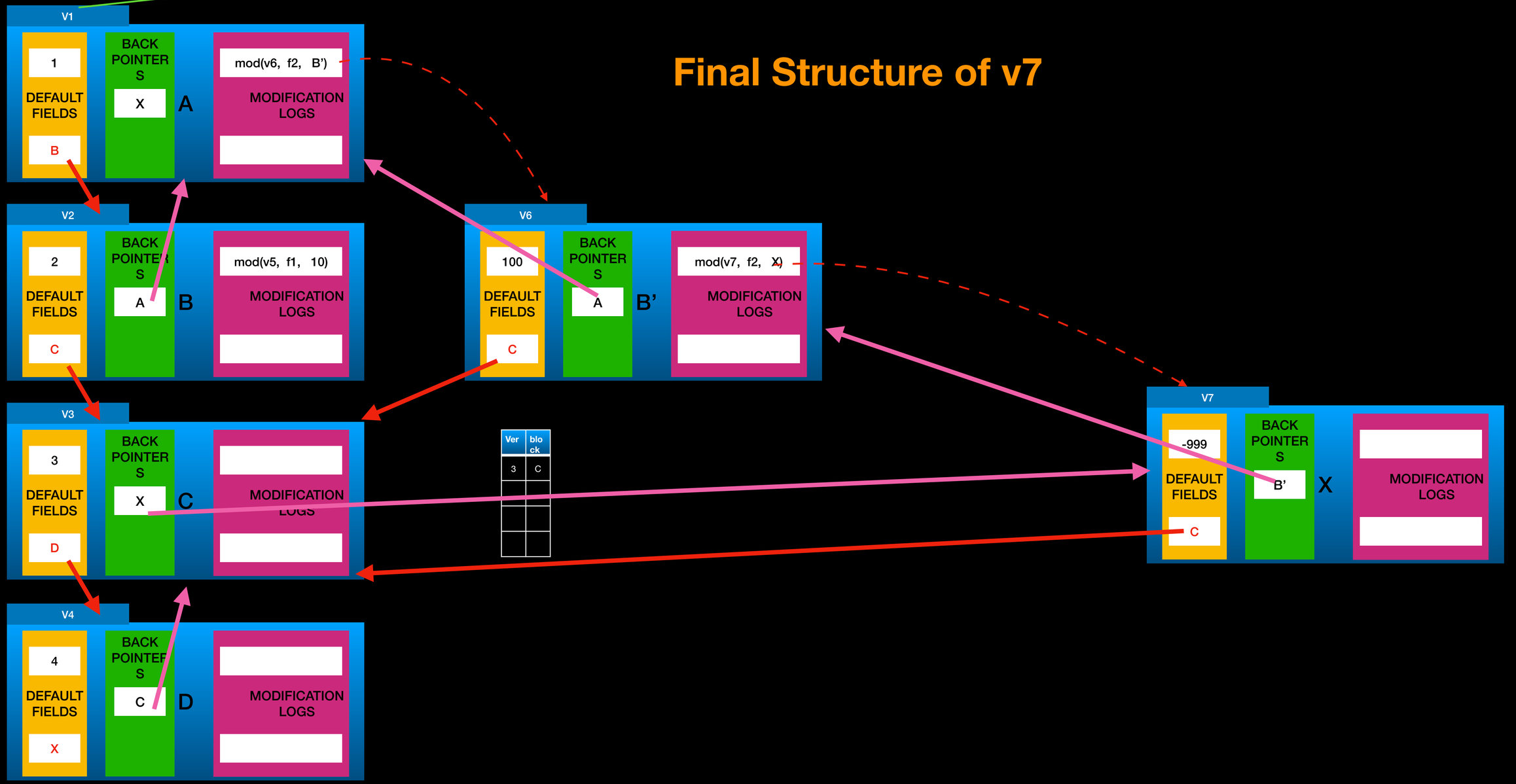
Current Version: v7

START MODULE

V0

add(X,B,_)

Final Structure of v7



VERSION REDIRECTOR MODULE

Ver	block
1	A

VERSION REDIRECTOR MODULE

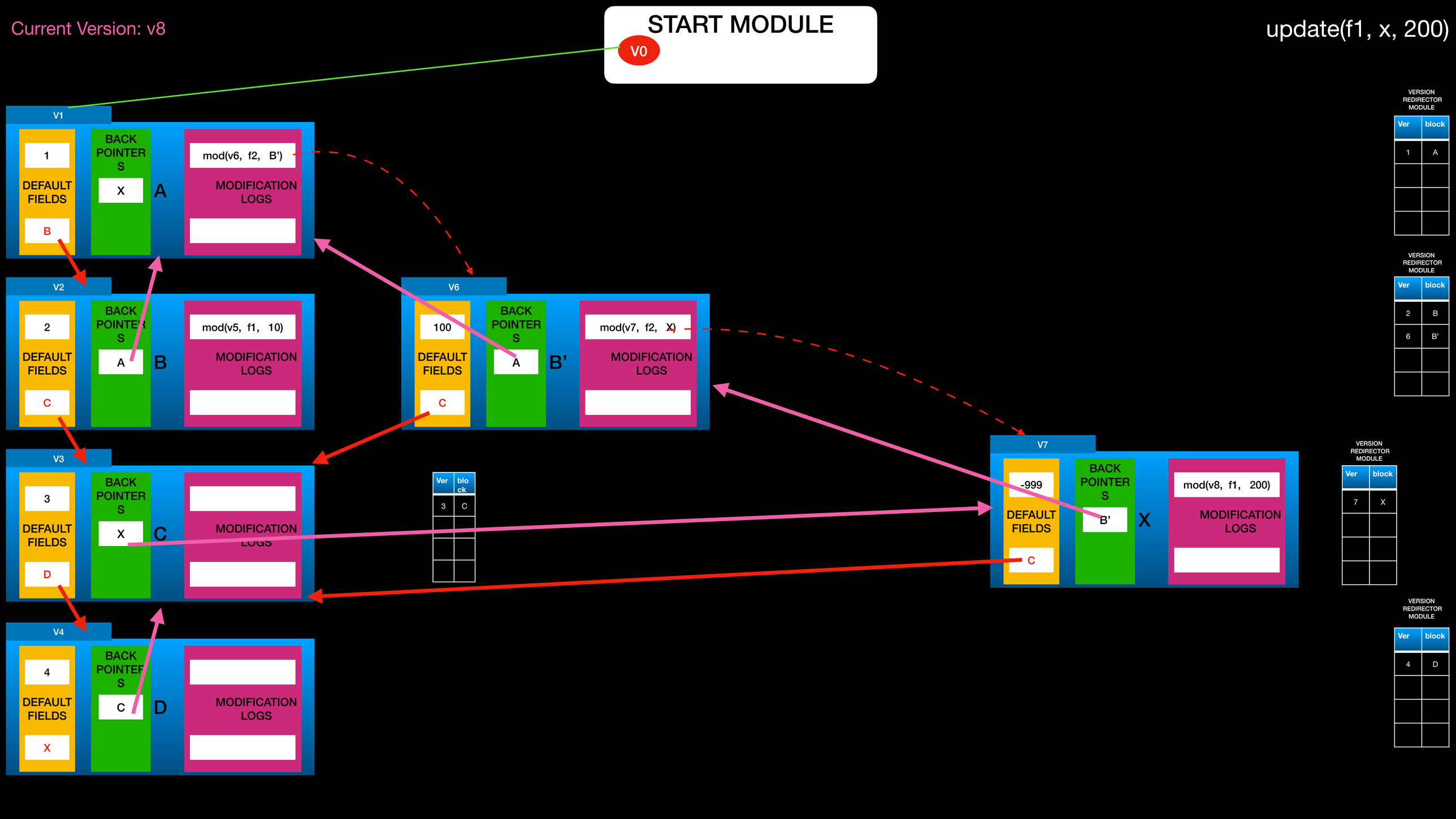
Ver	block
2	B
6	B'

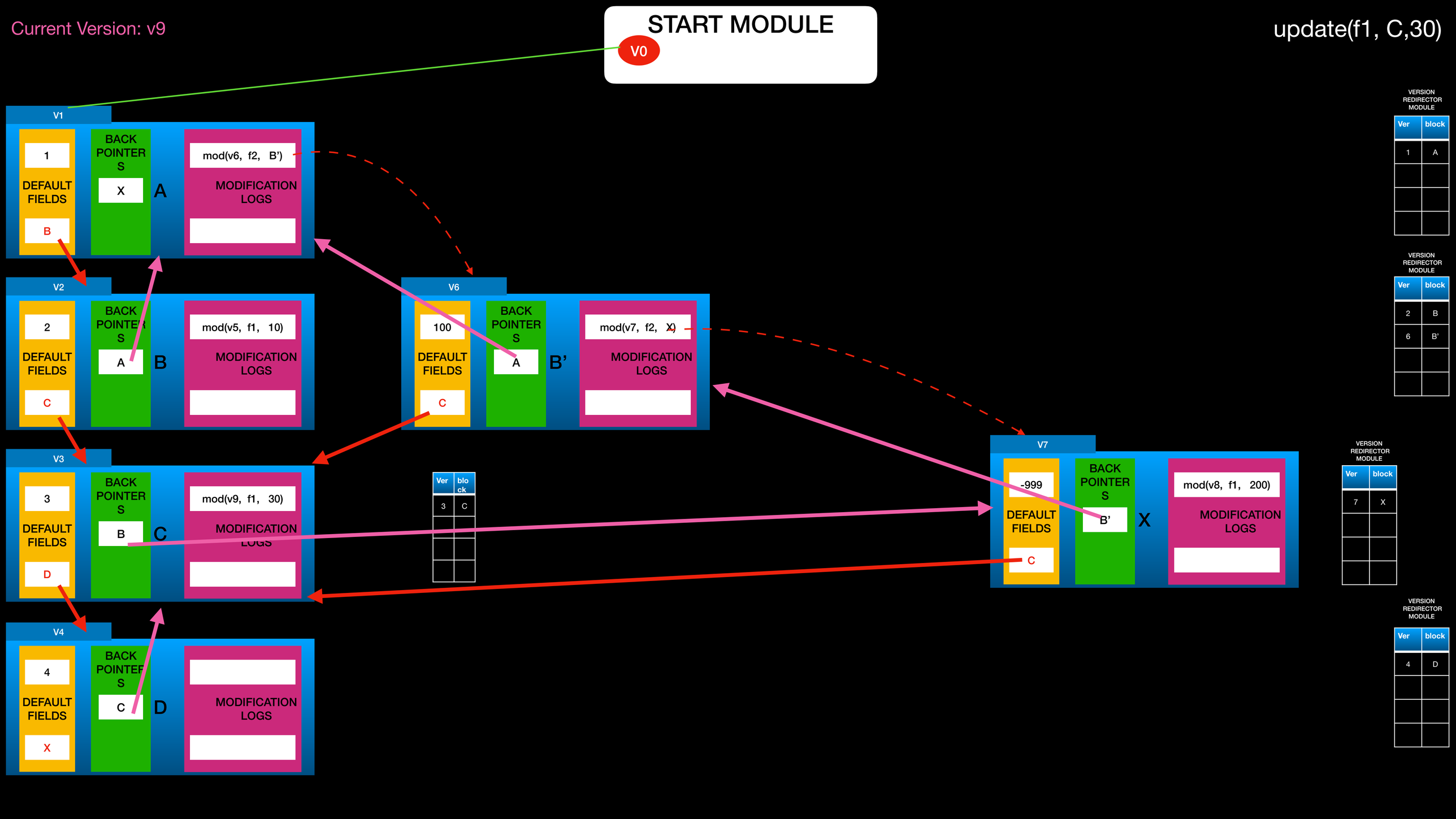
VERSION REDIRECTOR MODULE

Ver	block
7	X

VERSION REDIRECTOR MODULE

Ver	block
4	D


Current Version: v8
START MODULE
V0
update(f1, x, 200)
V1
1
DEFAULT FIELDS
B
BACK POINTERS
X
A
mod(v6, f2, B')
MODIFICATION LOGS
V2
2
DEFAULT FIELDS
C
BACK POINTERS
A
B
mod(v5, f1, 10)
MODIFICATION LOGS
V6
100
DEFAULT FIELDS
C
BACK POINTERS
A
B'
mod(v7, f2, X)
MODIFICATION LOGS
V3
3
DEFAULT FIELDS
D
BACK POINTERS
X
C
MODIFICATION LOGS
V4
4
DEFAULT FIELDS
X
BACK POINTERS
C
D
MODIFICATION LOGS
V7
-999
DEFAULT FIELDS
C
BACK POINTERS
B'
X
mod(v8, f1, 200)
MODIFICATION LOGS
Ver
block
3
C
VERSION REDIRECTOR MODULE
Ver
block
1
A
VERSION REDIRECTOR MODULE
Ver
block
2
B
6
B'
VERSION REDIRECTOR MODULE
Ver
block
7
X
VERSION REDIRECTOR MODULE
Ver
block
4
D


Current Version: v9
START MODULE
V0
update(f1, C,30)
V1
1
DEFAULT FIELDS
B
BACK POINTERS
X
A
mod(v6, f2, B')
MODIFICATION LOGS
V2
2
DEFAULT FIELDS
C
BACK POINTERS
A
B
mod(v5, f1, 10)
MODIFICATION LOGS
V6
100
DEFAULT FIELDS
C
BACK POINTERS
A
B'
mod(v7, f2, X)
MODIFICATION LOGS
V3
3
DEFAULT FIELDS
D
BACK POINTERS
B
C
mod(v9, f1, 30)
MODIFICATION LOGS
V4
4
DEFAULT FIELDS
X
BACK POINTERS
C
D
MODIFICATION LOGS
V7
-999
DEFAULT FIELDS
C
BACK POINTERS
B'
X
mod(v8, f1, 200)
MODIFICATION LOGS
Ver
block
3
C
VERSION REDIRECTOR MODULE
Ver
block
1
A
VERSION REDIRECTOR MODULE
Ver
block
2
B
6
B'
VERSION REDIRECTOR MODULE
Ver
block
7
X
VERSION REDIRECTOR MODULE
Ver
block
4
D

Current Version: v10

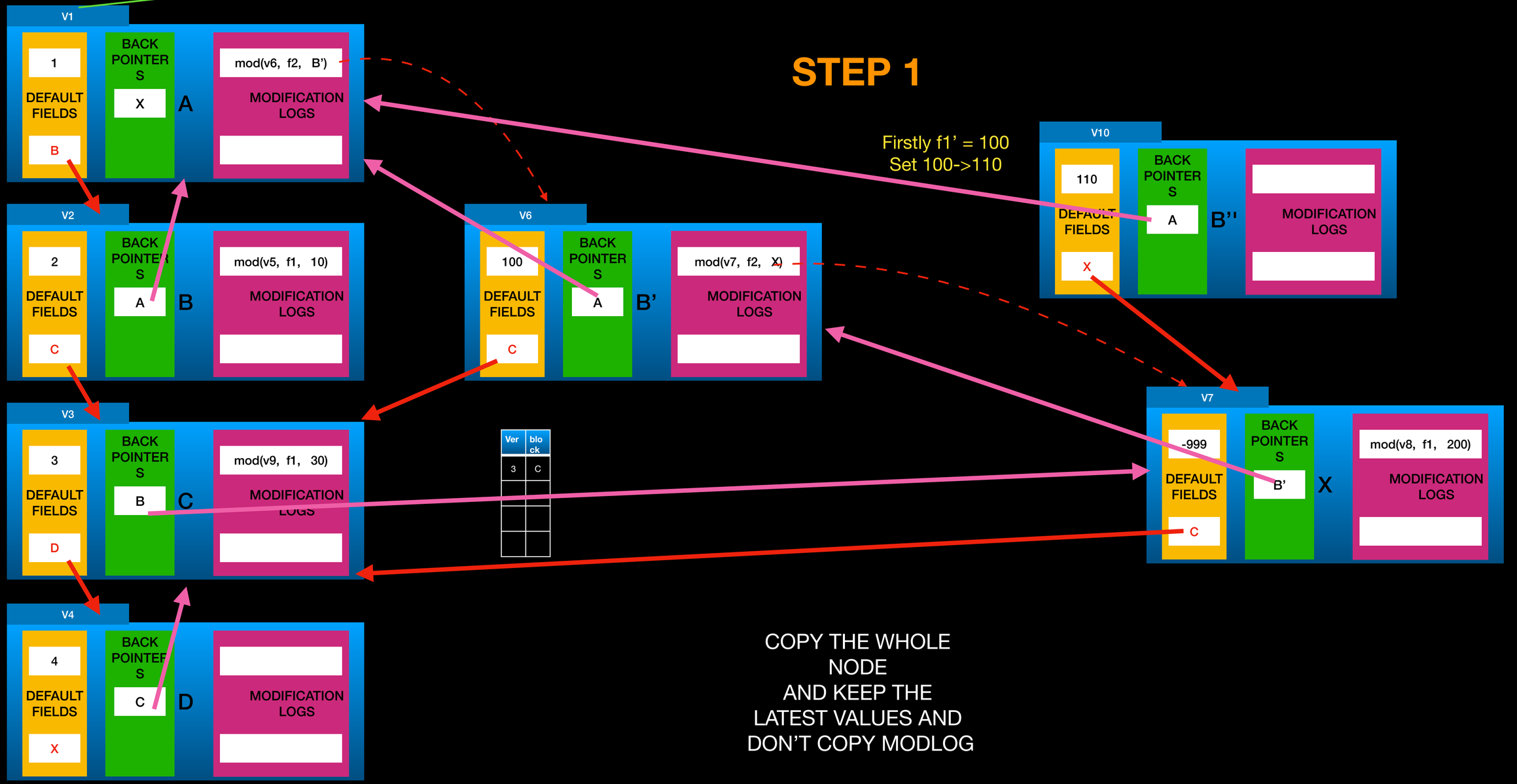
START MODULE

V0

update(f1, B,110)

STEP 1

Firstly f1' = 100
Set 100->110



Ver	block
3	C

VERSION REDIRECTOR MODULE

Ver	block
1	A

VERSION REDIRECTOR MODULE

Ver	block
2	B
6	B'
10	B''

VERSION REDIRECTOR MODULE

Ver	block
7	X

VERSION REDIRECTOR MODULE

Ver	block
4	D

COPY THE WHOLE NODE
AND KEEP THE LATEST VALUES AND
DON'T COPY MODLOG

Current Version: v10

START MODULE

V0

update(f1, B,110)

STEP 2

V1

1

DEFAULT FIELDS

B

BACK POINTERS

X

A

mod(v6, f2, B')

MODIFICATION LOGS

V2

2

DEFAULT FIELDS

C

BACK POINTERS

A

B

mod(v5, f1, 10)

MODIFICATION LOGS

V3

3

DEFAULT FIELDS

D

BACK POINTERS

B

C

mod(v9, f1, 30)

MODIFICATION LOGS

V4

4

DEFAULT FIELDS

X

BACK POINTERS

C

D

MODIFICATION LOGS

V6

100

DEFAULT FIELDS

C

BACK POINTERS

A

B'

mod(v7, f2, X)

MODIFICATION LOGS

V10

110

DEFAULT FIELDS

X

BACK POINTERS

A

B''

MODIFICATION LOGS

V7

-999

DEFAULT FIELDS

C

BACK POINTERS

B"

X

mod(v8, f1, 200)

MODIFICATION LOGS

Ver	block
3	C

VERSION REDIRECTOR MODULE

Ver	block
1	A

VERSION REDIRECTOR MODULE

Ver	block
2	B
6	B'
10	B"

VERSION REDIRECTOR MODULE

Ver	block
7	X

VERSION REDIRECTOR MODULE

Ver	block
4	D

Rewrite THE BACK POINTERS
OF THE SUCCESSOR NODE
TO THE NEWLY CREATED NODE

GO TO SUCCESSOR NODES USING
THE FORWARD POINTERS
IN O(1)

Current Version: v10

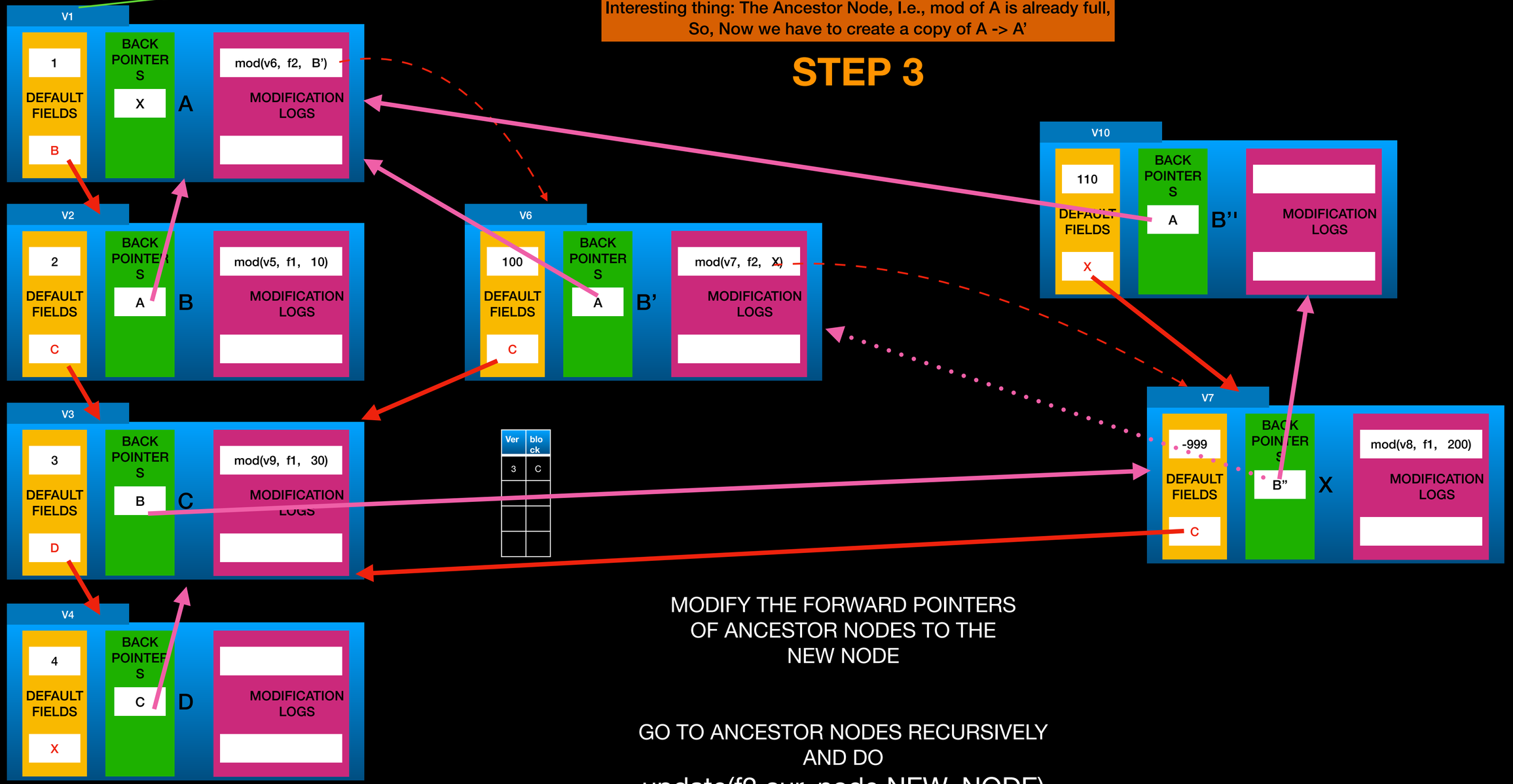
update(f1, B,110)

START MODULE

V0

Interesting thing: The Ancestor Node, I.e., mod of A is already full,
So, Now we have to create a copy of A -> A'

STEP 3



Ver	block
3	C

VERSION REDIRECTOR MODULE

Ver	block
1	A

VERSION REDIRECTOR MODULE

Ver	block
2	B
6	B'
10	B"

VERSION REDIRECTOR MODULE

Ver	block
7	X

VERSION REDIRECTOR MODULE

Ver	block
4	D

MODIFY THE FORWARD POINTERS
OF ANCESTOR NODES TO THE
NEW NODE

GO TO ANCESTOR NODES RECURSIVELY
AND DO

update(f2,cur_node,NEW_NODE)

Current Version: v10

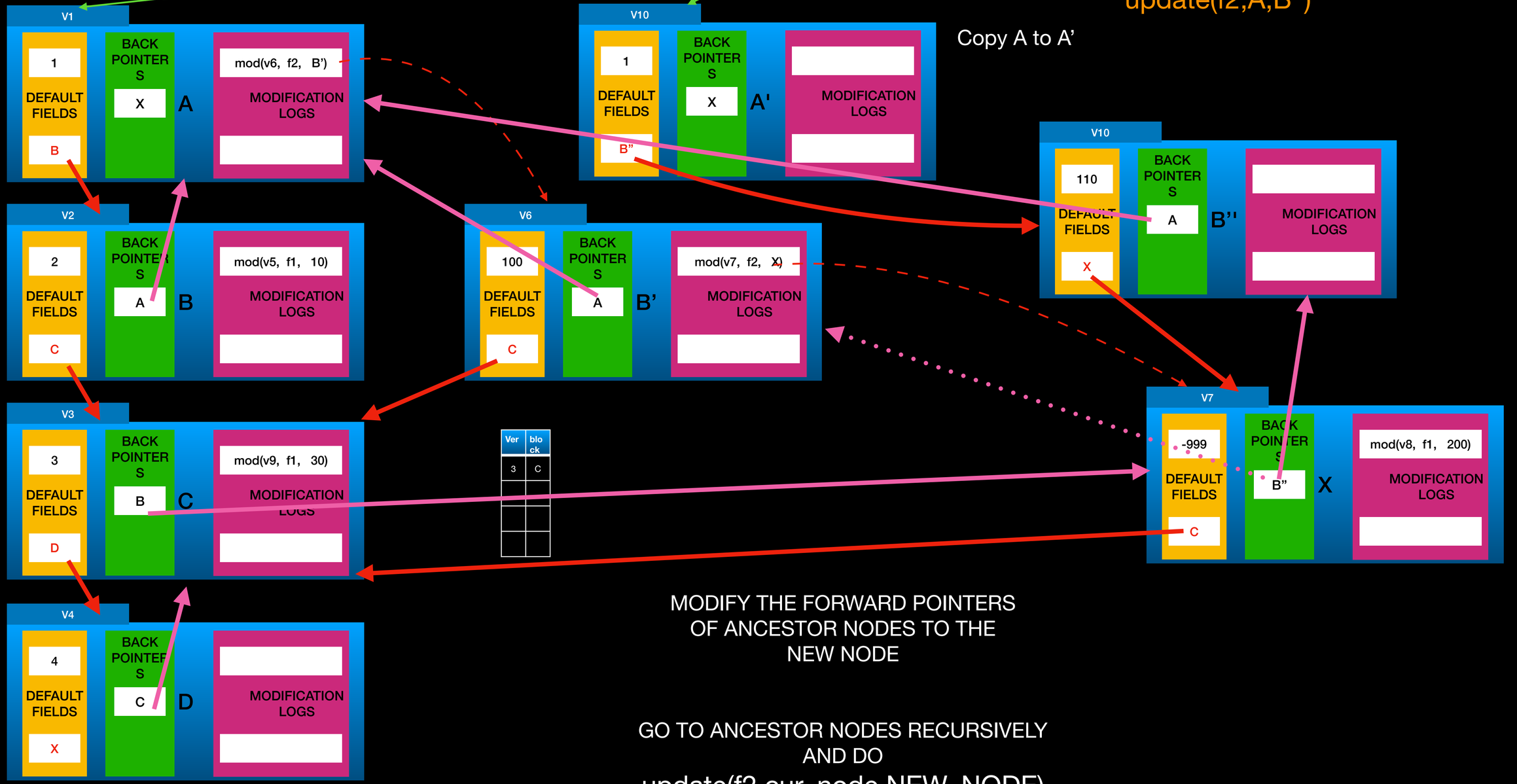
START MODULE

V0 V10

# STEP 3
# SUBSTEP 1
update(f2,A,B")

update(f1, B,110)

Copy A to A'



Ver	block
3	C

VERSION REDIRECTOR MODULE

Ver	block
1	A
10	A"

VERSION REDIRECTOR MODULE

Ver	block
2	B
6	B'
10	B"

VERSION REDIRECTOR MODULE

Ver	block
7	X

VERSION REDIRECTOR MODULE

Ver	block
4	D

MODIFY THE FORWARD POINTERS OF ANCESTOR NODES TO THE NEW NODE

GO TO ANCESTOR NODES RECURSIVELY AND DO

update(f2,cur_node,NEW_NODE)

Current Version: v10

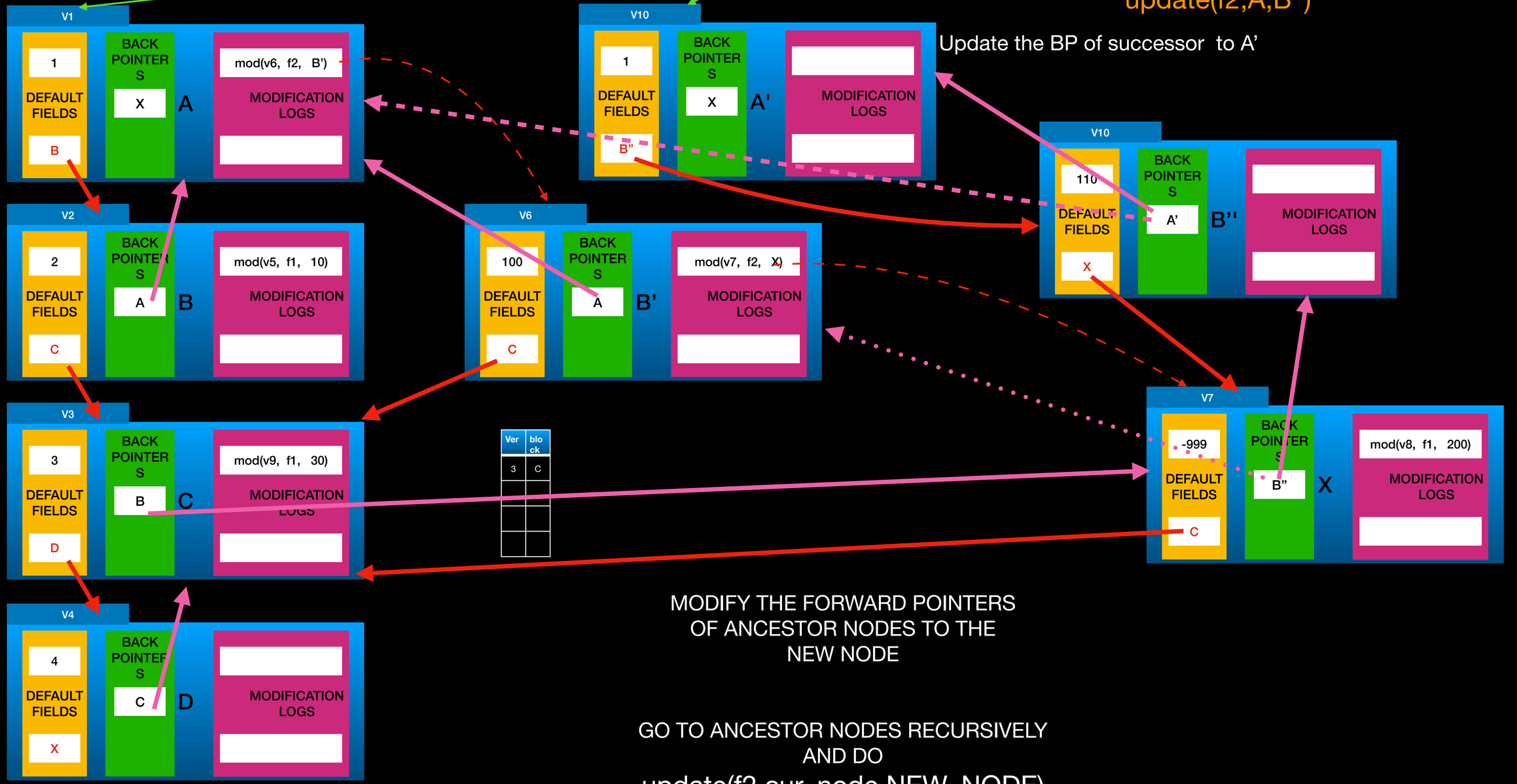
START MODULE

V0 V10

# STEP 3
# SUBSTEP 2
update(f2,A,B")

update(f1, B,110)

Update the BP of successor to A'



Ver	block
3	C

VERSION REDIRECTOR MODULE

Ver	block
1	A
10	A"

VERSION REDIRECTOR MODULE

Ver	block
2	B
6	B'
10	B"

VERSION REDIRECTOR MODULE

Ver	block
7	X

VERSION REDIRECTOR MODULE

Ver	block
4	D

MODIFY THE FORWARD POINTERS OF ANCESTOR NODES TO THE NEW NODE

GO TO ANCESTOR NODES RECURSIVELY AND DO

update(f2,cur_node,NEW_NODE)

Current Version: v10

START MODULE

V0 V10

STEP 3

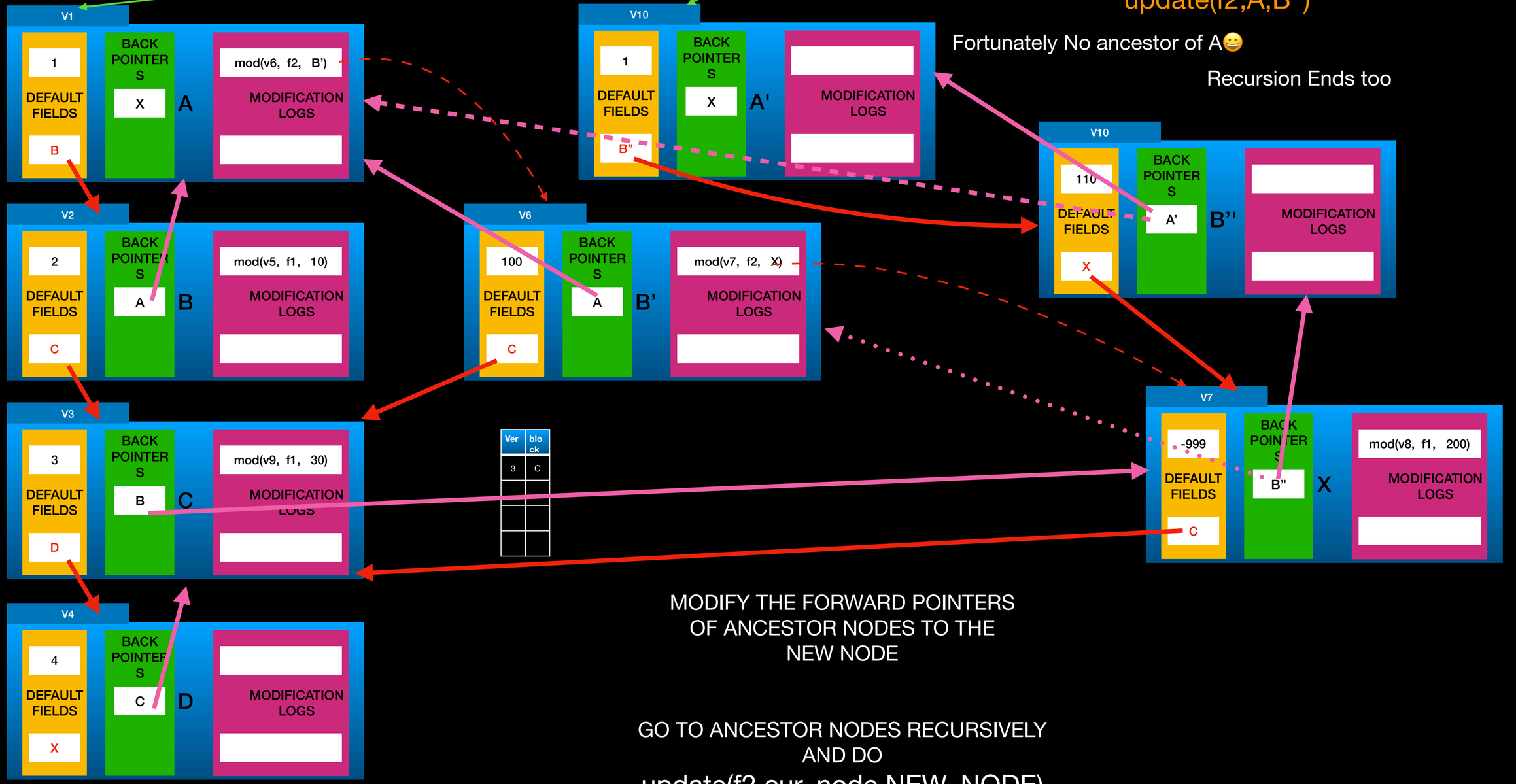
SUBSTEP 3

update(f2,A,B")

update(f1, B,110)

Fortunately No ancestor of A😀

Recursion Ends too



Ver	block
3	C

VERSION REDIRECTOR MODULE

Ver	block
1	A
10	A"

VERSION REDIRECTOR MODULE

Ver	block
2	B
6	B'
10	B"

VERSION REDIRECTOR MODULE

Ver	block
7	X

VERSION REDIRECTOR MODULE

Ver	block
4	D

MODIFY THE FORWARD POINTERS OF ANCESTOR NODES TO THE NEW NODE

GO TO ANCESTOR NODES RECURSIVELY AND DO

update(f2,cur_node,NEW_NODE)

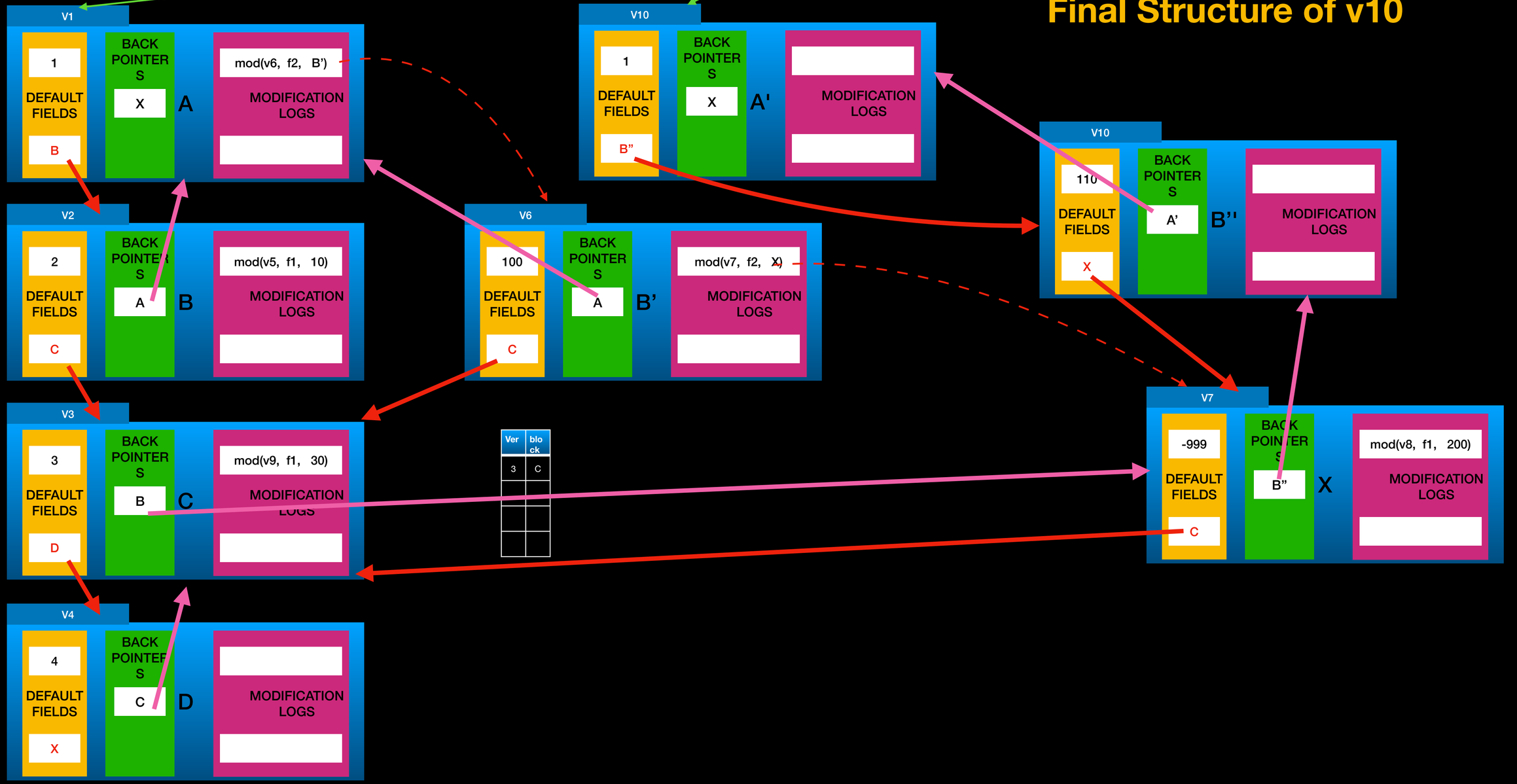
Current Version: v10

update(f1, B,110)

START MODULE

V0 V10

Final Structure of v10



Ver	block
3	C

VERSION REDIRECTOR MODULE

Ver	block
1	A
10	A"

VERSION REDIRECTOR MODULE

Ver	block
2	B
6	B'
10	B"

VERSION REDIRECTOR MODULE

Ver	block
7	X

VERSION REDIRECTOR MODULE

Ver	block
4	D

Current Version: v11

START MODULE

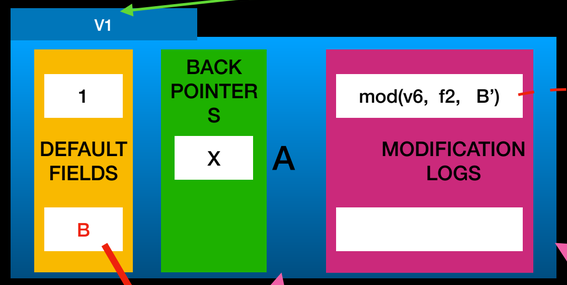
V0 V10

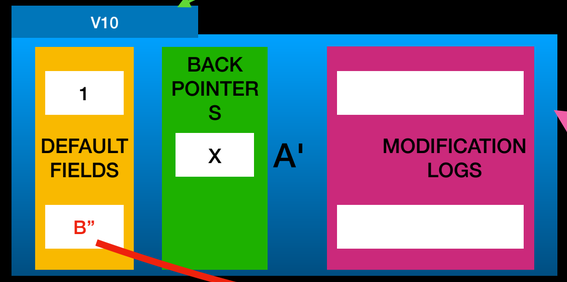
remove(C)

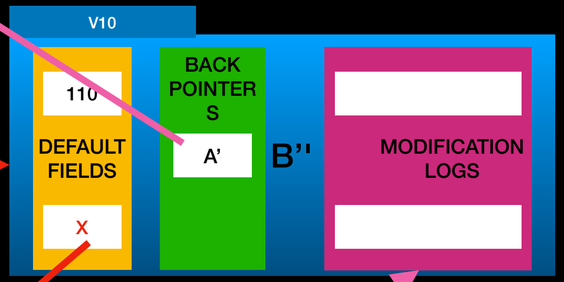
Remove C consists of

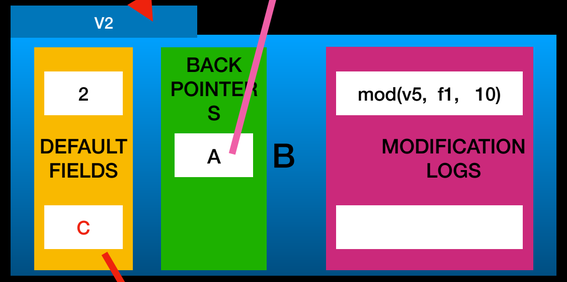
Modify F2 of Parent of C (i.e., X) -> F2 of C (successor of C)

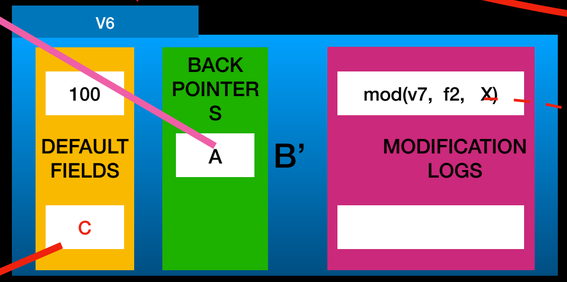
Then, set the BP1 of Successor of C, I.e. D, to X’

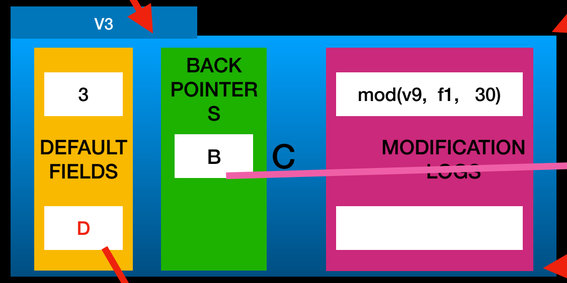




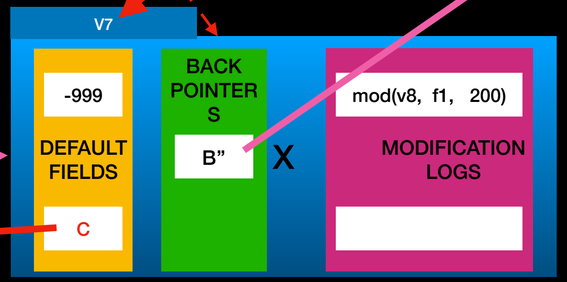


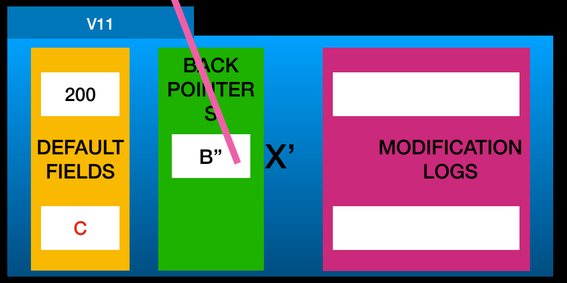


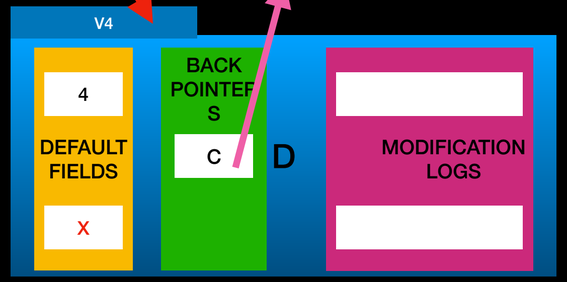




Ver	block
3	C







COPY THE WHOLE NODE AND KEEP THE LATEST VALUES AND DON’T COPY MODLOG

VERSION REDIRECTOR MODULE

Ver	block
1	A
10	A”

VERSION REDIRECTOR MODULE

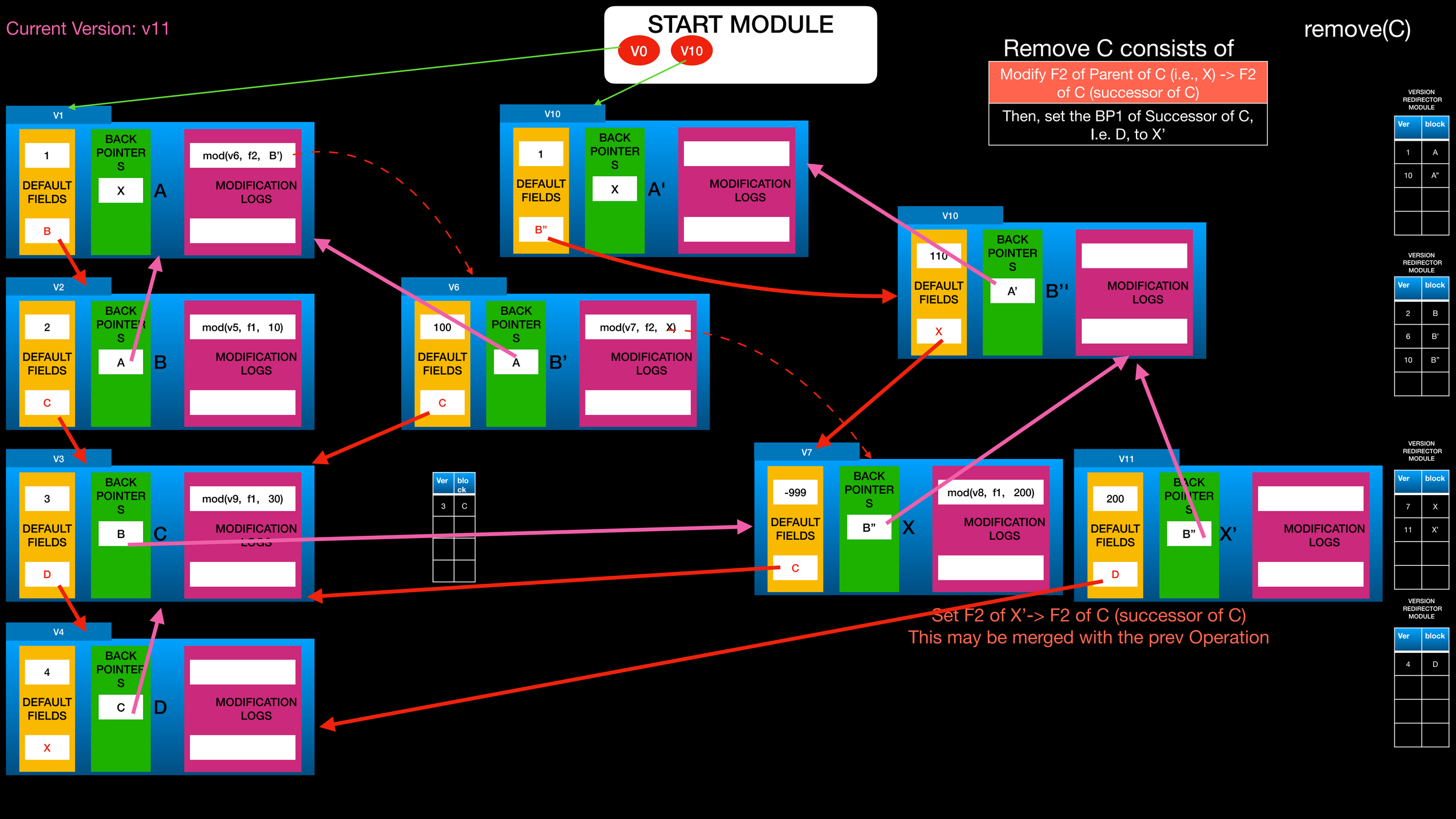
Ver	block
2	B
6	B’
10	B”

VERSION REDIRECTOR MODULE

Ver	block
7	X
11	X’

VERSION REDIRECTOR MODULE

Ver	block
4	D


Current Version: v11
START MODULE
V0
V10
remove(C)
Remove C consists of
Modify F2 of Parent of C (i.e., X) -> F2 of C (successor of C)
Then, set the BP1 of Successor of C, I.e. D, to X'
V1
1
DEFAULT FIELDS
B
BACK POINTERS
X
A
mod(v6, f2, B')
MODIFICATION LOGS
V10
1
DEFAULT FIELDS
B"
BACK POINTERS
X
A'
MODIFICATION LOGS
V10
110
DEFAULT FIELDS
X
BACK POINTERS
A'
B''
MODIFICATION LOGS
V2
2
DEFAULT FIELDS
C
BACK POINTERS
A
B
mod(v5, f1, 10)
MODIFICATION LOGS
V6
100
DEFAULT FIELDS
C
BACK POINTERS
A
B'
mod(v7, f2, X)
MODIFICATION LOGS
V3
3
DEFAULT FIELDS
D
BACK POINTERS
B
C
mod(v9, f1, 30)
MODIFICATION LOGS
Ver
block
3
C
V7
-999
DEFAULT FIELDS
C
BACK POINTERS
B"
X
mod(v8, f1, 200)
MODIFICATION LOGS
V11
200
DEFAULT FIELDS
D
BACK POINTERS
B"
X'
MODIFICATION LOGS
V4
4
DEFAULT FIELDS
X
BACK POINTERS
C
D
MODIFICATION LOGS
Set F2 of X'-> F2 of C (successor of C)
This may be merged with the prev Operation
VERSION REDIRECTOR MODULE
Ver
block
1
A
10
A"
VERSION REDIRECTOR MODULE
Ver
block
2
B
6
B'
10
B"
VERSION REDIRECTOR MODULE
Ver
block
7
X
11
X'
VERSION REDIRECTOR MODULE
Ver
block
4
D

Current Version: v11

START MODULE

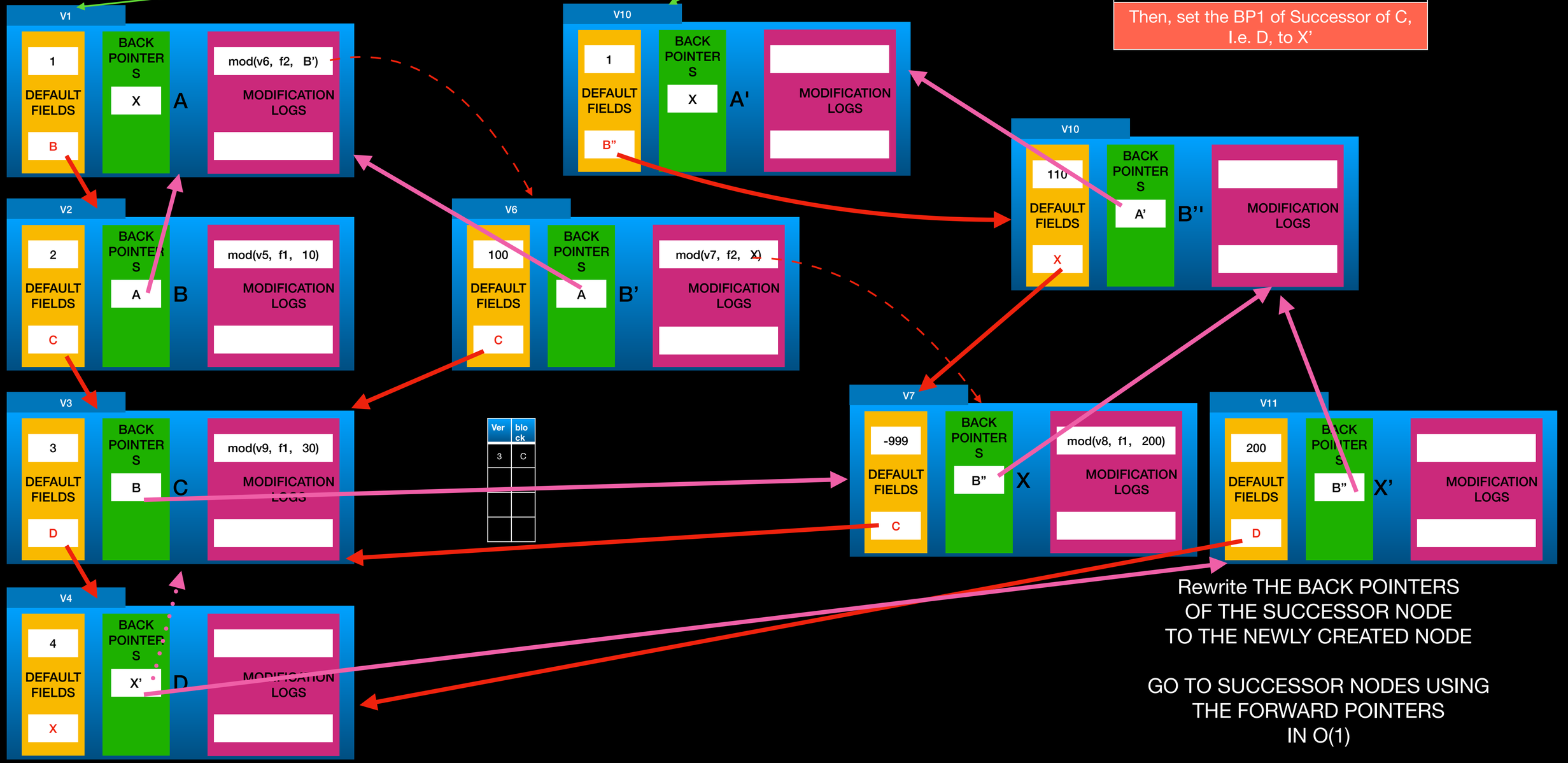
V0 V10

remove(C)

Remove C consists of

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

Then, set the BP1 of Successor of C, I.e. D, to X'



Ver	block
3	C

VERSION REDIRECTOR MODULE

Ver	block
1	A
10	A"

VERSION REDIRECTOR MODULE

Ver	block
2	B
6	B'
10	B"

VERSION REDIRECTOR MODULE

Ver	block
7	X
11	X'

VERSION REDIRECTOR MODULE

Ver	block
4	D

Rewrite THE BACK POINTERS OF THE SUCCESSOR NODE TO THE NEWLY CREATED NODE

GO TO SUCCESSOR NODES USING THE FORWARD POINTERS IN O(1)

Current Version: v11

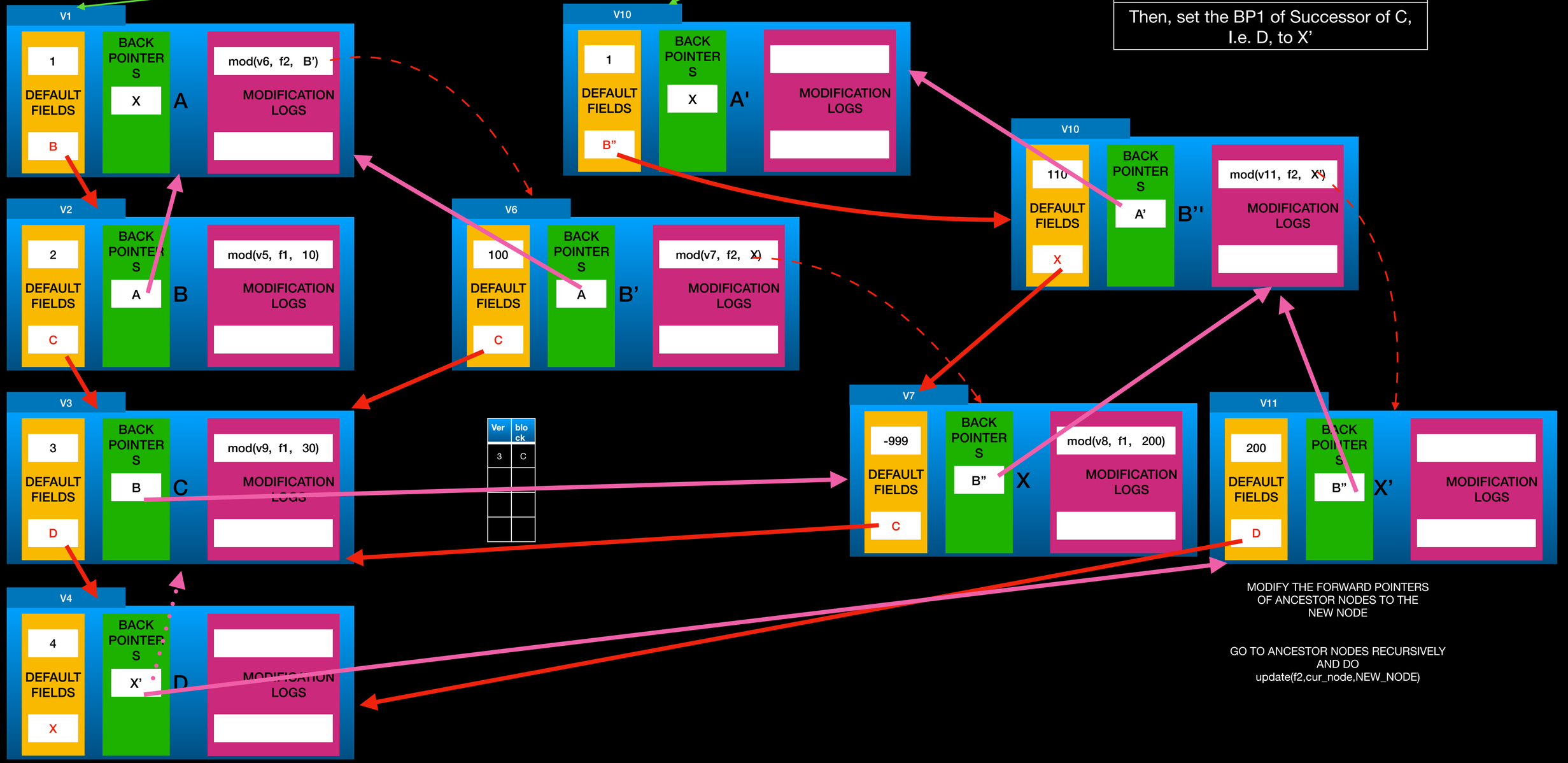
START MODULE

V0 V10

remove(C)

Remove C consists of

Modify F2 of Parent of C (i.e., X) -> F2 of C (successor of C)
Then, set the BP1 of Successor of C, I.e. D, to X’



Ver	block
3	C

MODIFY THE FORWARD POINTERS OF ANCESTOR NODES TO THE NEW NODE

GO TO ANCESTOR NODES RECURSIVELY AND DO update(f2,cur_node,NEW_NODE)

VERSION REDIRECTOR MODULE

Ver	block
1	A
10	A”

VERSION REDIRECTOR MODULE

Ver	block
2	B
6	B’
10	B”

VERSION REDIRECTOR MODULE

Ver	block
7	X
11	X’

VERSION REDIRECTOR MODULE

Ver	block
4	D

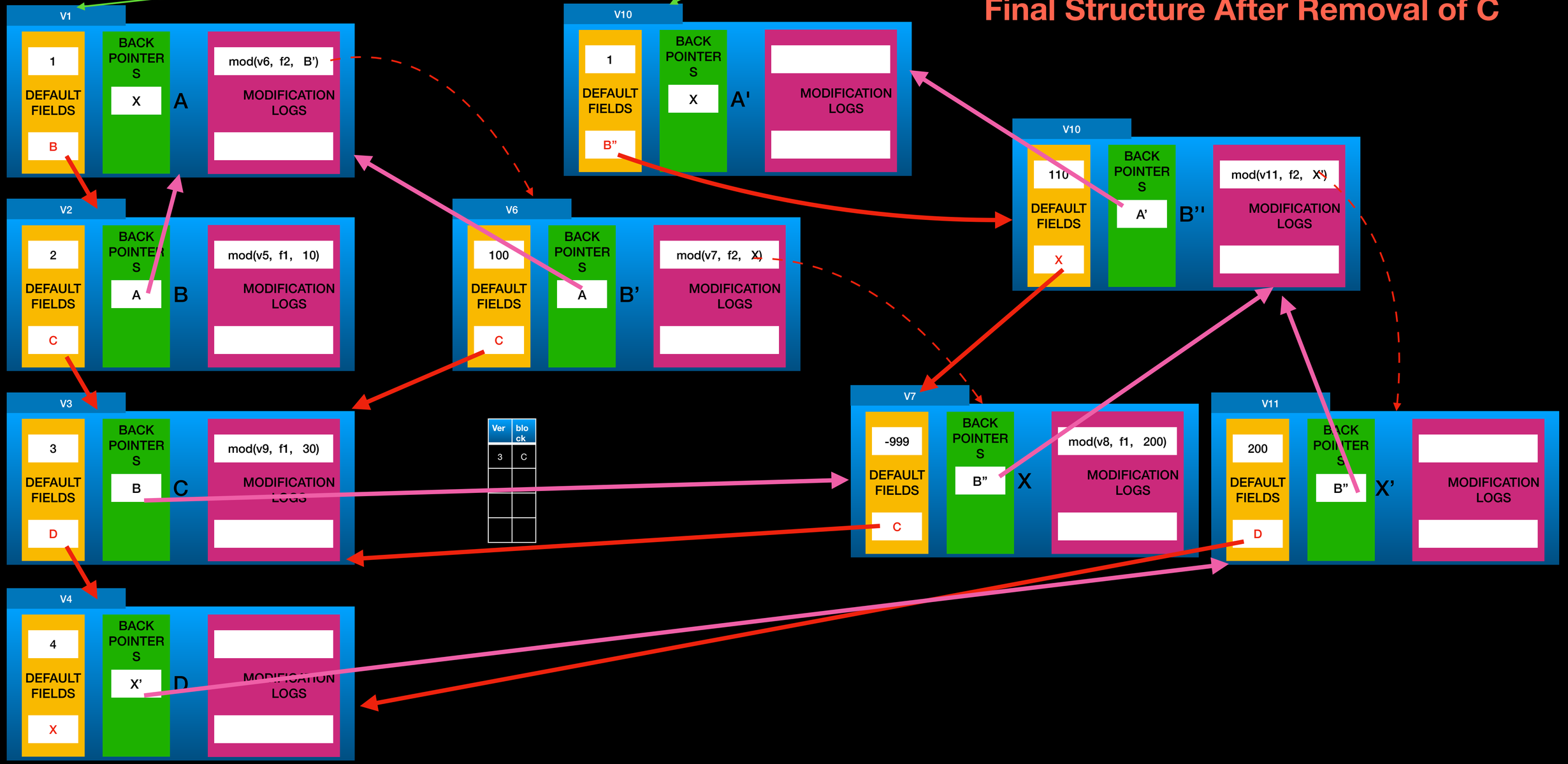
Current Version: v11

remove(C)

START MODULE

V0 V10

Final Structure After Removal of C



VERSION REDIRECTOR MODULE

Ver	block
1	A
10	A"

VERSION REDIRECTOR MODULE

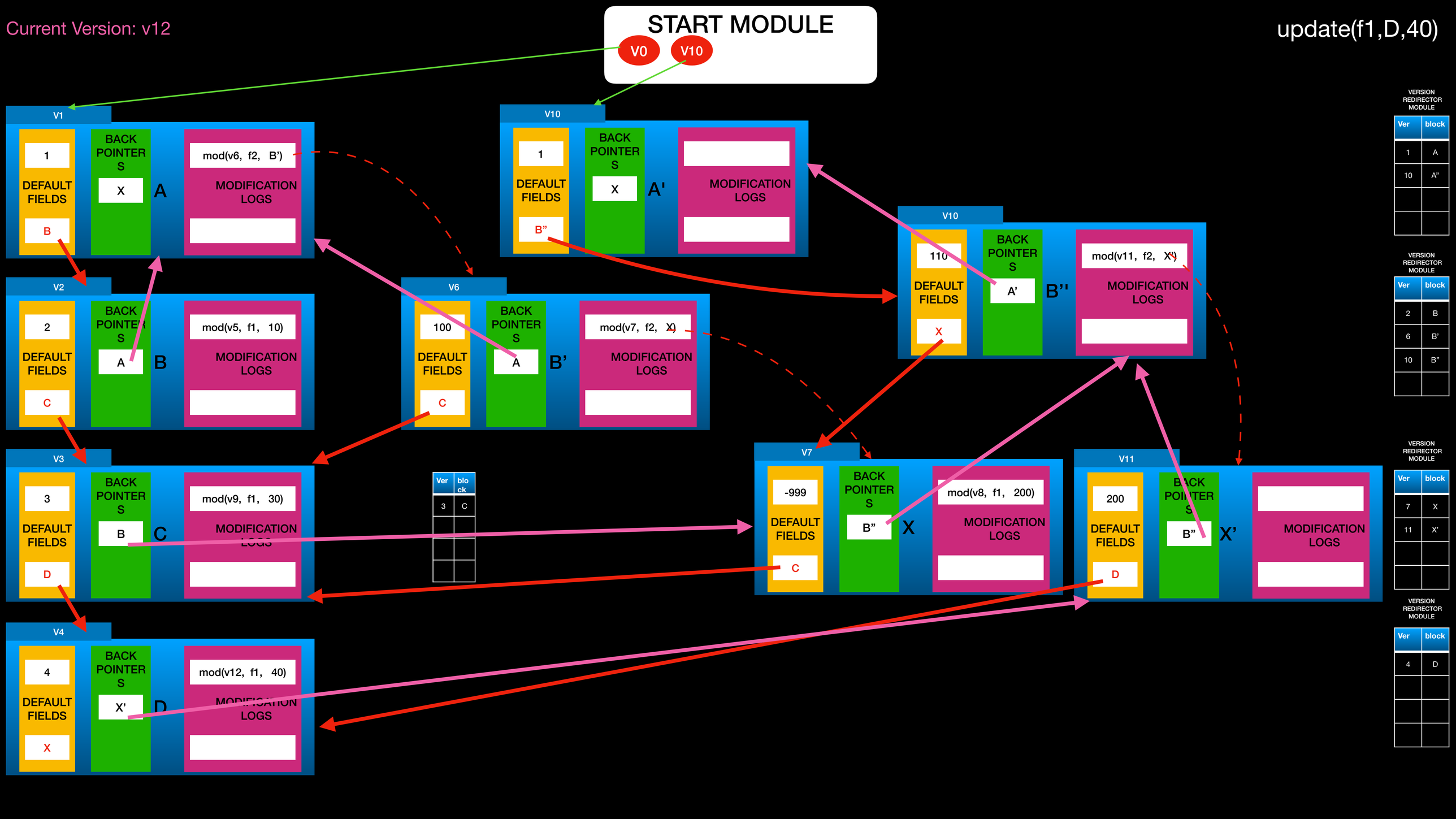
Ver	block
2	B
6	B'
10	B"

VERSION REDIRECTOR MODULE

Ver	block
7	X
11	X'

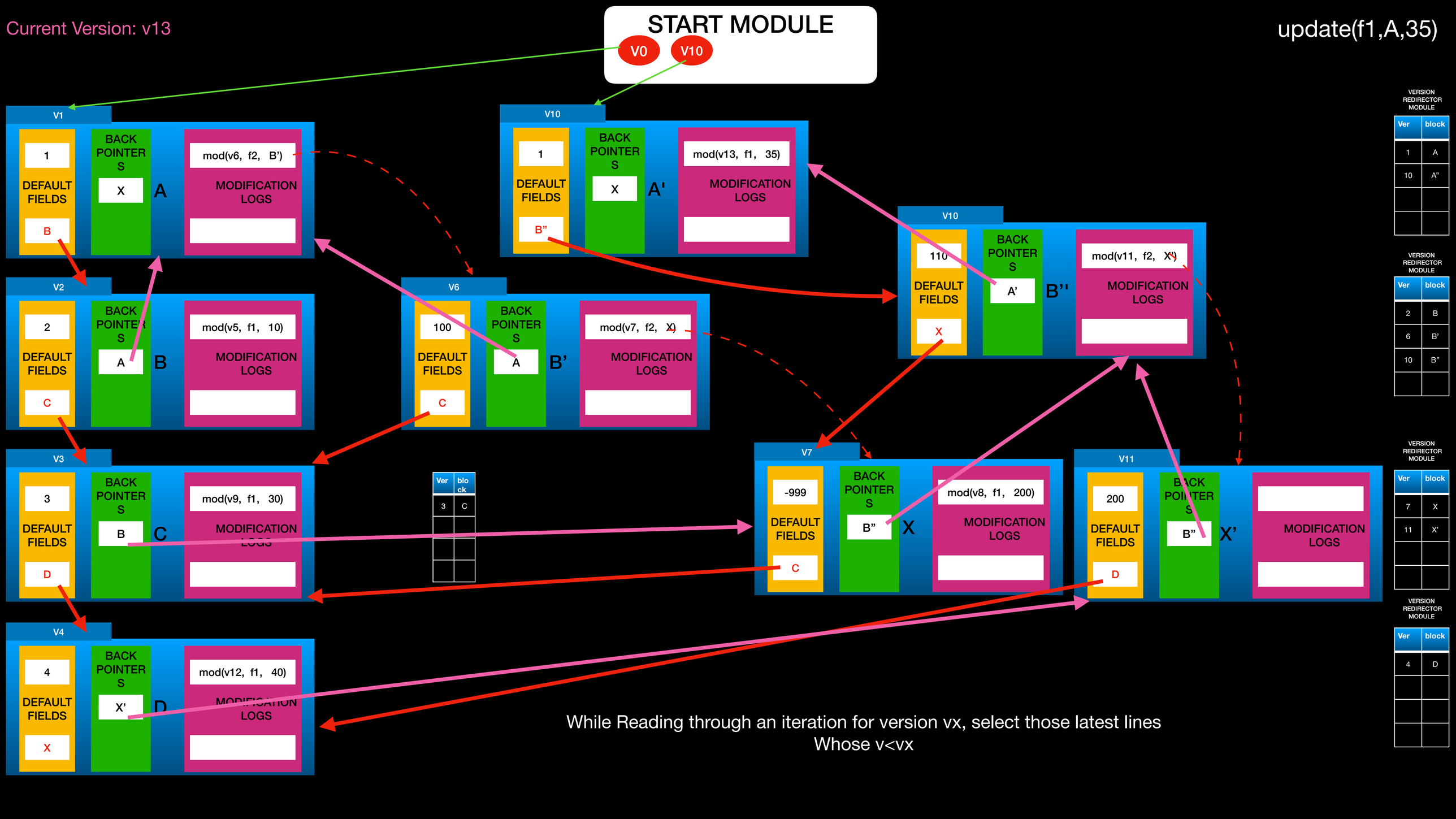
VERSION REDIRECTOR MODULE

Ver	block
4	D


Current Version: v12
START MODULE
V0
V10
update(f1,D,40)
V1
1
DEFAULT FIELDS
B
BACK POINTERS
X
A
mod(v6, f2, B')
MODIFICATION LOGS
V10
1
DEFAULT FIELDS
B"
BACK POINTERS
X
A'
MODIFICATION LOGS
V10
110
DEFAULT FIELDS
X
BACK POINTERS
A'
B''
mod(v11, f2, X')
MODIFICATION LOGS
V2
2
DEFAULT FIELDS
C
BACK POINTERS
A
B
mod(v5, f1, 10)
MODIFICATION LOGS
V6
100
DEFAULT FIELDS
C
BACK POINTERS
A
B'
mod(v7, f2, X)
MODIFICATION LOGS
V3
3
DEFAULT FIELDS
D
BACK POINTERS
B
C
mod(v9, f1, 30)
MODIFICATION LOGS
Ver
block
3
C
V7
-999
DEFAULT FIELDS
C
BACK POINTERS
B"
X
mod(v8, f1, 200)
MODIFICATION LOGS
V11
200
DEFAULT FIELDS
D
BACK POINTERS
B"
X'
MODIFICATION LOGS
V4
4
DEFAULT FIELDS
X
BACK POINTERS
X'
D
mod(v12, f1, 40)
MODIFICATION LOGS
VERSION REDIRECTOR MODULE
Ver
block
1
A
10
A"
VERSION REDIRECTOR MODULE
Ver
block
2
B
6
B'
10
B"
VERSION REDIRECTOR MODULE
Ver
block
7
X
11
X'
VERSION REDIRECTOR MODULE
Ver
block
4
D

Current Version: v13

update(f1,A,35)



While Reading through an iteration for version vx, select those latest lines
Whose v<vx

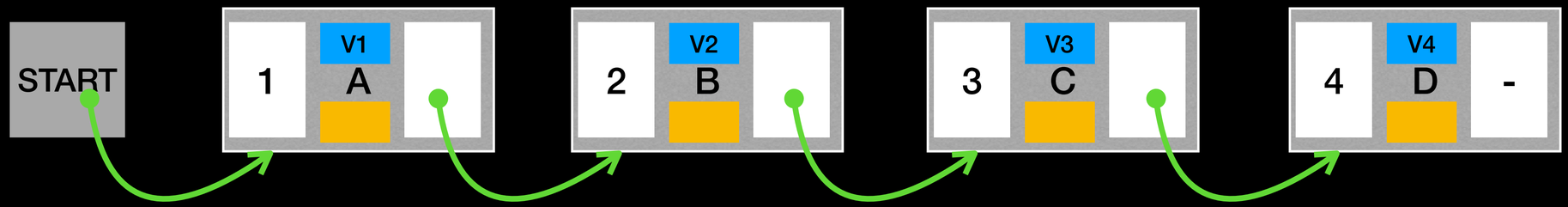
iterate_LL_at_v(vx)

To print the list at v_x

- Start from start module
- 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).
- 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.

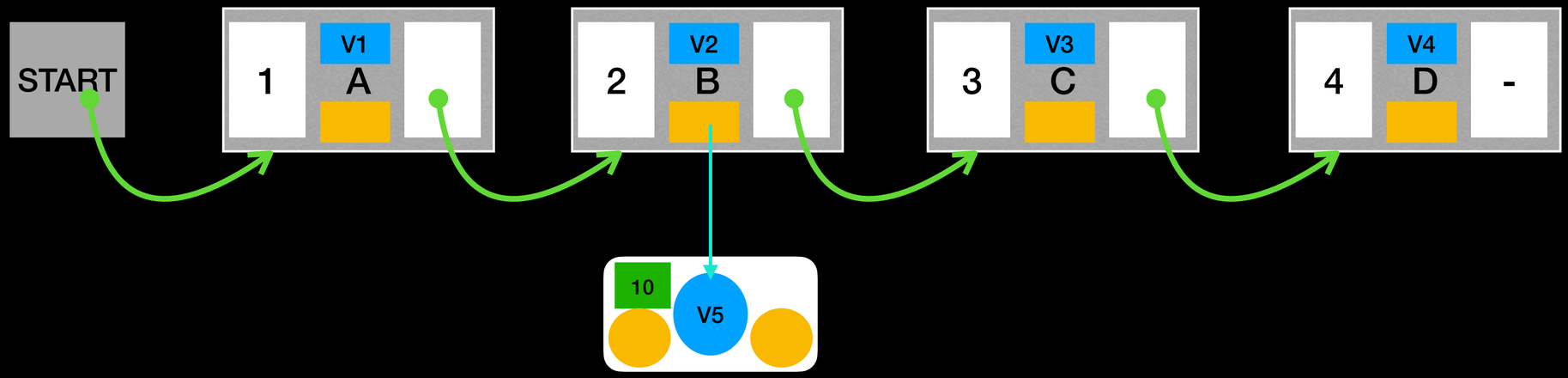
Why such approach is better ? 🥲

Normal Implementation (using balanced BST [may use just LL])



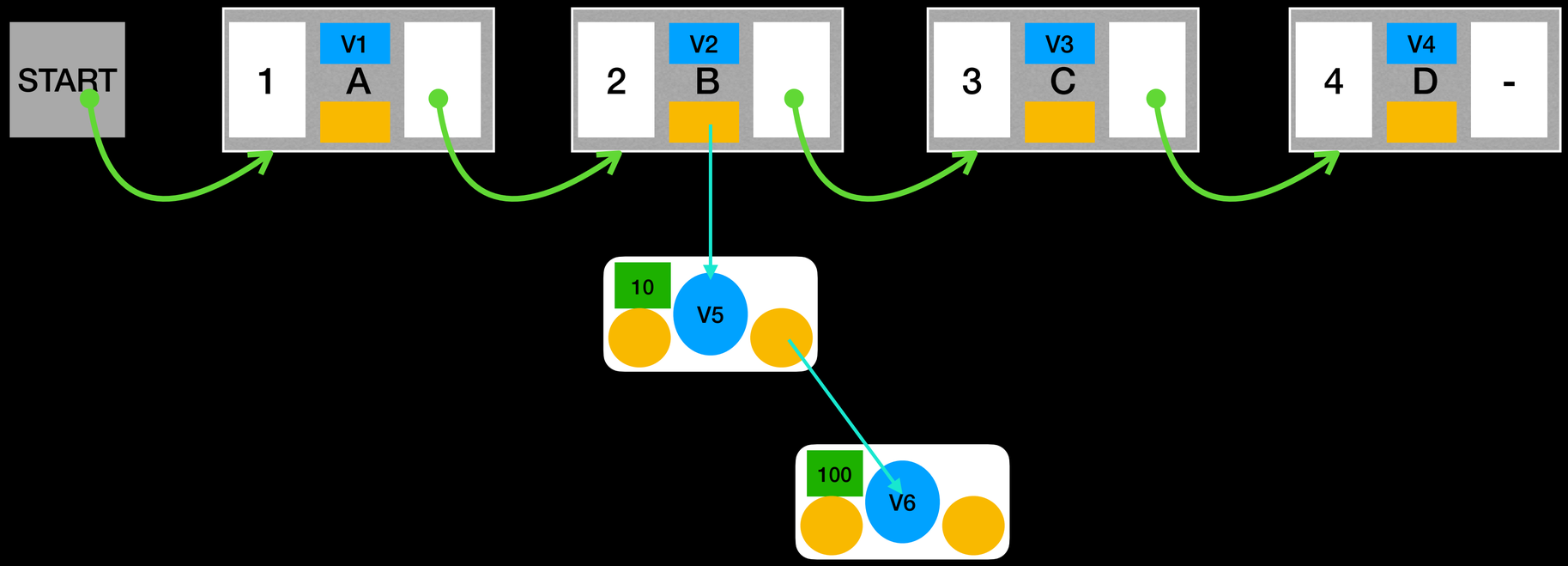
Why such approach is better ? 🥲

Normal Implementation (using balanced BST)



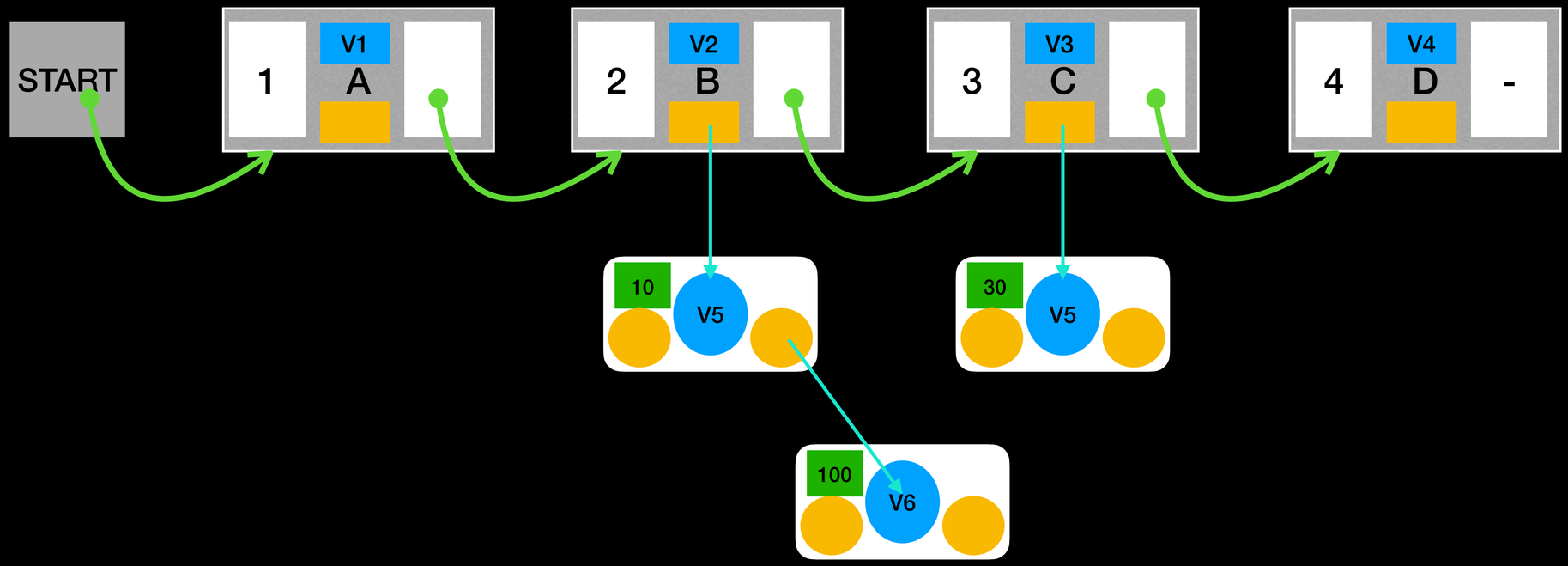
Why such approach is better ? 🥲

Normal Implementation (using balanced BST)



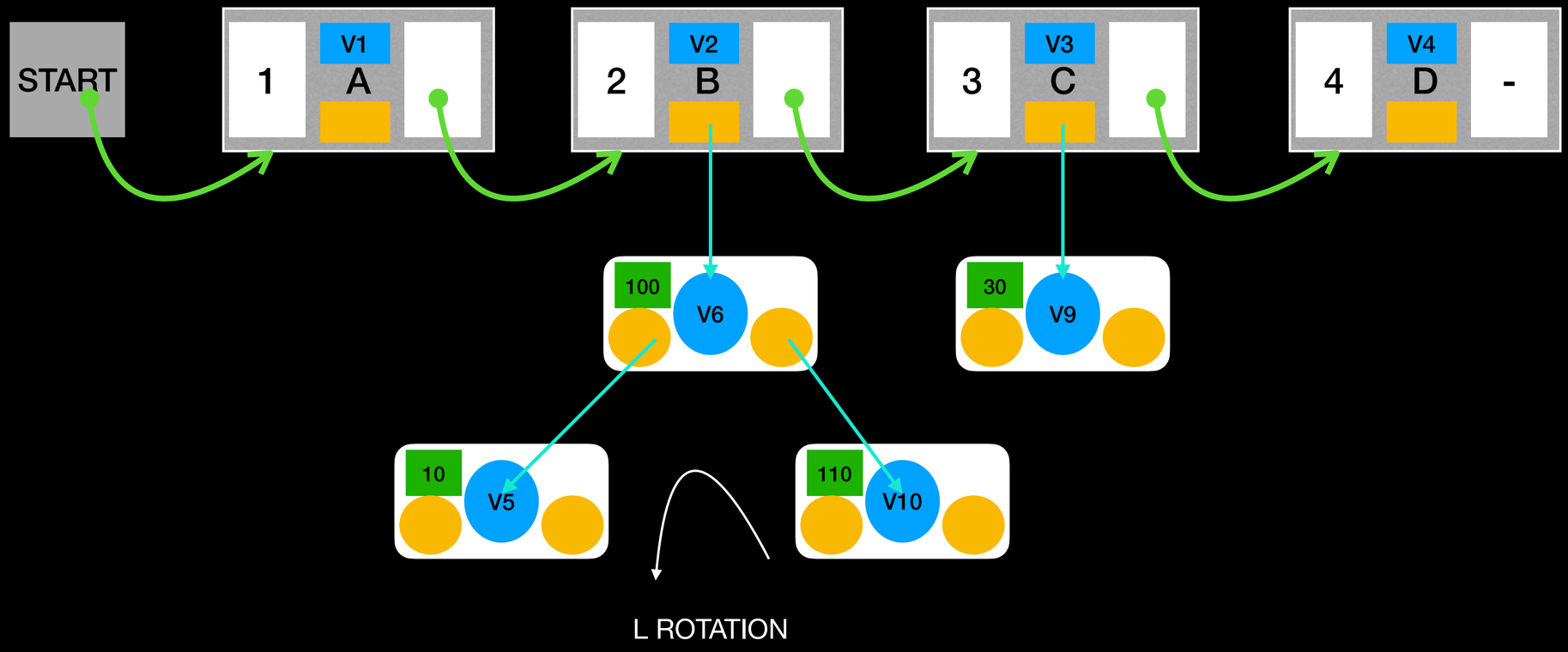
Why such approach is better ? 🥲

Normal Implementation (using balanced BST)



Why such approach is better ? 🥲

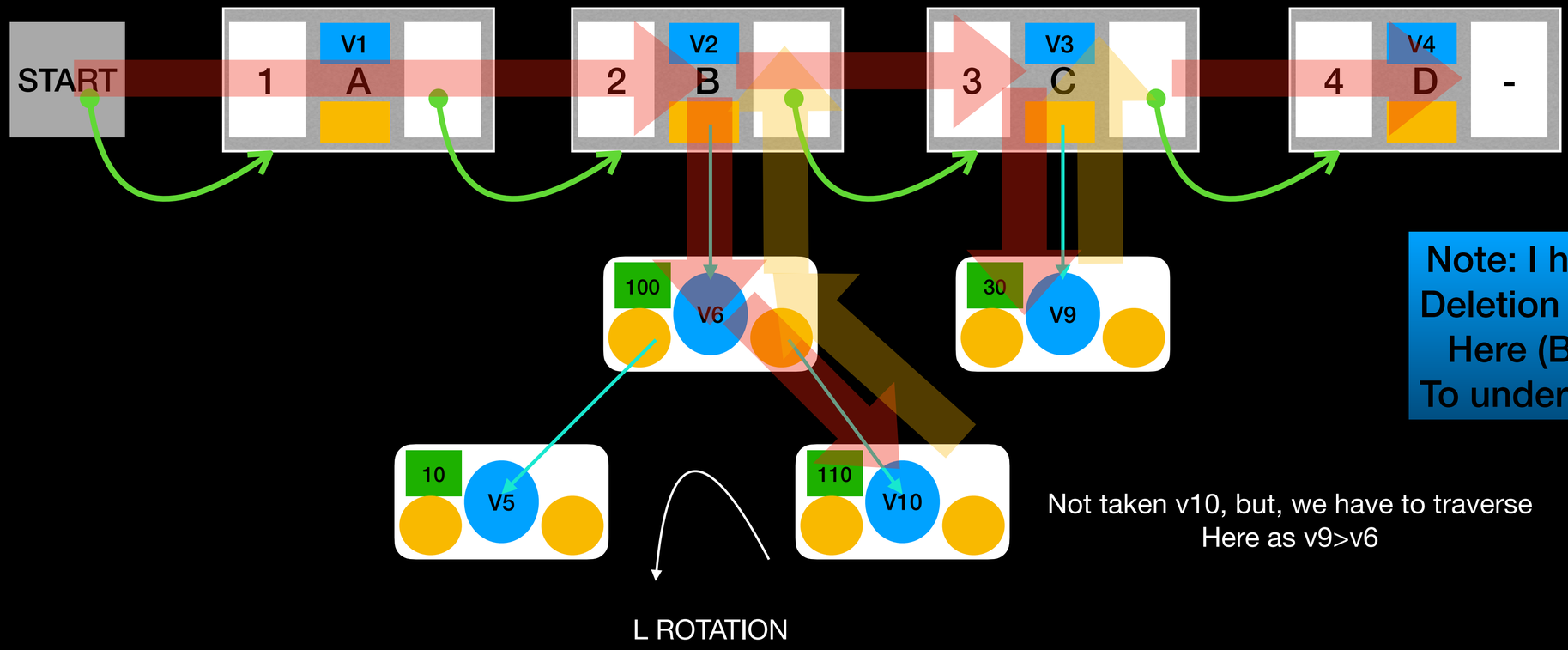
Normal Implementation (using balanced BST)



Why such approach is better ? 🥲

Normal Implementation (using balanced BST) (Reading v9)

Output: 1 -> 100 -> 30 -> 4



Not taken v10, but, we have to traverse
Here as v9>v6

Note: I have not considered the Deletion and Insertion operation Here (But, this is not relevant To understand the performance)

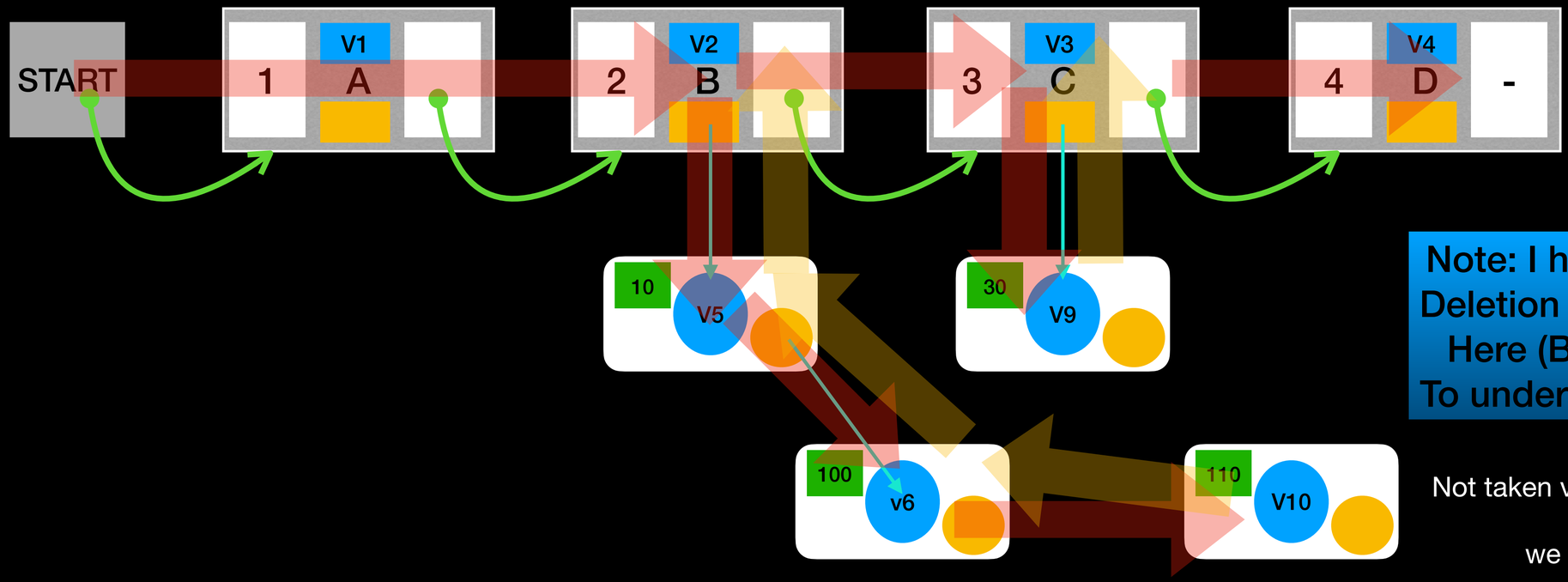
So, the time complexity of reading list v9 … n * O(log m)
n is length of LL and m is is #mods per node

Also the creation of the AVL of version to node Takes O(log m) each time

Why such approach is better ? 🥲

Normal Implementation (Using List) (Reading v9)

Output: 1 -> 100 -> 30 -> 4



Note: I have not considered the Deletion and Insertion operation Here (But, this is not relevant To understand the performance)

Not taken v10, but, we have to traverse Here as v9>v6, we take v6 from this line

So, the time complexity of reading list v9 … n * O(m)
n is length of LL and m is is #mods per node

The creation of the List of version to node Takes O(1) each time

Why such approach is better ? 🥲

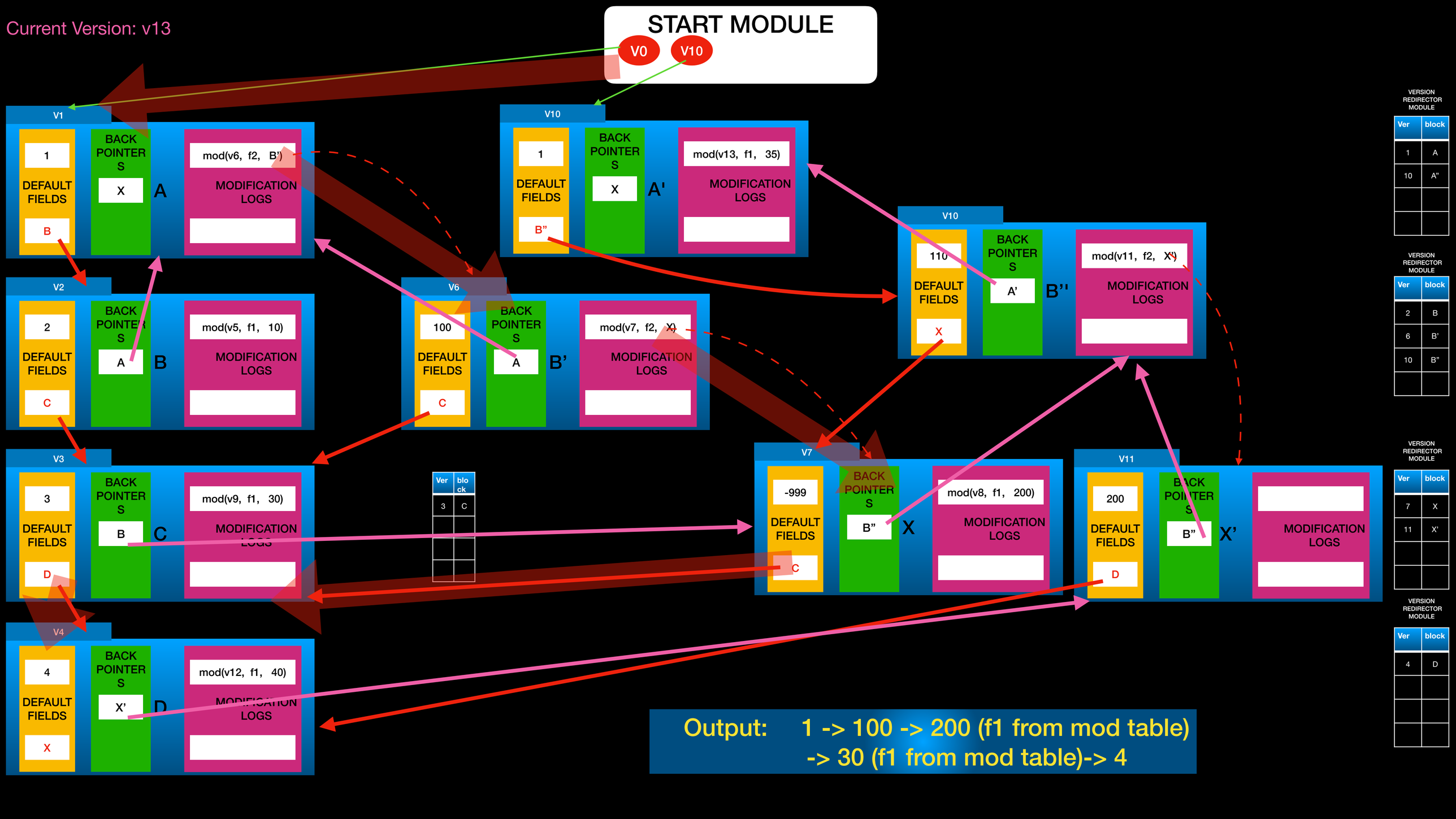
Pointer Machine (Reading v9)

We are taking O(2k.n)->O(n) time for iterating over the Linked List
Here k is the in-degree (Here 1) of the DS

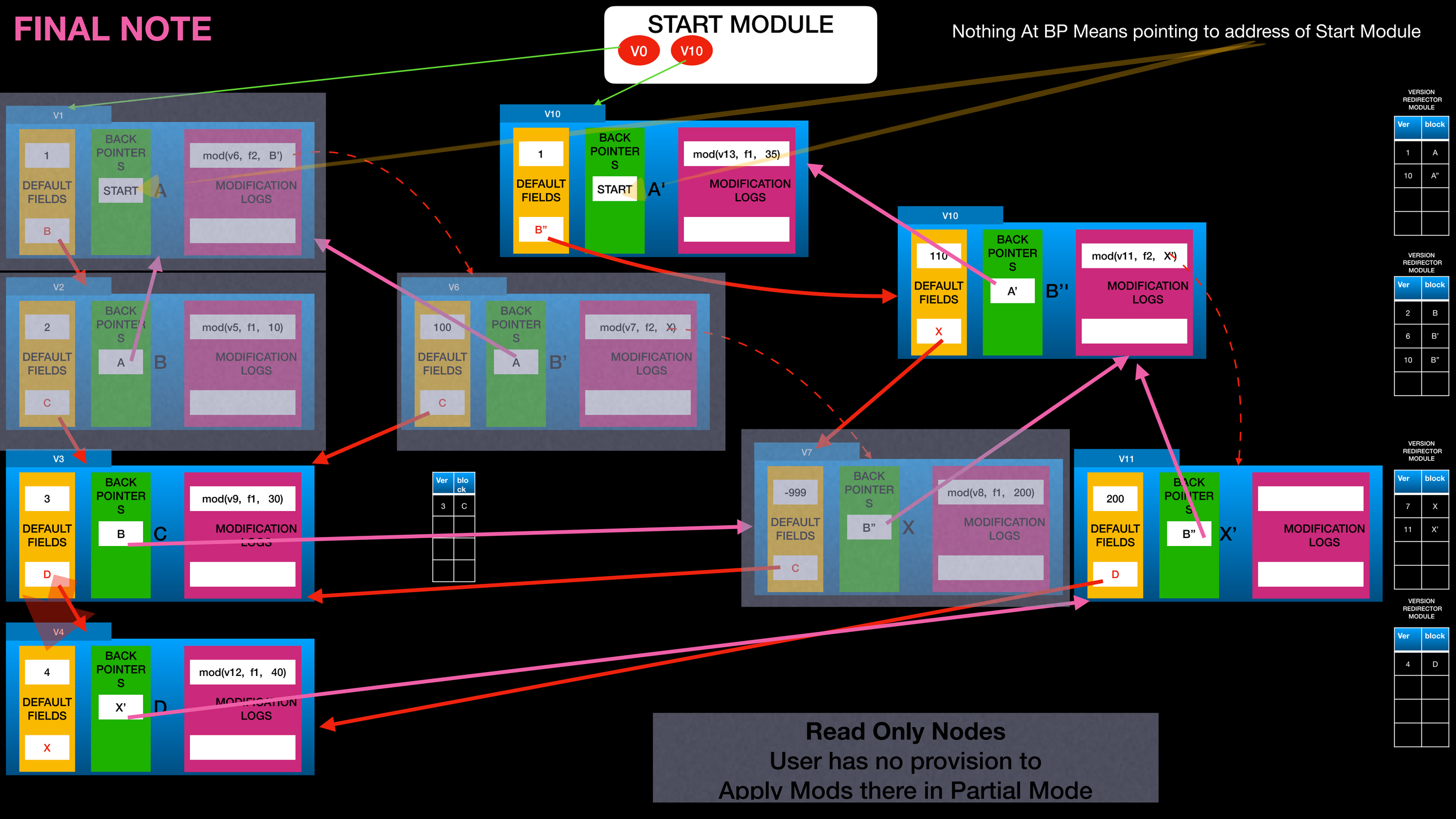
We are taking O(2) **amortised time** for modifying/adding each node to the Linked List
Here k is the in-degree (Here 1) of the DS

Simulation is shown at next page …

Current Version: v13



FINAL NOTE


START MODULE
V0
V10
Nothing At BP Means pointing to address of Start Module
V1
1
DEFAULT FIELDS
B
BACK POINTERS
START
A
mod(v6, f2, B')
MODIFICATION LOGS
V10
1
DEFAULT FIELDS
B"
BACK POINTERS
START
A'
mod(v13, f1, 35)
MODIFICATION LOGS
V10
110
DEFAULT FIELDS
X
BACK POINTERS
A'
B''
mod(v11, f2, X')
MODIFICATION LOGS
V2
2
DEFAULT FIELDS
C
BACK POINTERS
A
B
mod(v5, f1, 10)
MODIFICATION LOGS
V6
100
DEFAULT FIELDS
C
BACK POINTERS
A
B'
mod(v7, f2, X)
MODIFICATION LOGS
V3
3
DEFAULT FIELDS
D
BACK POINTERS
B
C
mod(v9, f1, 30)
MODIFICATION LOGS
V7
-999
DEFAULT FIELDS
C
BACK POINTERS
B"
X
mod(v8, f1, 200)
MODIFICATION LOGS
V11
200
DEFAULT FIELDS
D
BACK POINTERS
B"
X'
MODIFICATION LOGS
V4
4
DEFAULT FIELDS
X
BACK POINTERS
X'
D
mod(v12, f1, 40)
MODIFICATION LOGS
Ver
block
3
C
VERSION REDIRECTOR MODULE
Ver
block
1
A
10
A"
VERSION REDIRECTOR MODULE
Ver
block
2
B
6
B'
10
B"
VERSION REDIRECTOR MODULE
Ver
block
7
X
11
X'
VERSION REDIRECTOR MODULE
Ver
block
4
D
Read Only Nodes
User has no provision to
Apply Mods there in Partial Mode

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:

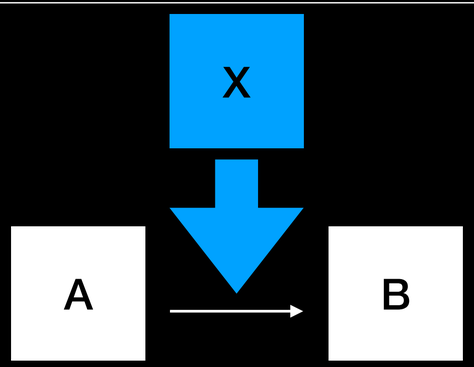
- start = init() : To initiate linked list and “start” pointer holds the starting position in v0
- add(x, y, a, v) : Add new node x after y at version v with f1 = a and f2 = NULL, and update the version in version tree.
- create_node(x, a): Allocate a new node x with value = a, and set its default version = 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

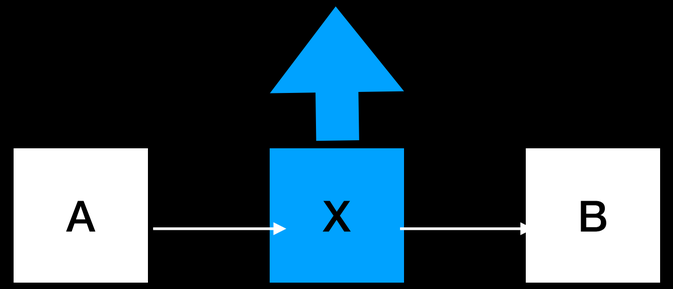
Create node X with value 123
Modify f2 of A to X : update(f2, A, X,v)
Modify f2 of X to C : update(f2, X, C,v)
Modify BP1 of X to A: update(bp1, X, A,v)
Modify BP1 of B to X: update(bp1, B, X,v)
~~Set f1 of X(optional)~~
Add current ver to the respective position in version tree



remove(x) consists of

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

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



Elementary Operations:

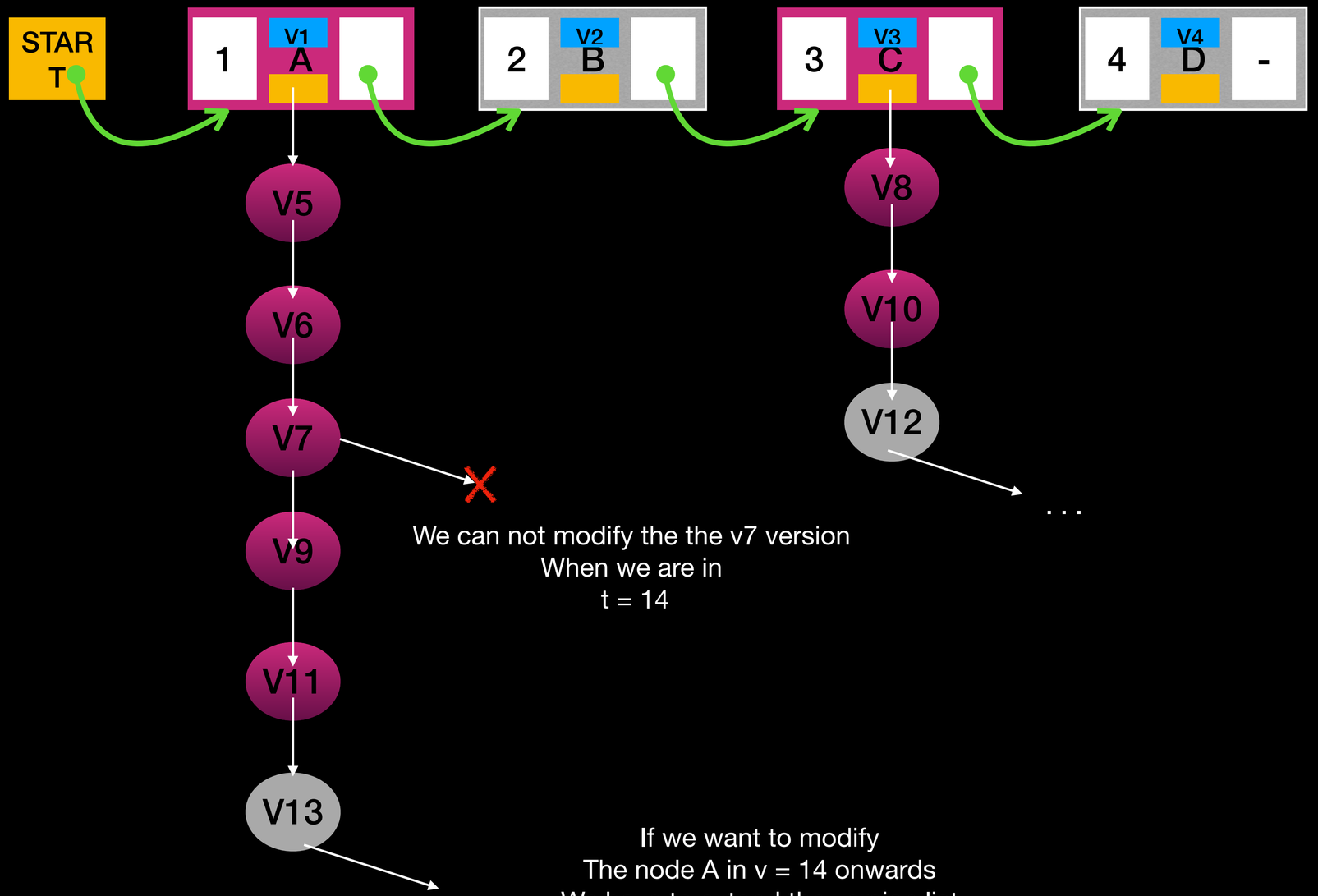
- start = init()
- create_node(x, a)
- iterate_over_LL(v)
- update(x,f_i,val,v)

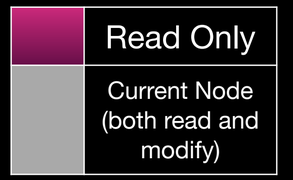
Why we need Full Persistent Data-structure?

Basic Idea

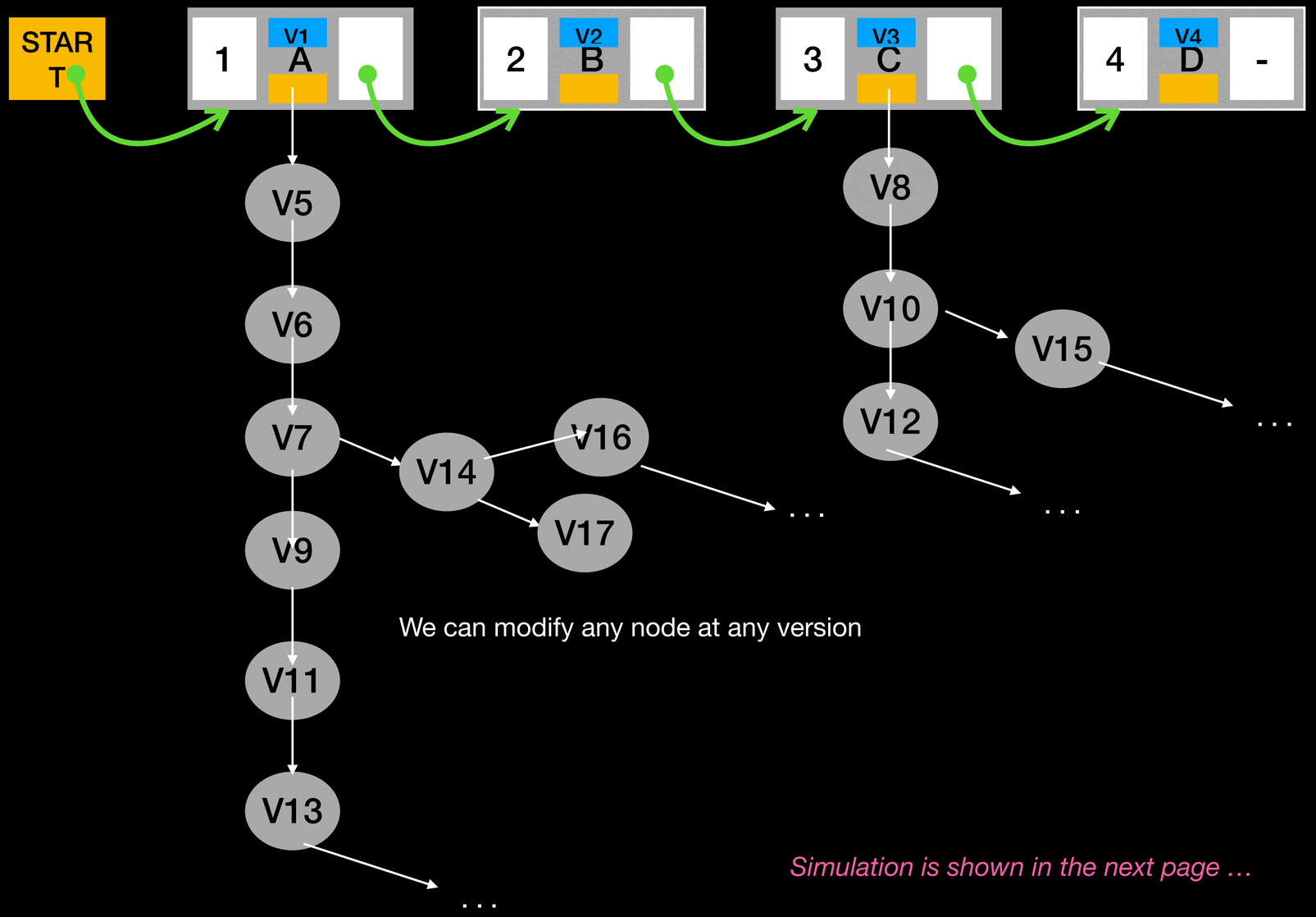
- Now we can modify any pervious version.
- Branching of versions - is possible
- In Partial Persistent Data Structure we saw Linear Ordering of versions, pervious versions were in read-only state. We could modify the latest version of any node.
- But, in Full Persistent Mode, we can branch the version 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 …

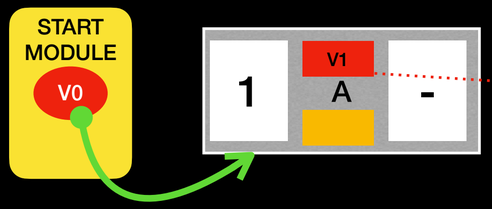
	Read Only
	Current Node (both read and modify)

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

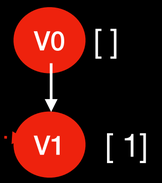
Query: add(A,_,v0,1)

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



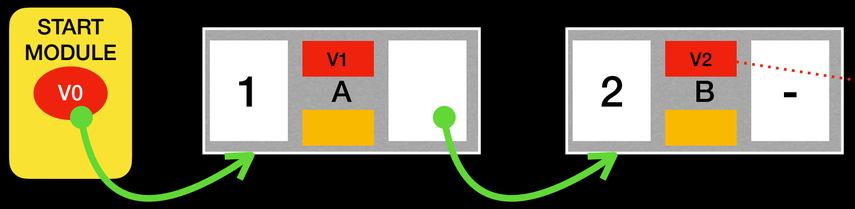
Current time, t = 1

Version tree



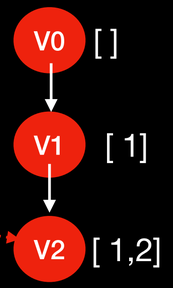
Query: add(B,A,v1,2)

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

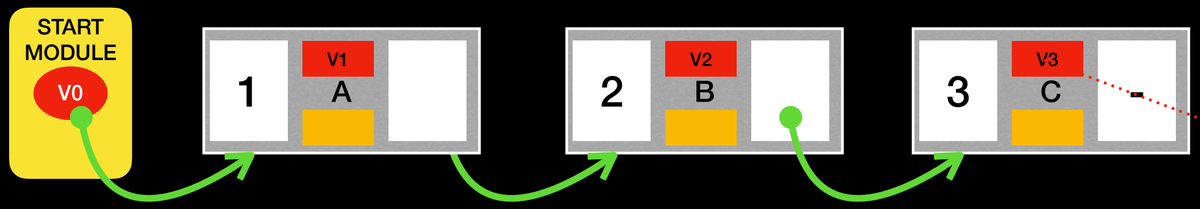


Current time, t = 2

Version tree



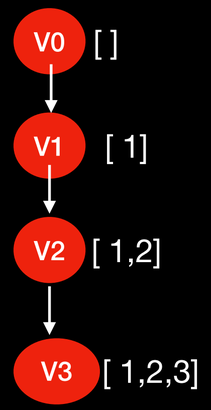
Query: add(C,B,v2,3)



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

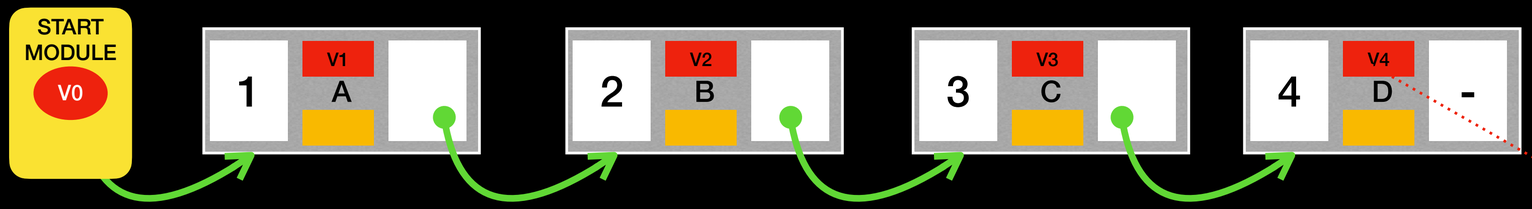
Current time, t = 3

Version tree



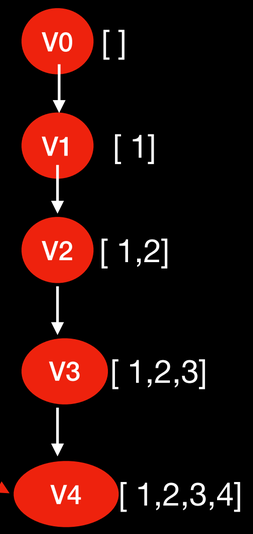
Query: add(D,C,v3,4)

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



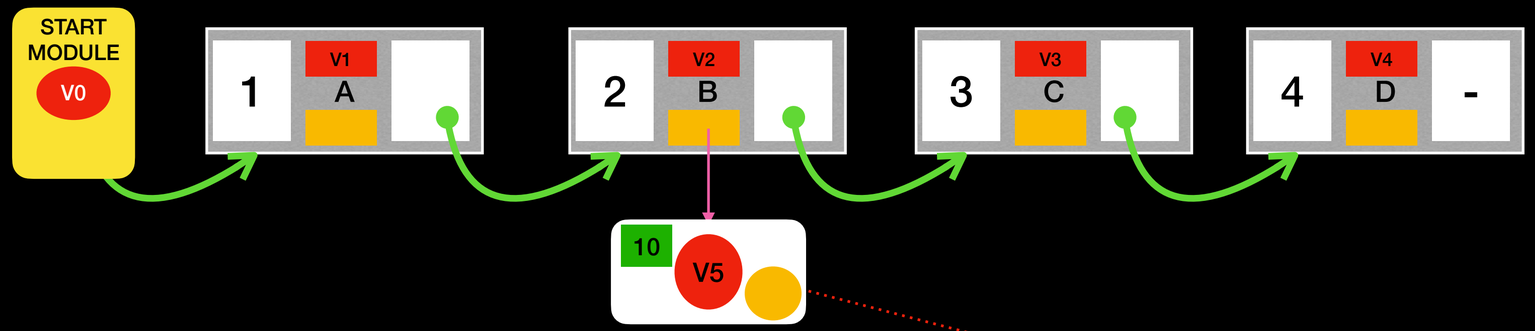
Current time, t = 4

Version tree



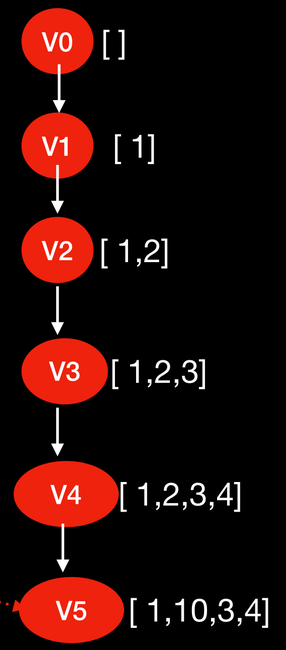
Query: update(B,f1,10,v4)

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



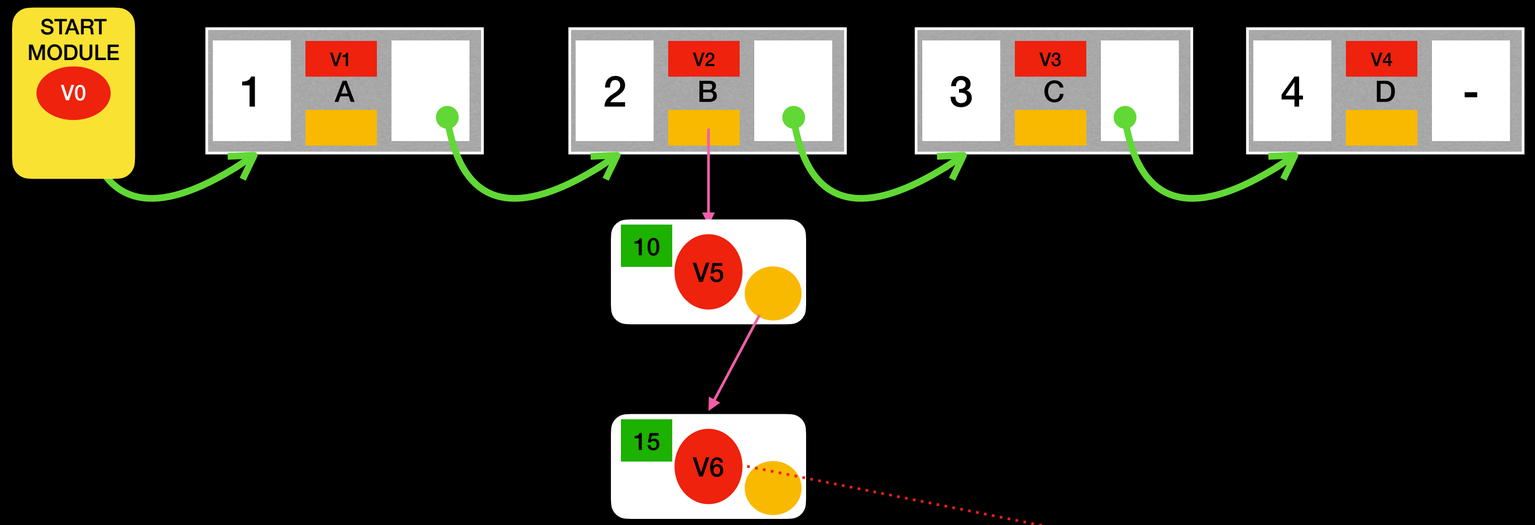
Current time, t = 5

Version tree



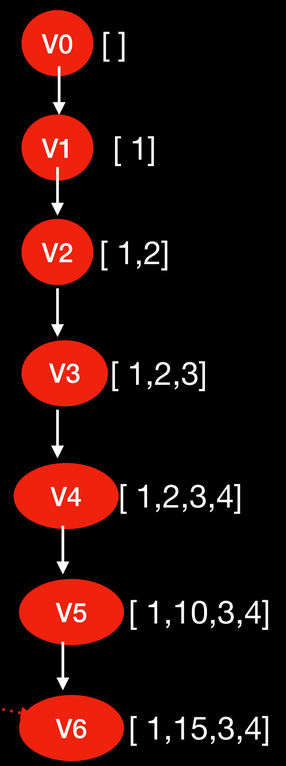
Query: update(B,f1,15,v5)

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



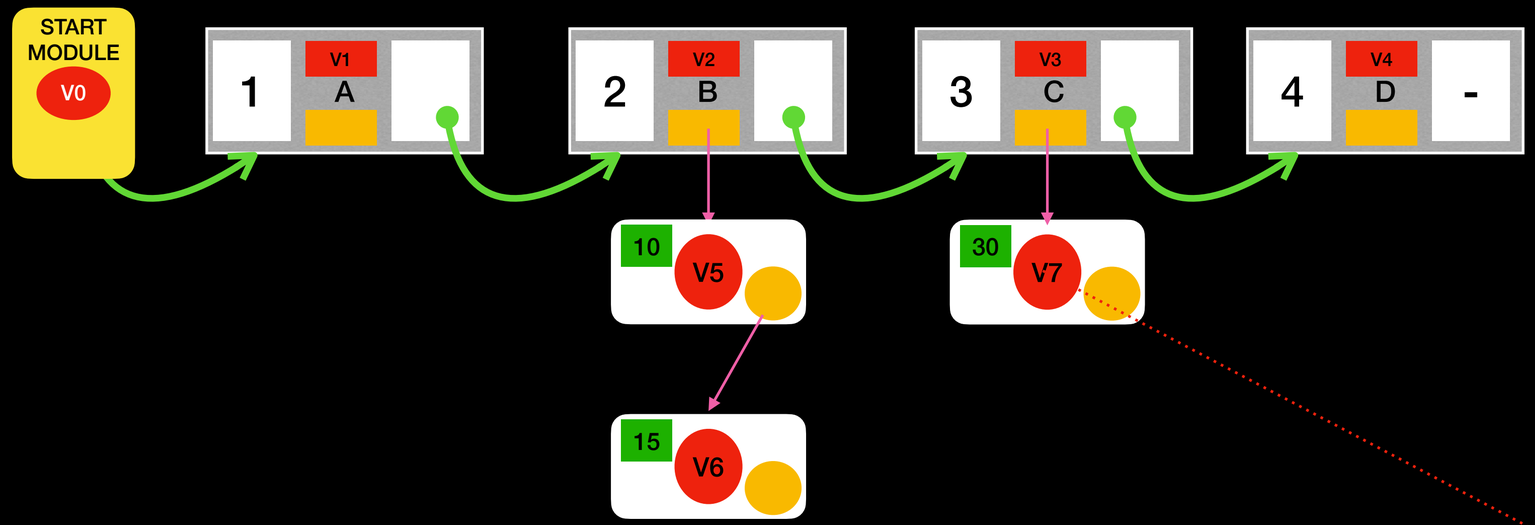
Current time, t = 6

Version tree



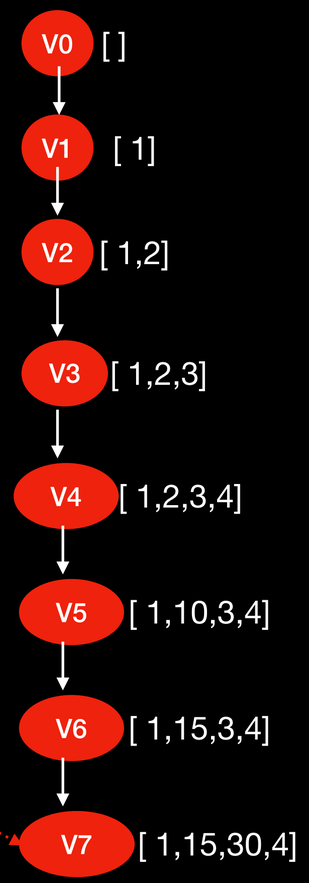
Query: update(C,f1,30,v6)

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

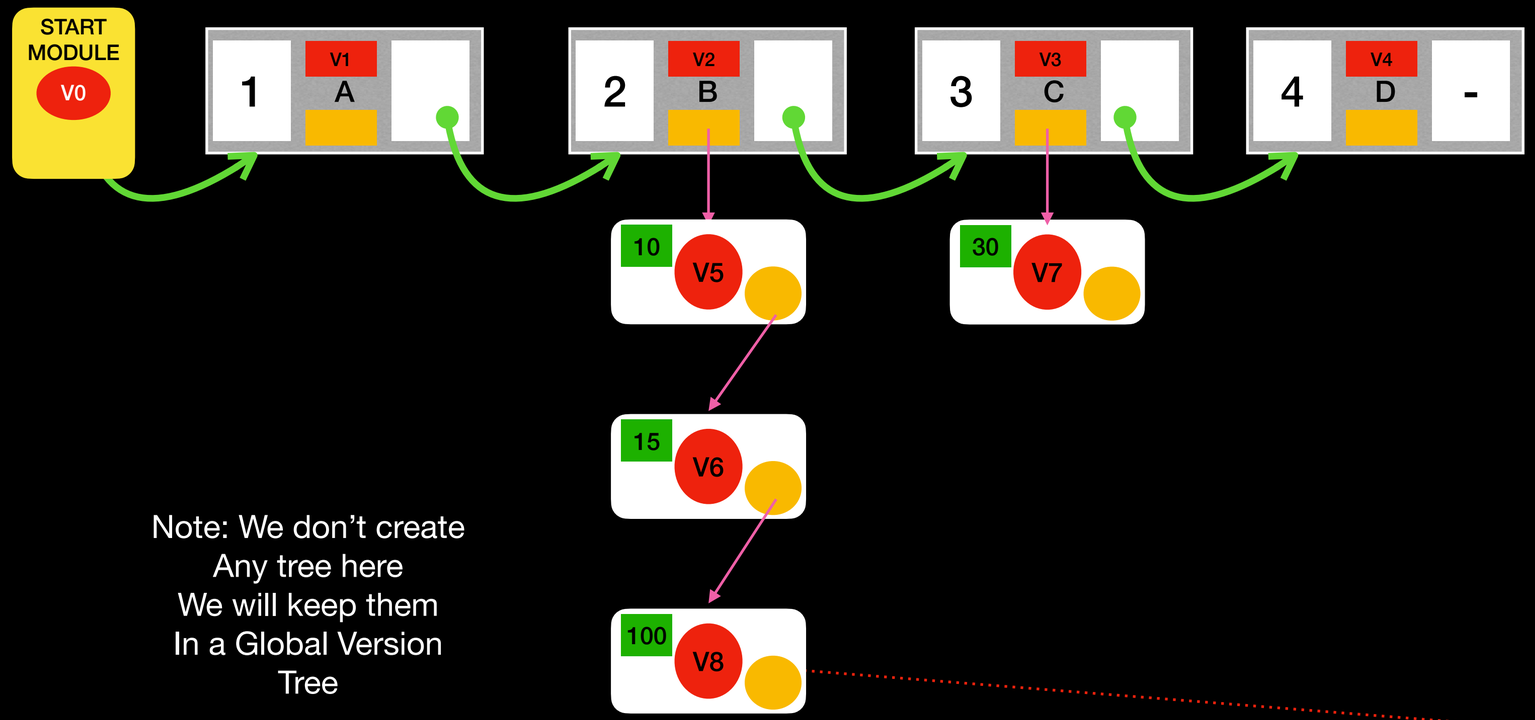


Current time, t = 7

Version tree



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

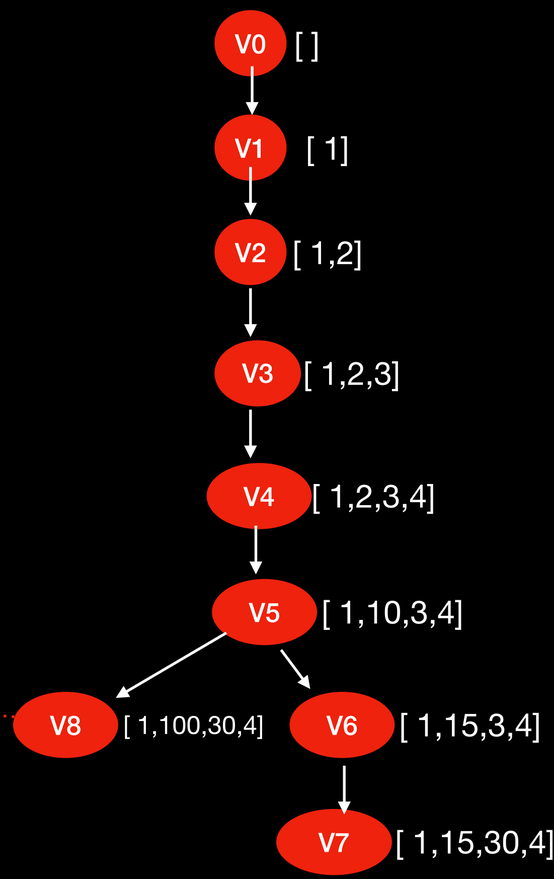


Note: We don't create Any tree here We will keep them In a Global Version Tree

As, That should be Accessible to all nodes

Current time, t = 8

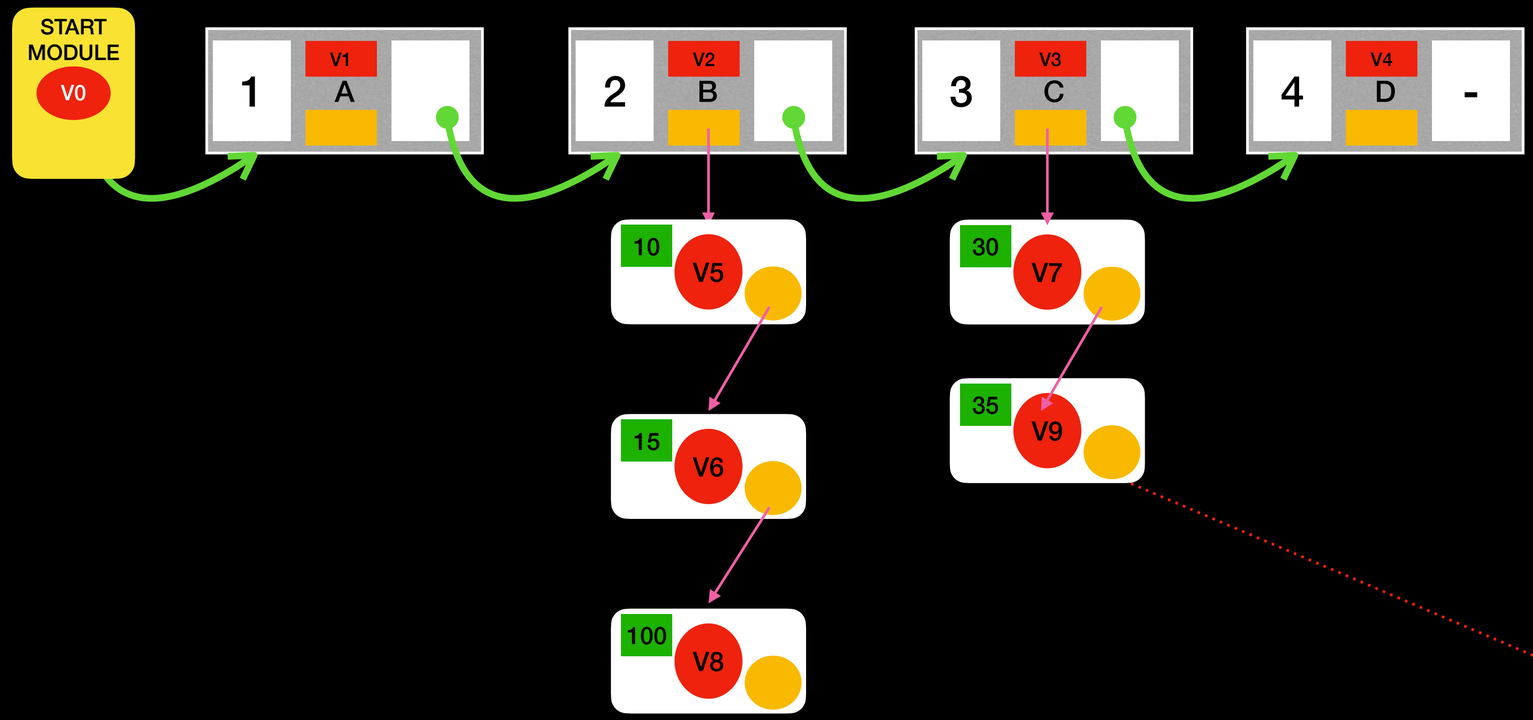
Version tree



How to do branching ?

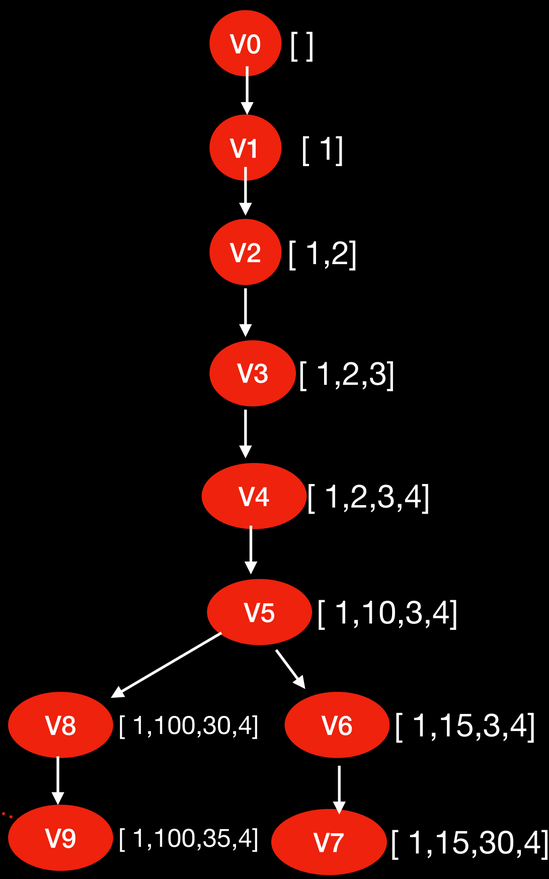
1. Create new Update Module with version = v8 and updated fields
2. Just add v8 node to v5 node in v-tree

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

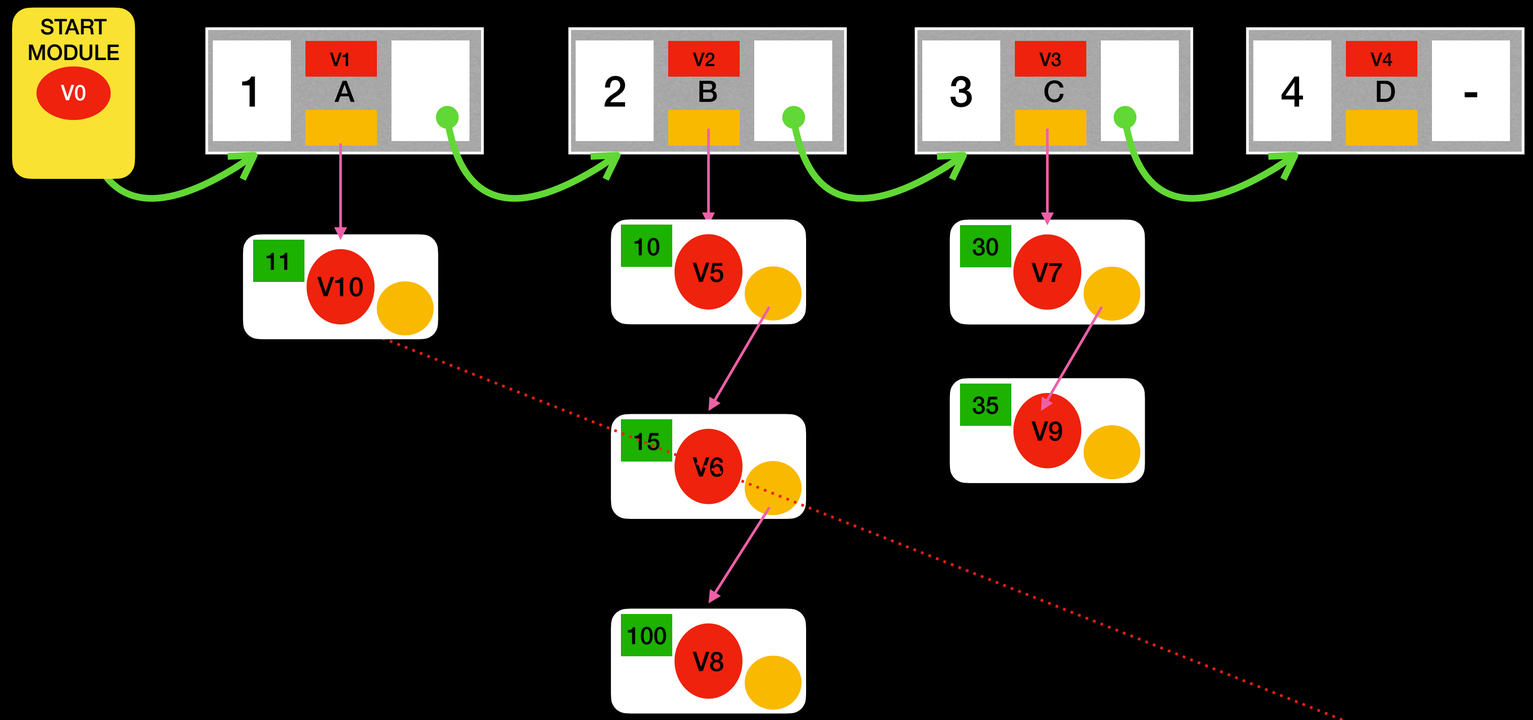


Current time, t = 9

Version tree

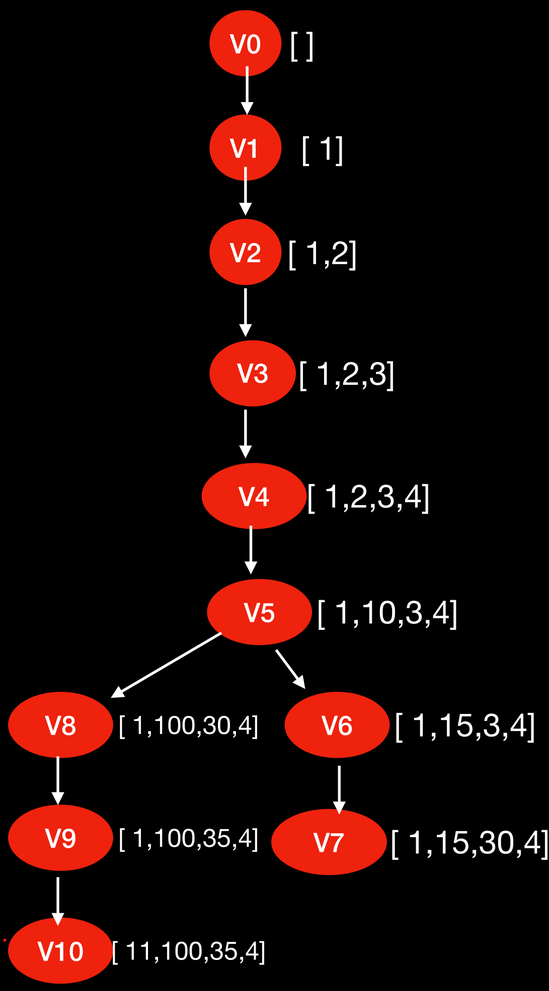


Query: update(A,f1,11,v9)

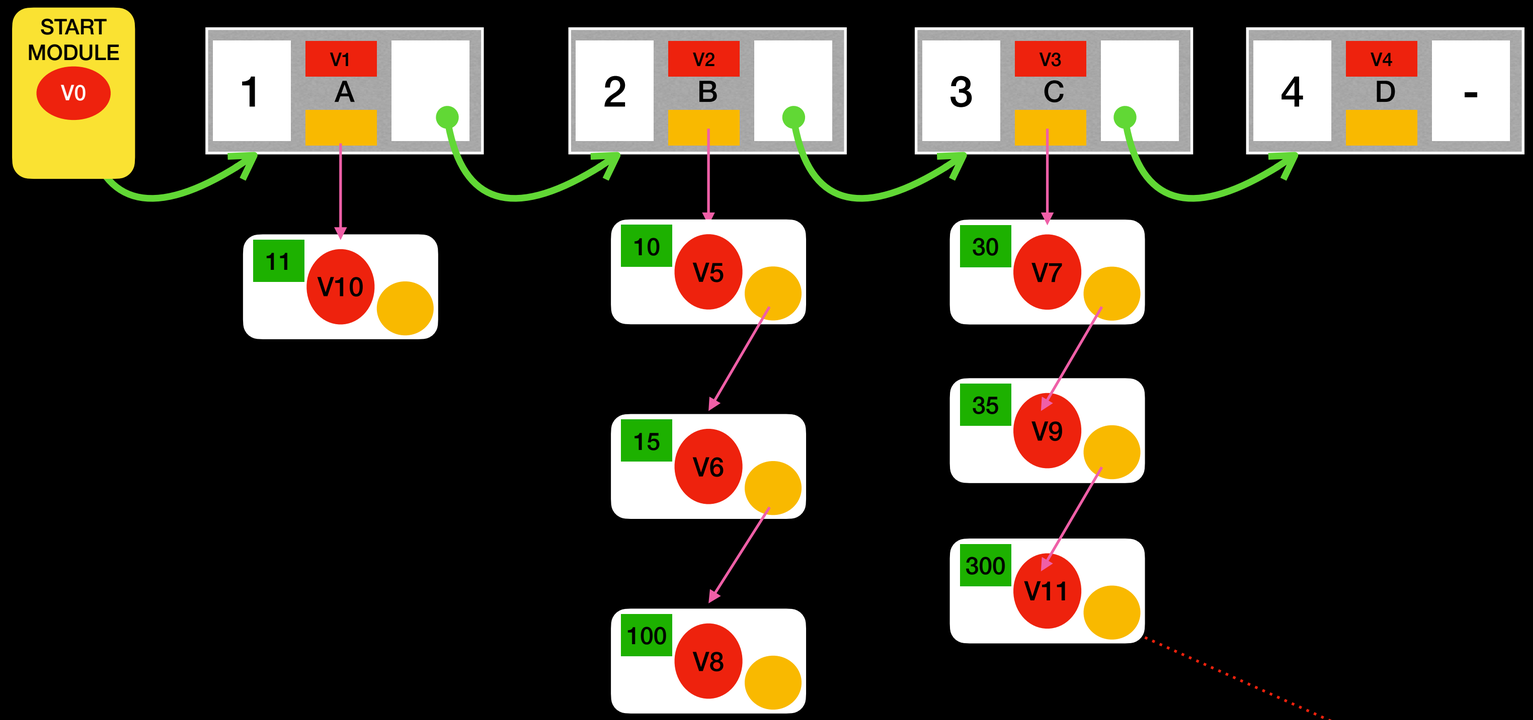


Current time, t = 10

Version tree

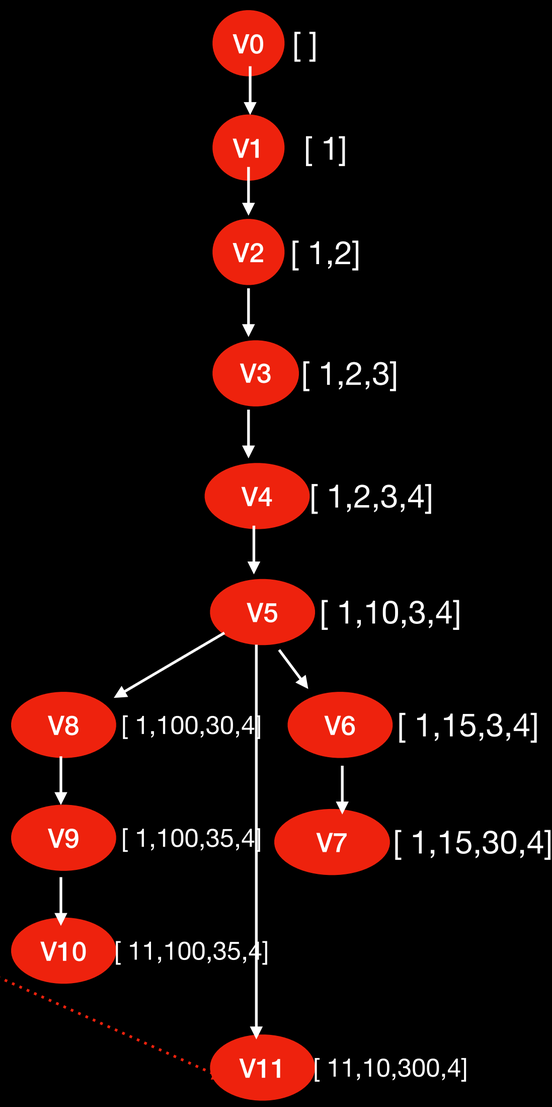


Query: update(C,f1,300,v5)

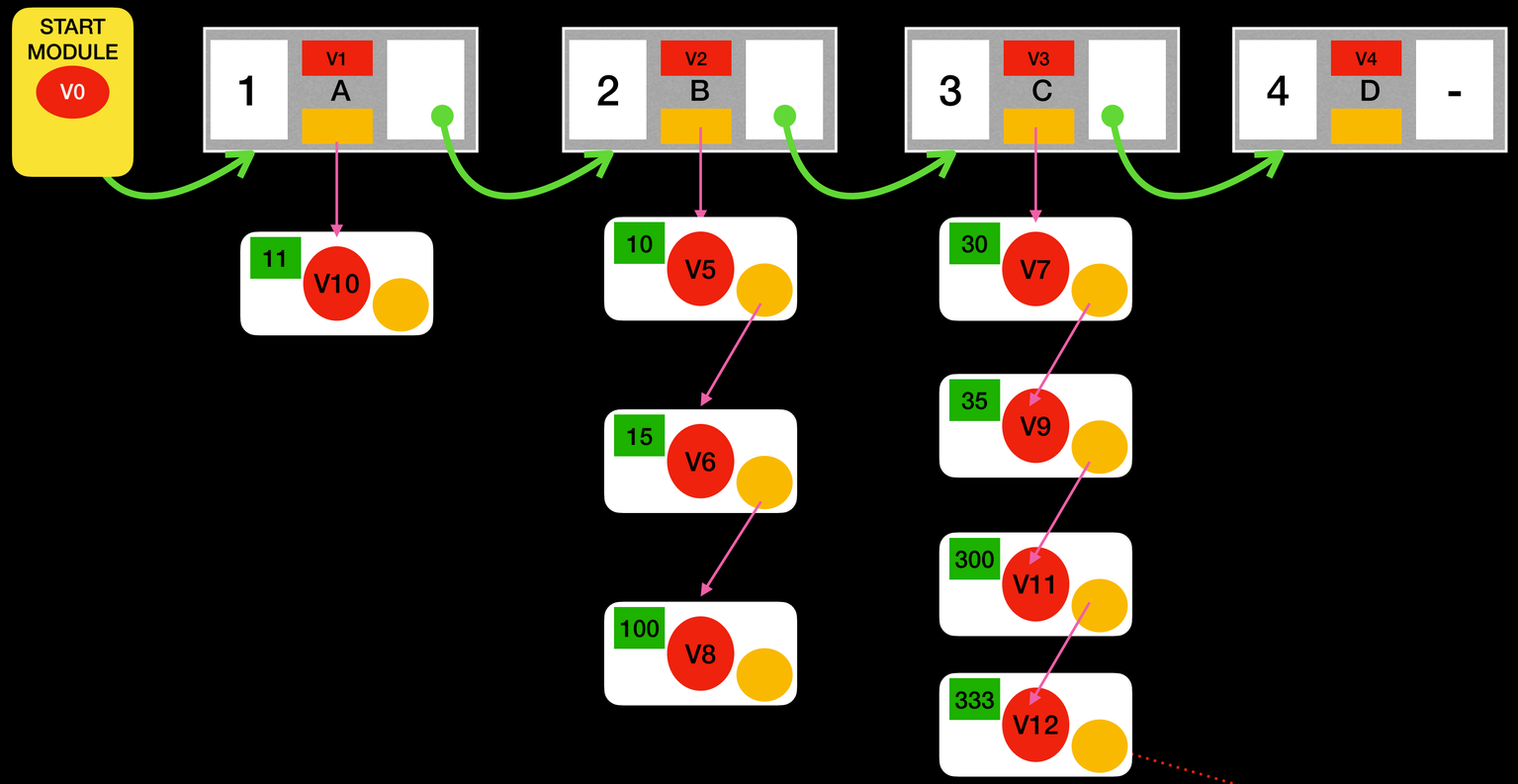


Current time, t = 11

Version tree

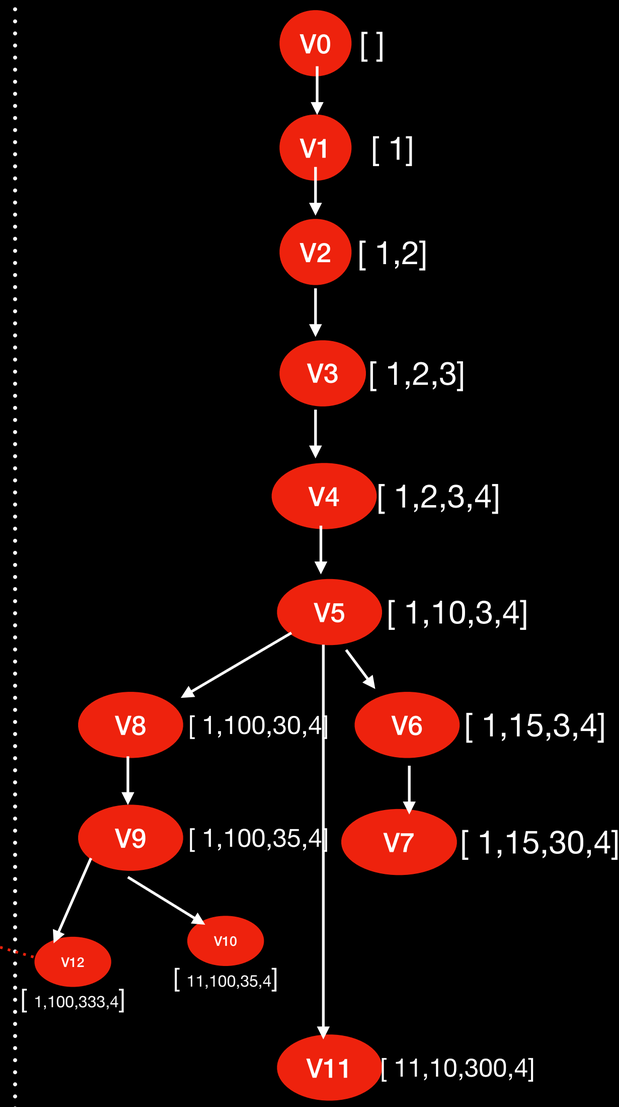


Query: update(C,f1,333,v9)

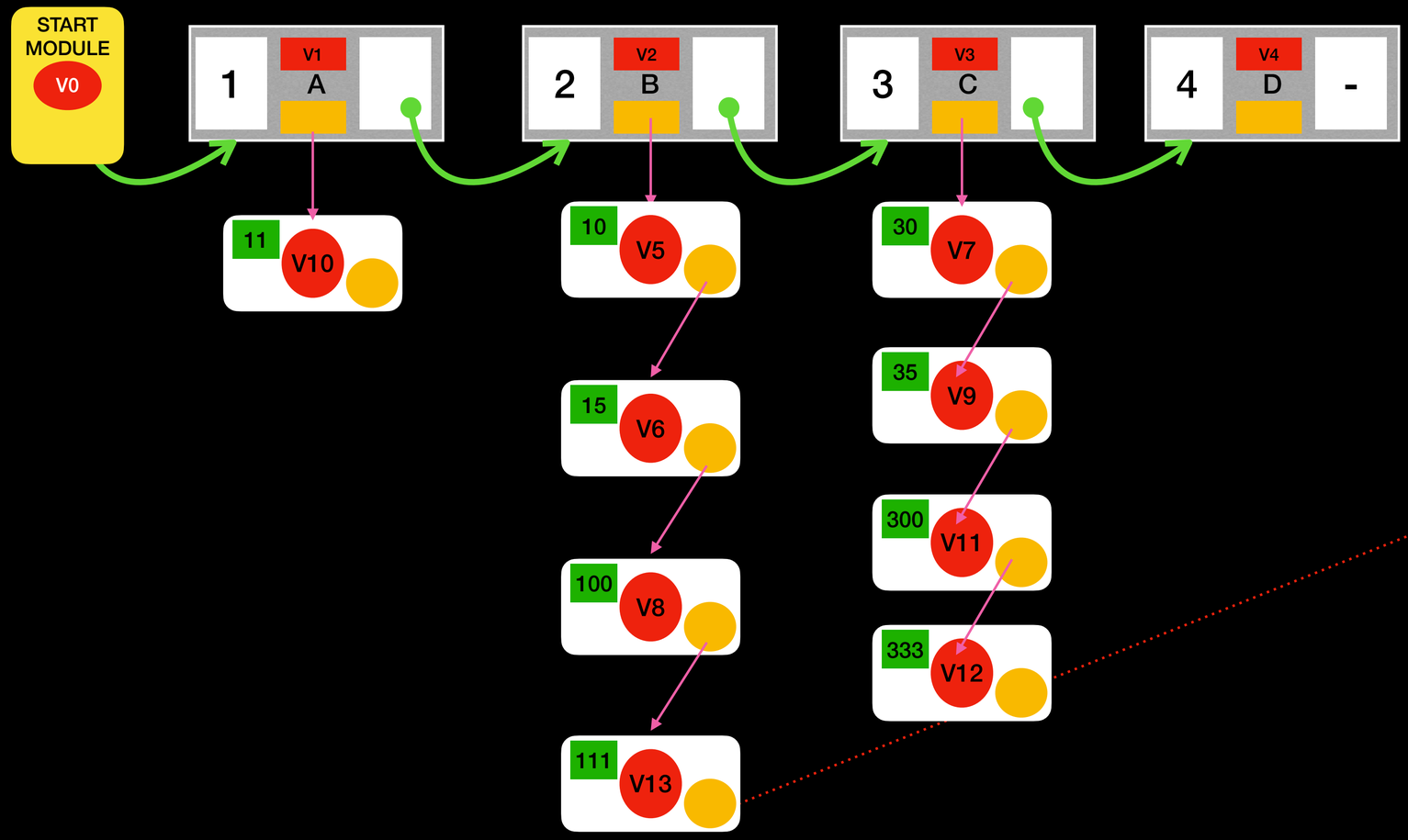


Current time, t = 12

Version tree

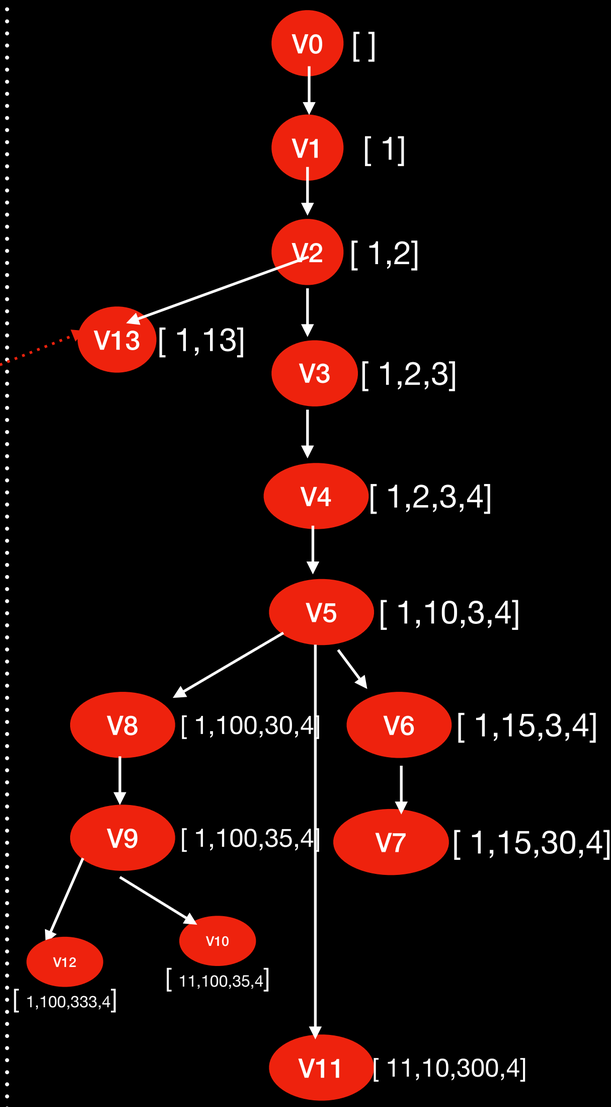


Query: update(B,f1,111,v2)

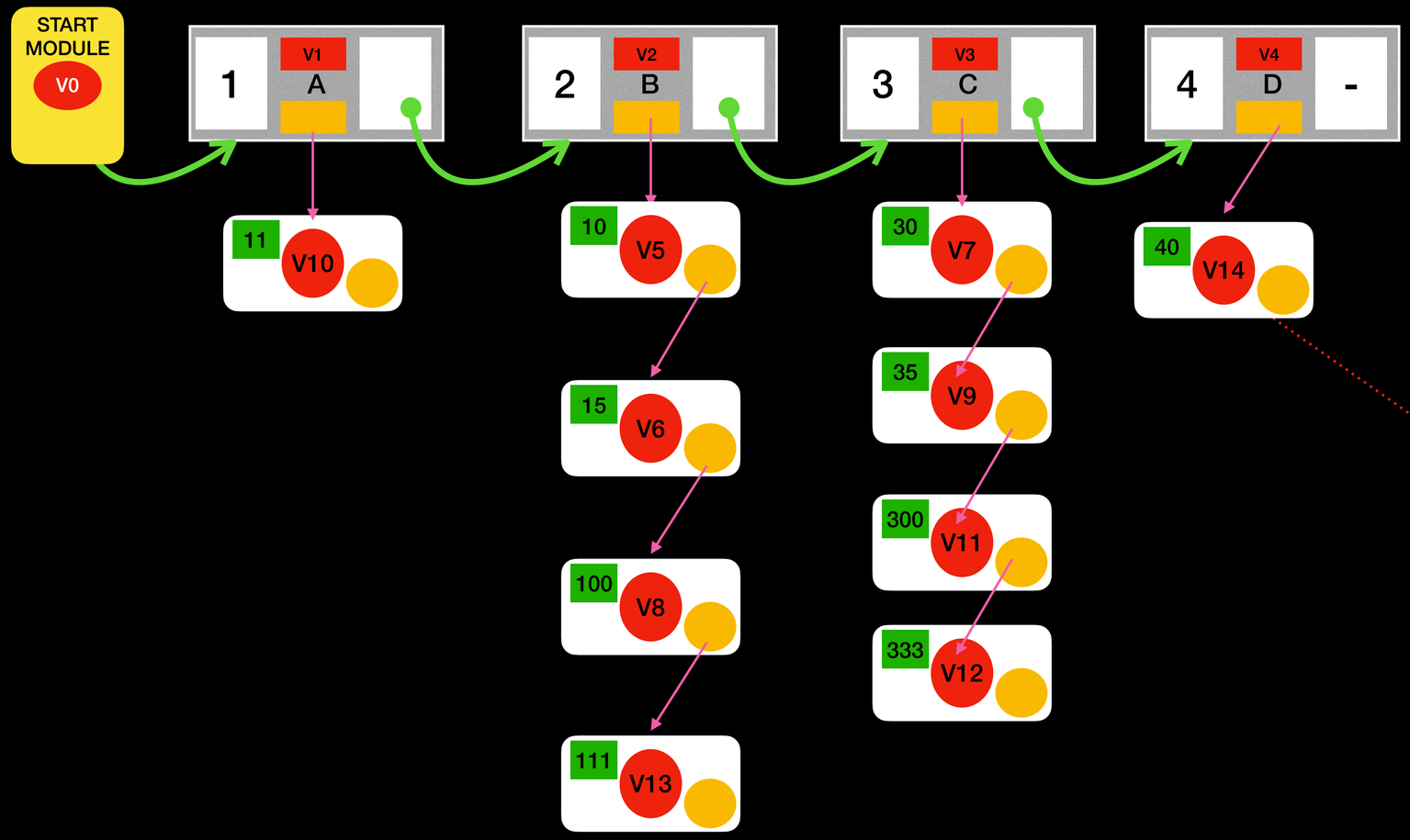


Current time, t = 13

Version tree

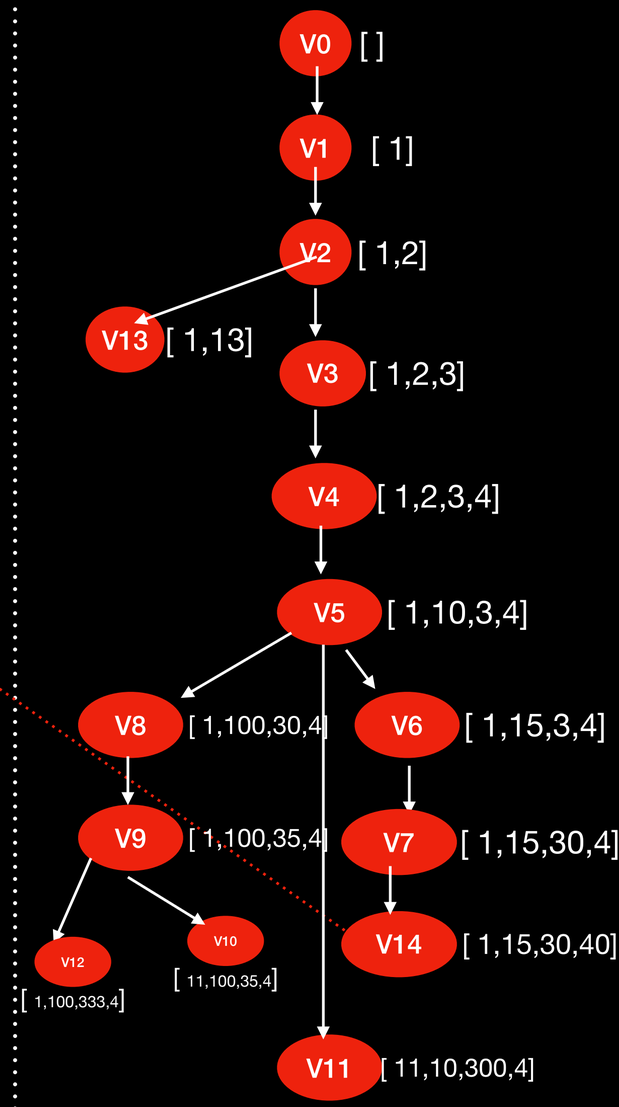


Query: update(D,f1,40,v7)

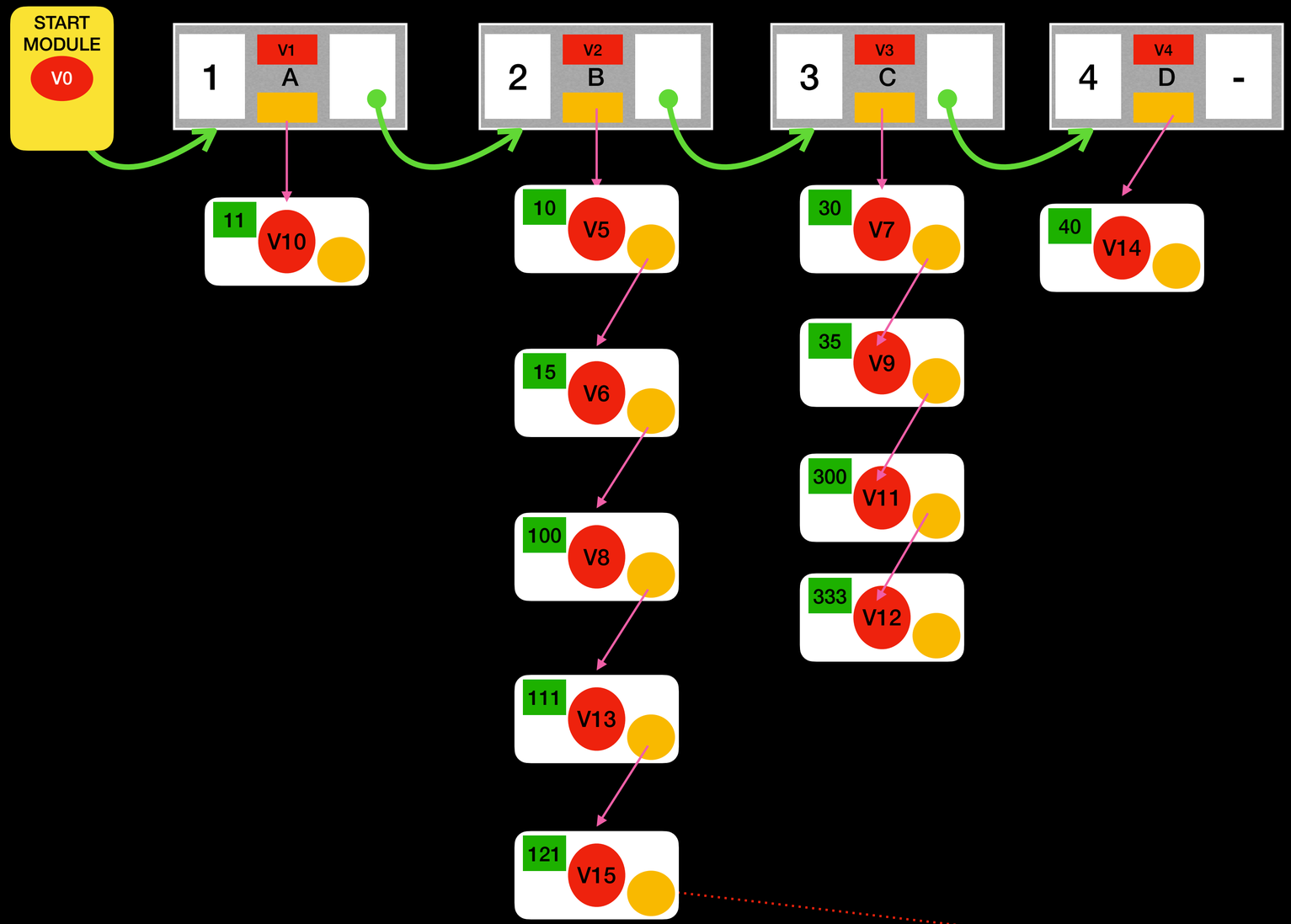


Current time, t = 14

Version tree

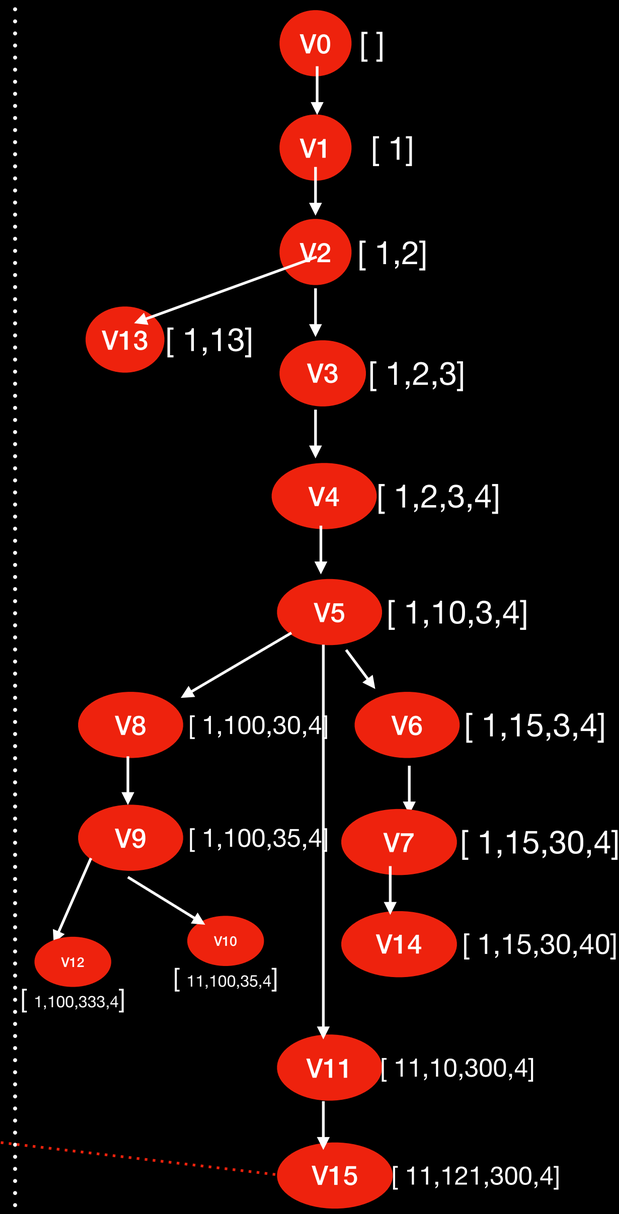


Query: update(B,f1,121,v11)

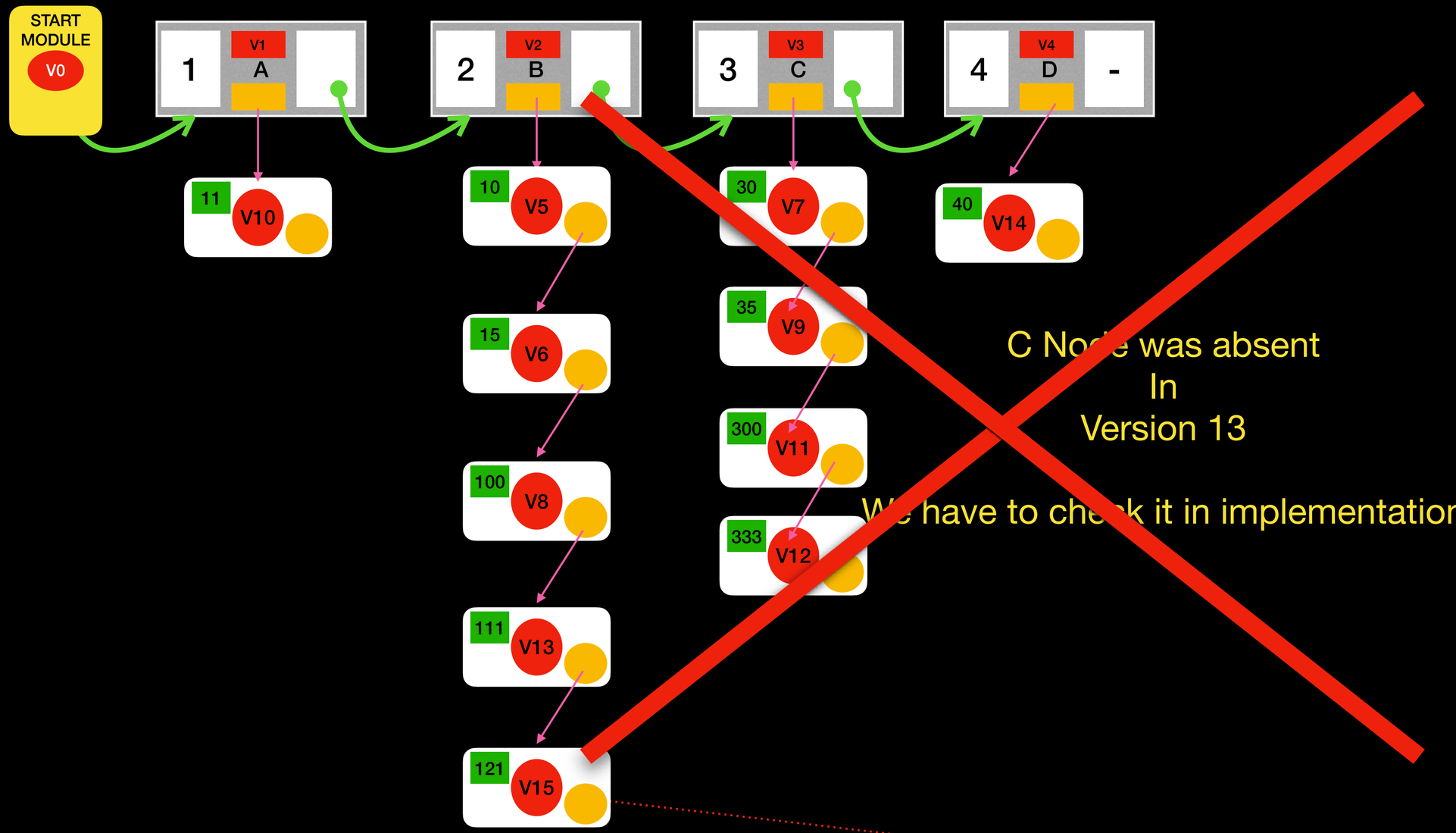


Current time, t = 15

Version tree

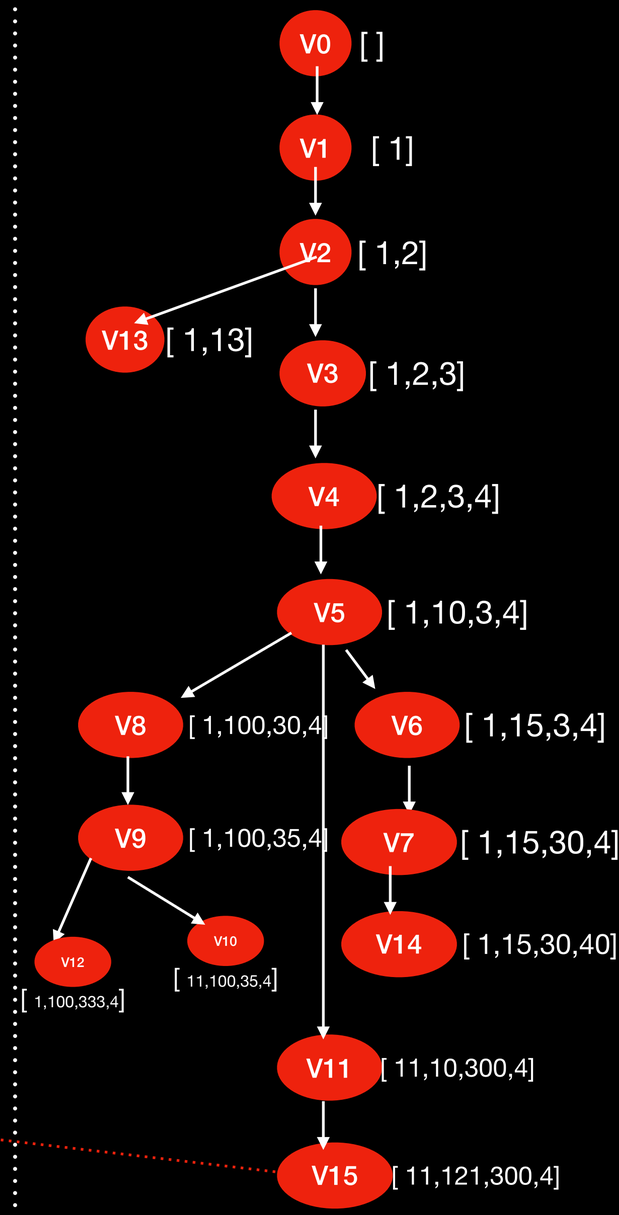


INVALID Query: update(C,f1,3000,v13)



Current time, t = 15

Version tree



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

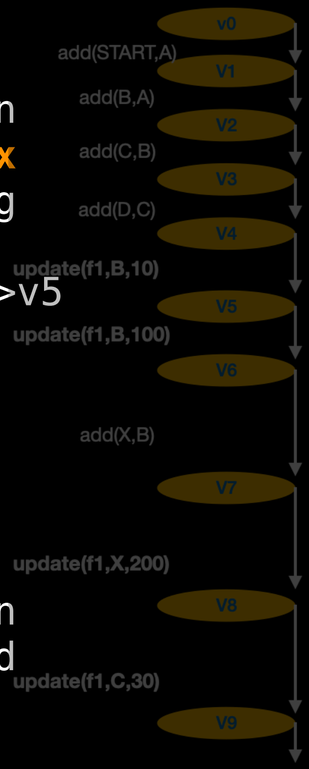
Iteration in Partial Mode vs Full Mode

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).



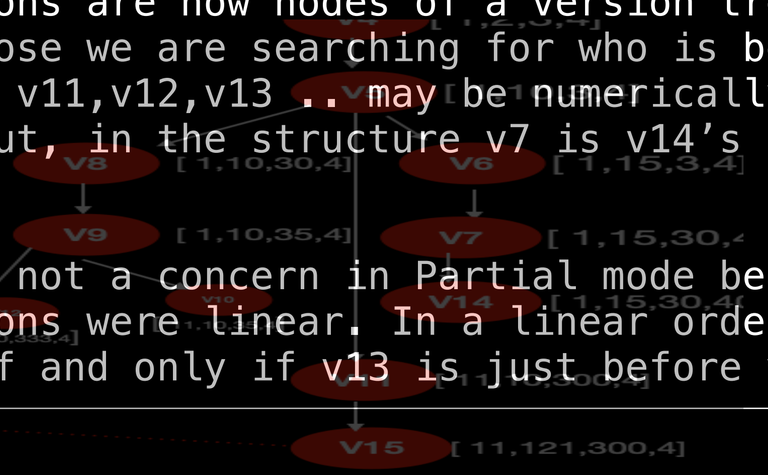
3. The word "JUST LESS" is written because if a NODE has two version lines in that, e.g. v2 and v4 and we are searching for v5, then we should prefer v4 line over v2.

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**

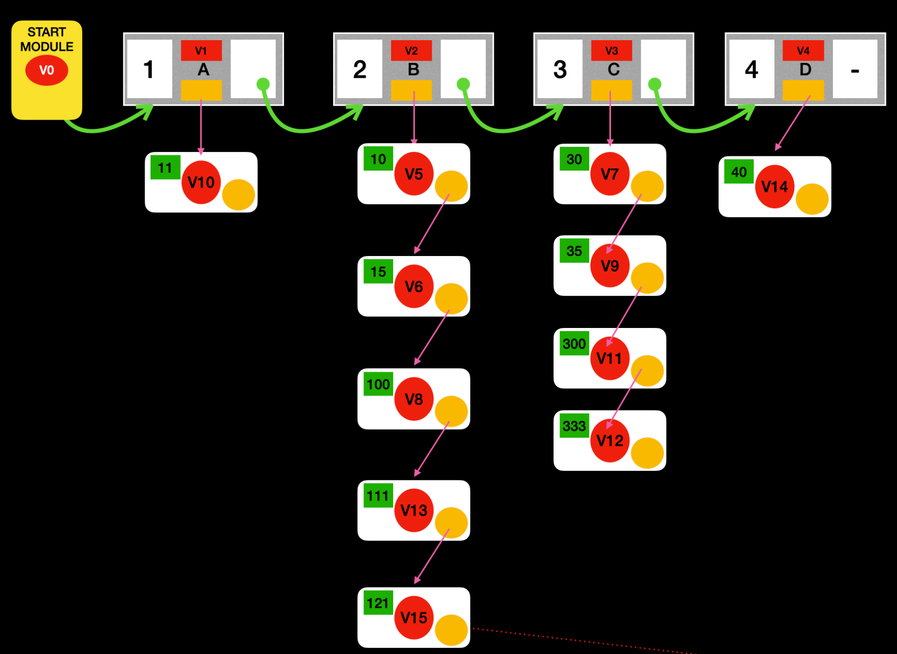
3. Here, we are searching for LOWEST ANCESTOR Because, versions are now not just number. The versions are now nodes of a version tree. E.g. suppose we are searching for who is before v14, here v11,v12,v13 .. may be numerically less than 14 but, in the structure v7 is v14's parent.

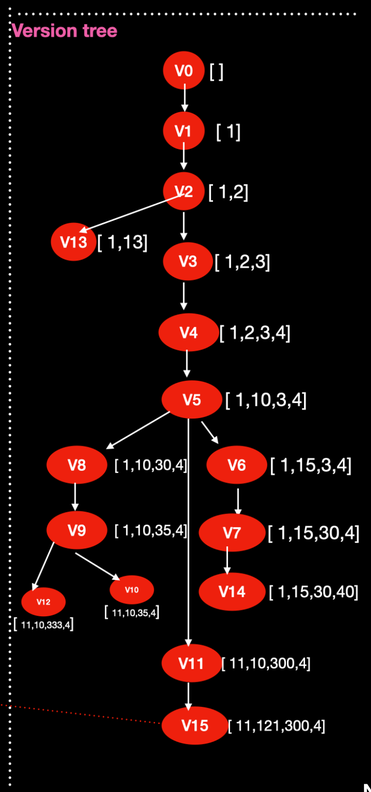


4. It was not a concern in Partial mode because the versions were linear. In a linear order v13<v14 if and only if v13 is just before v14.

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

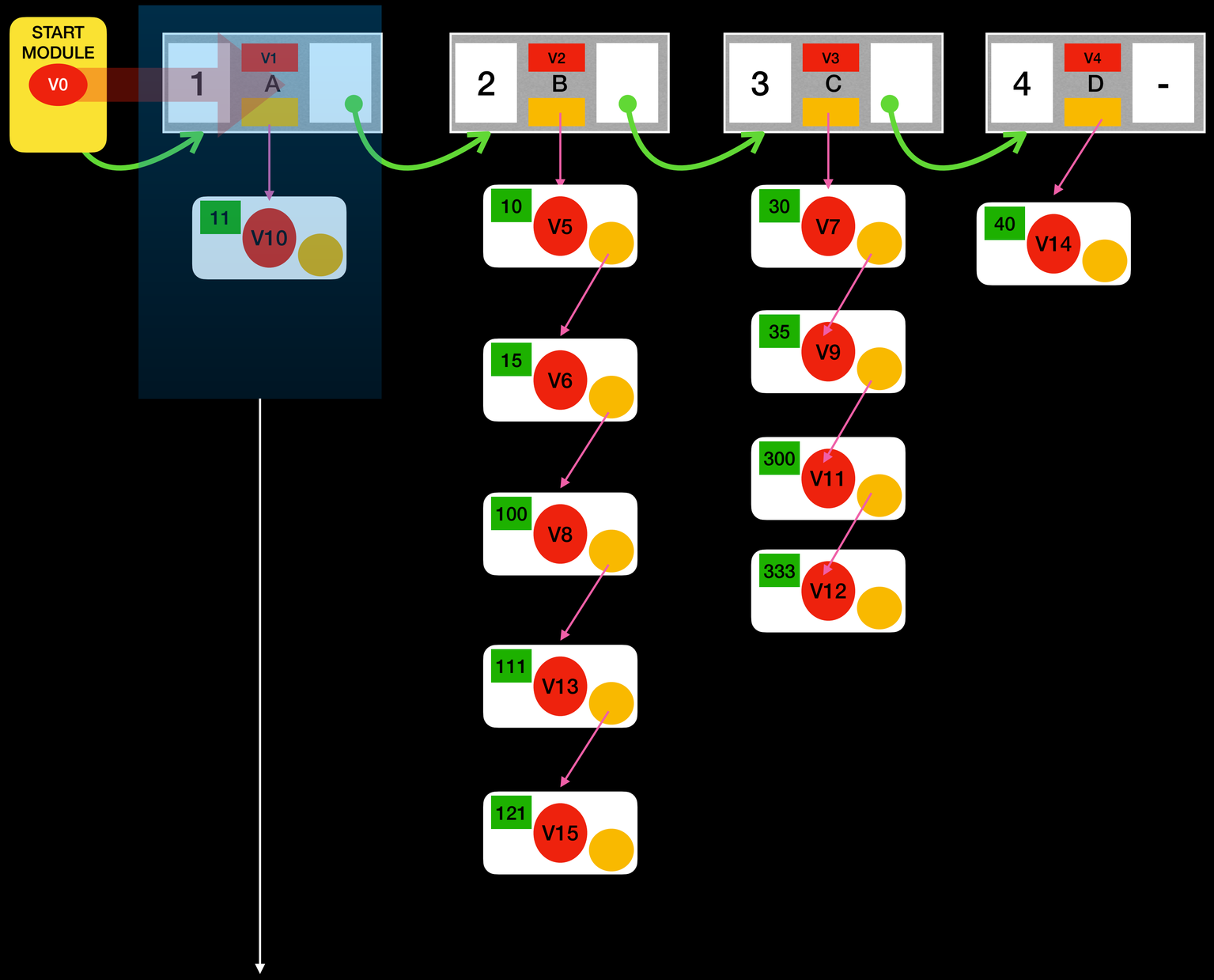
Full Persistent Data-structure

is a general form of

Partial Persistent Data-structure

iterate_LL_at_v(v12)

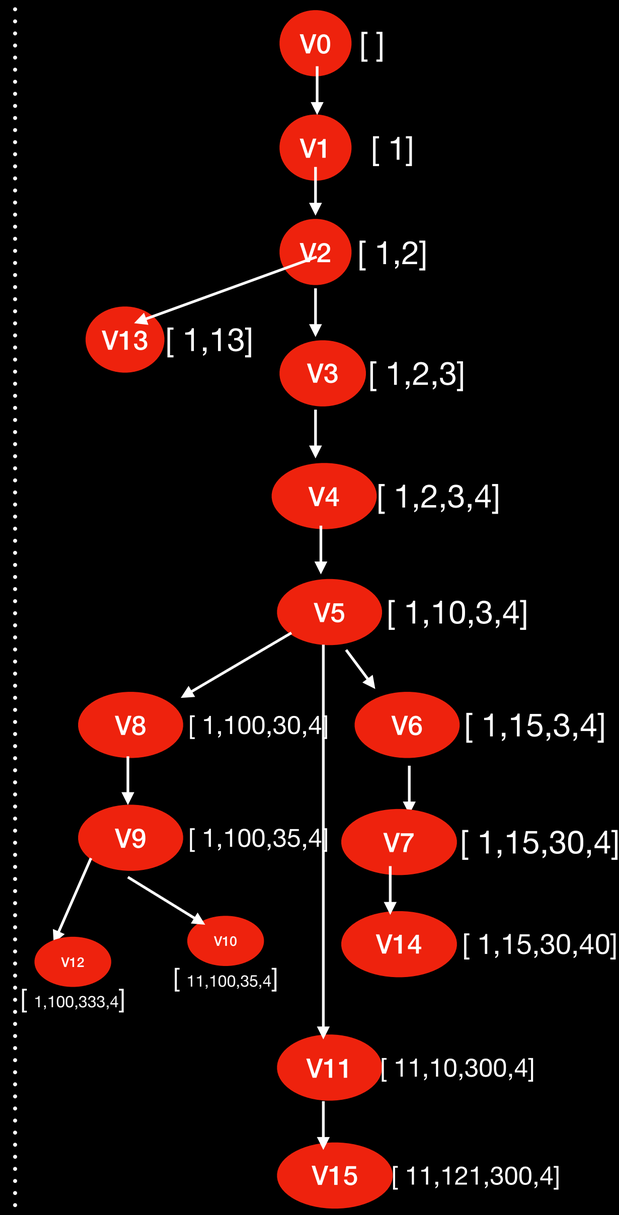
iterate_LL_at_v(v12)



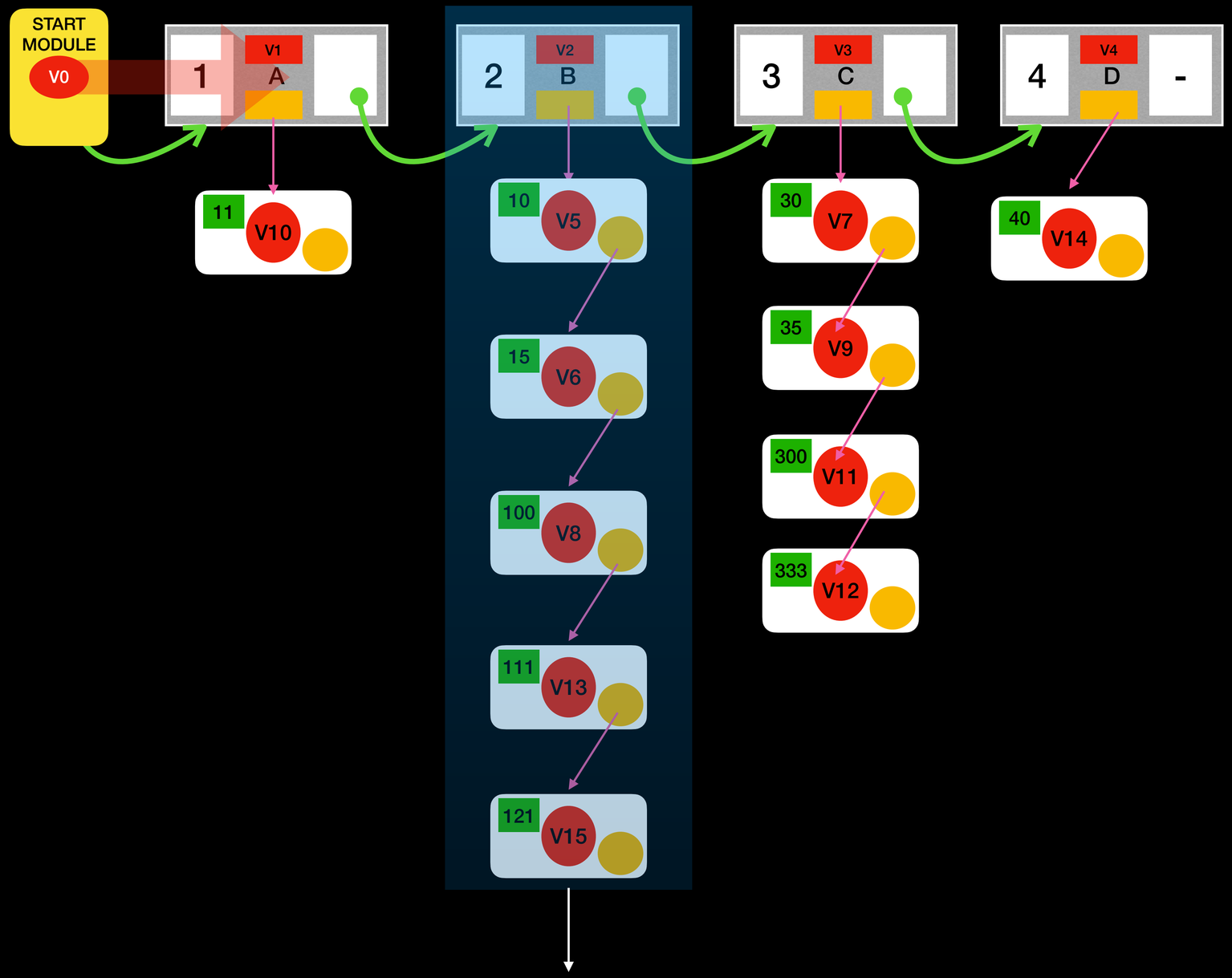
OUPTUT : 1(v1)

Current time, t = 15

Version tree



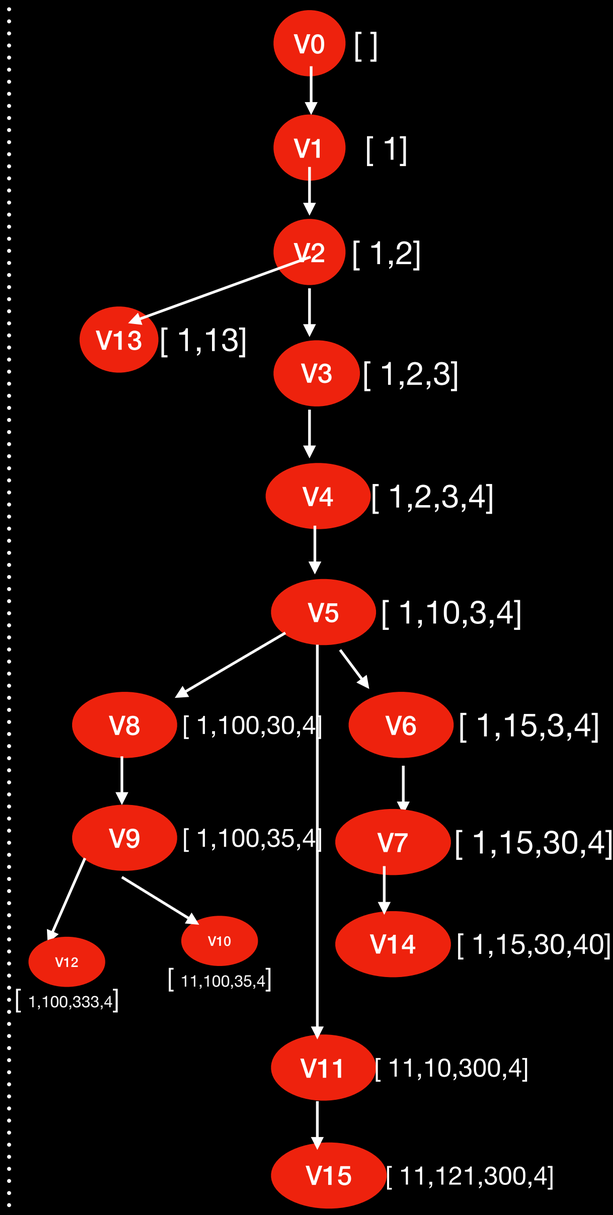
iterate_LL_at_v(v12)



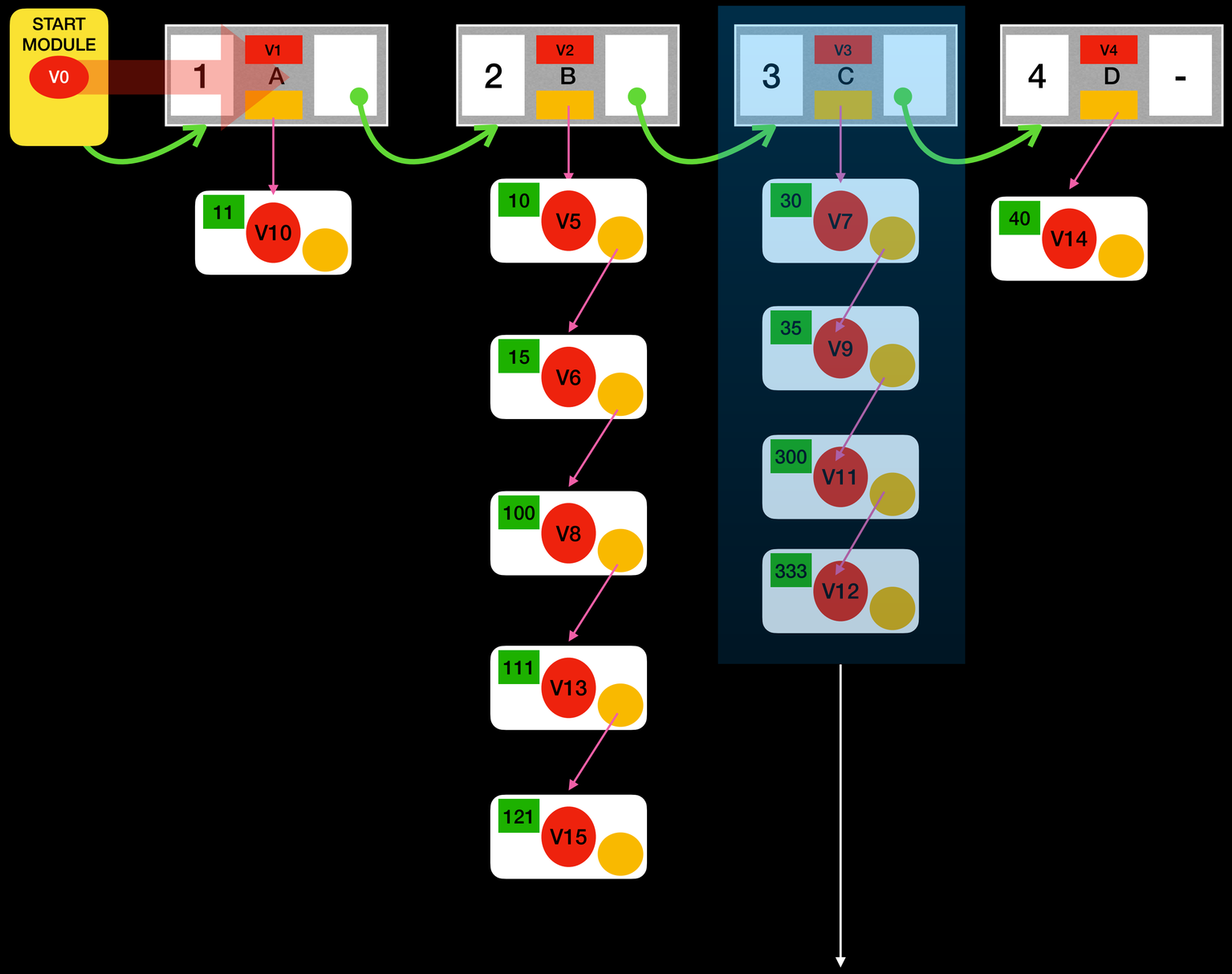
OUPTUT : 1 (v1) 100 (v8)

Current time, t = 15

Version tree



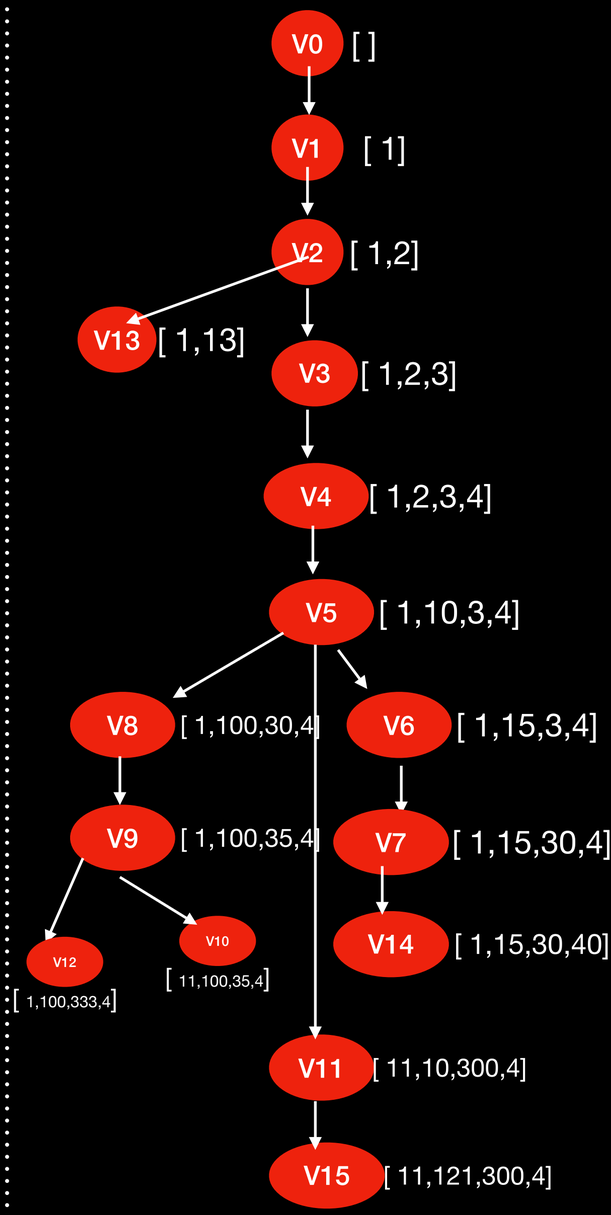
iterate_LL_at_v(v12)



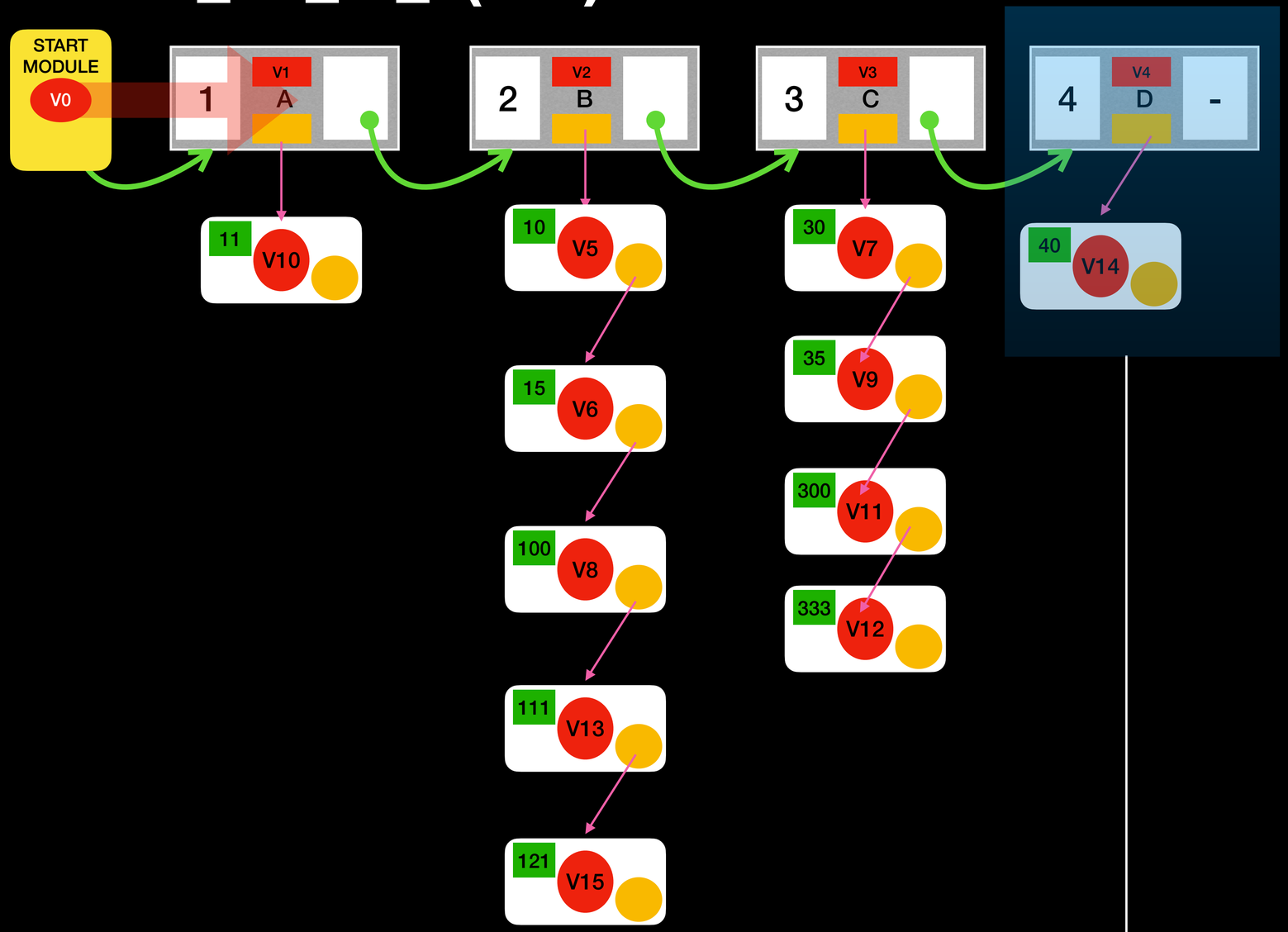
OUPTUT : 1 (v1) 100 (v8) 333(v12)

Current time, t = 15

Version tree



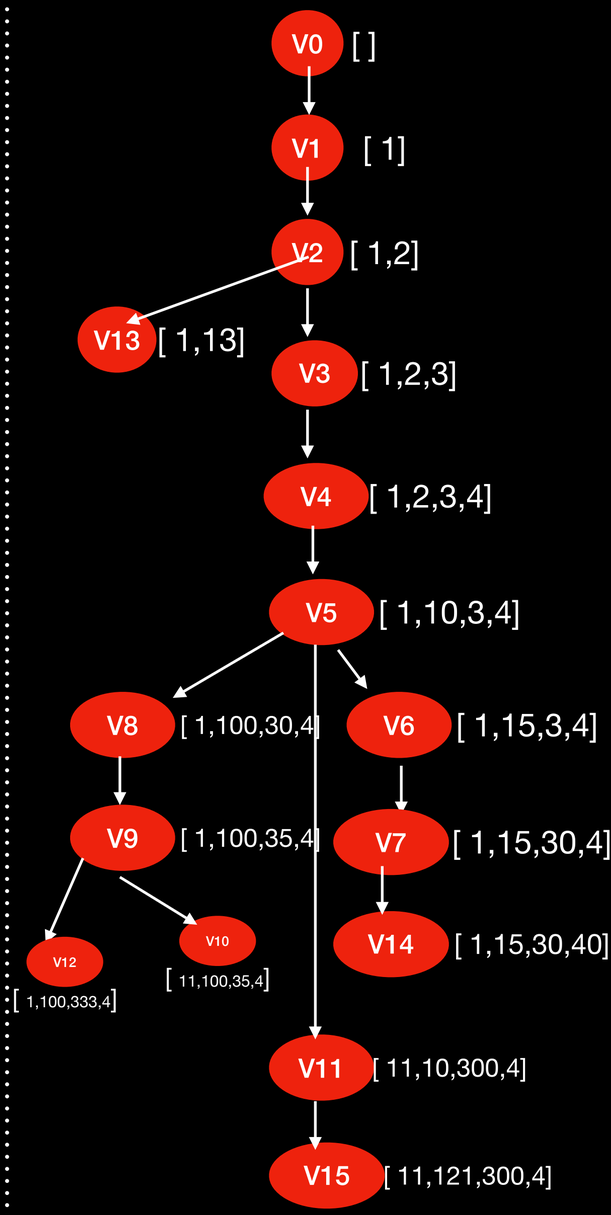
iterate_LL_at_v(v12)



OUPTUT : 1 (v1) 100 (v8) 333(v12) 4 (v4)

Current time, t = 15

Version tree

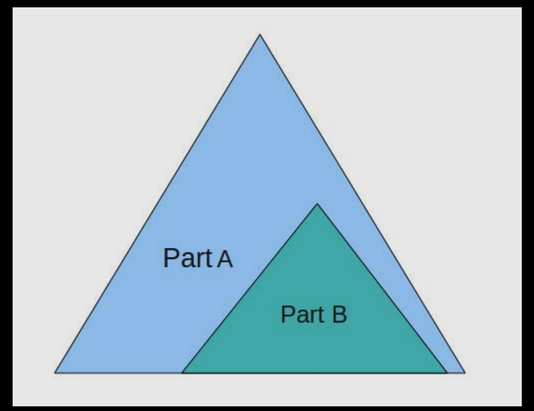


Implementation Using Pointer Machine

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?

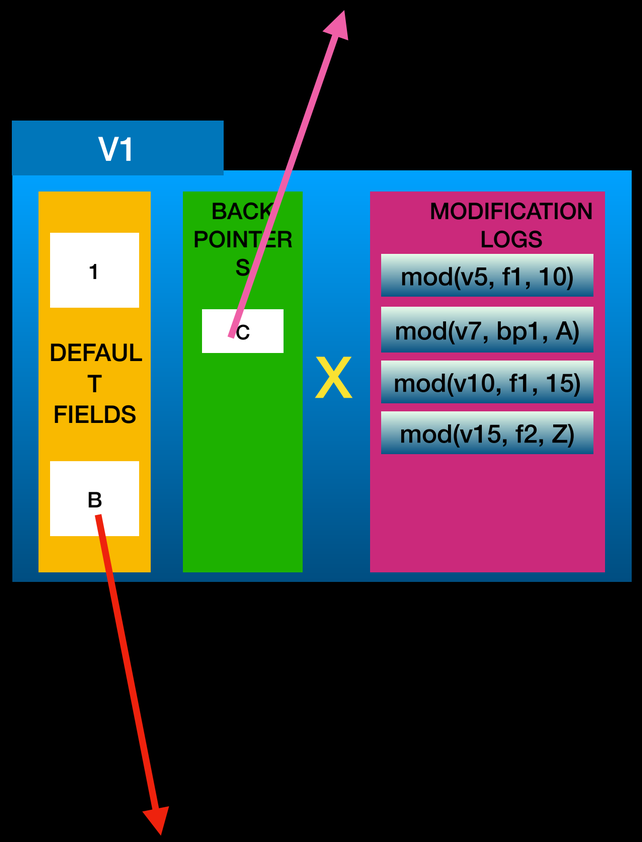
To reduce the pressure on a node of particular version.
As, we can modify a particular node of a specific version multiple number of time in Full Persistent Strategy.

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.

Visualisation is given at next page

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

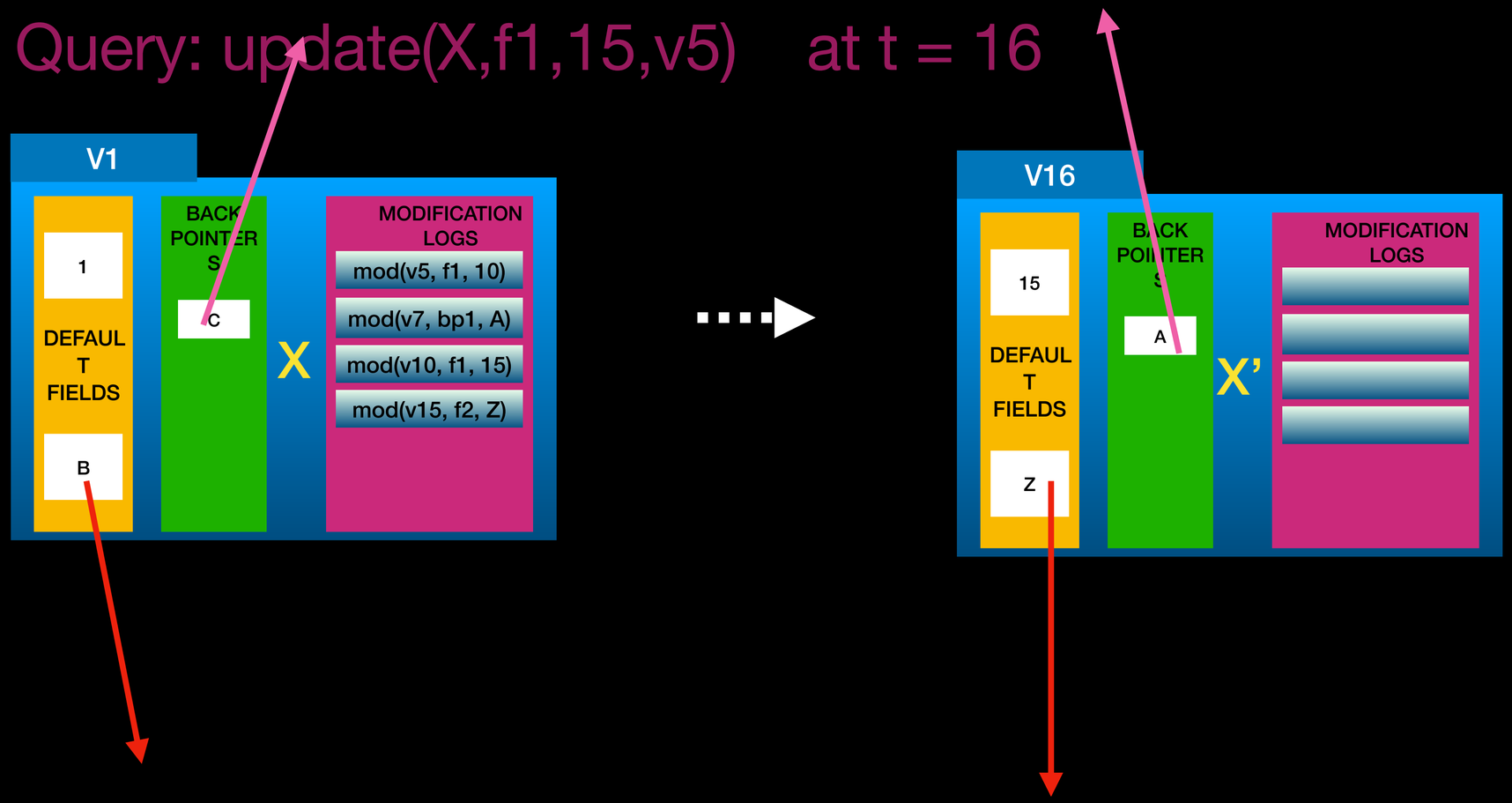


Suppose Node in any version (cur t >=15) looks like this

NOTE: THIS IS NOT DONE IN FULL PERSISTENT STRATEGY

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

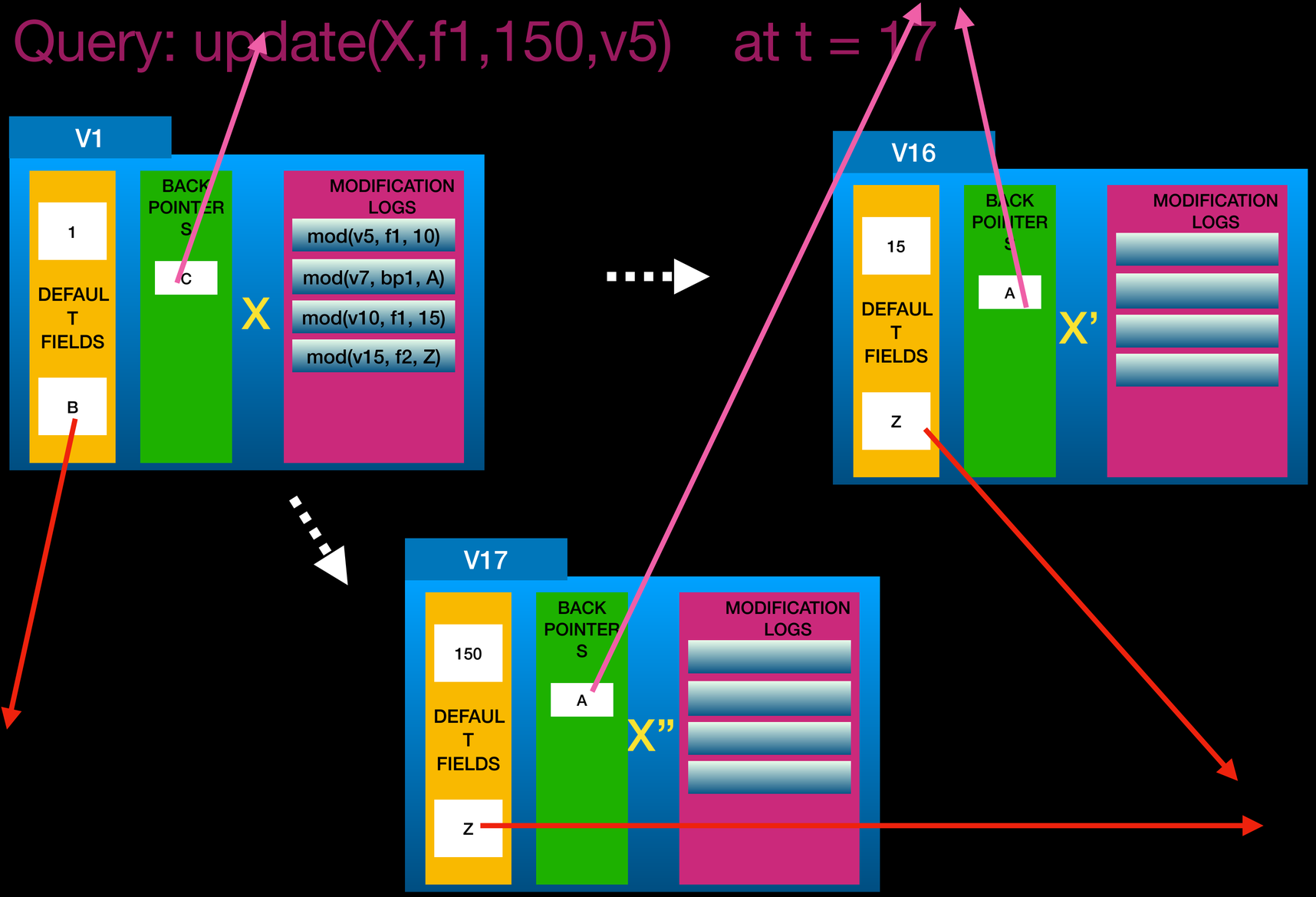
Query: update(X,f1,15,v5) at t = 16



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



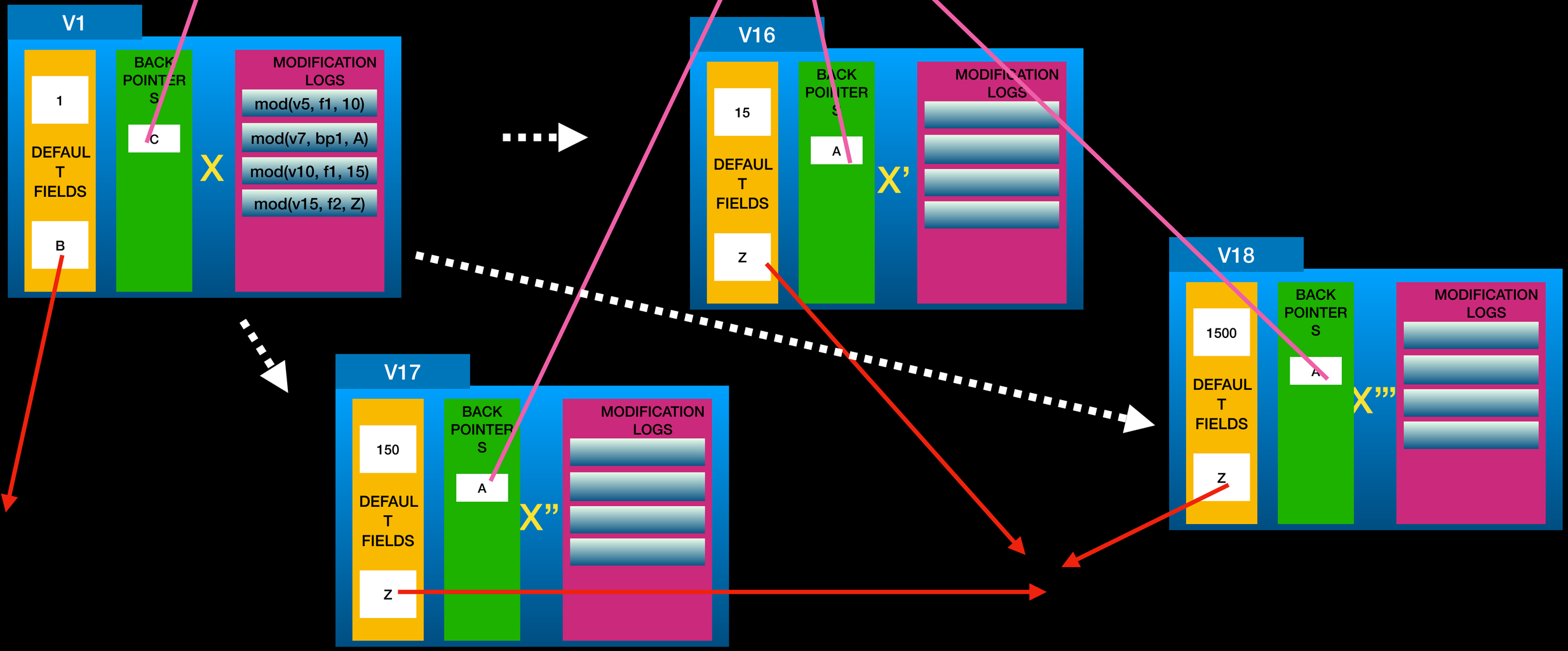
NOTE: THIS IS NOT DONE IN FULL PERSISTENT STRATEGY

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

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

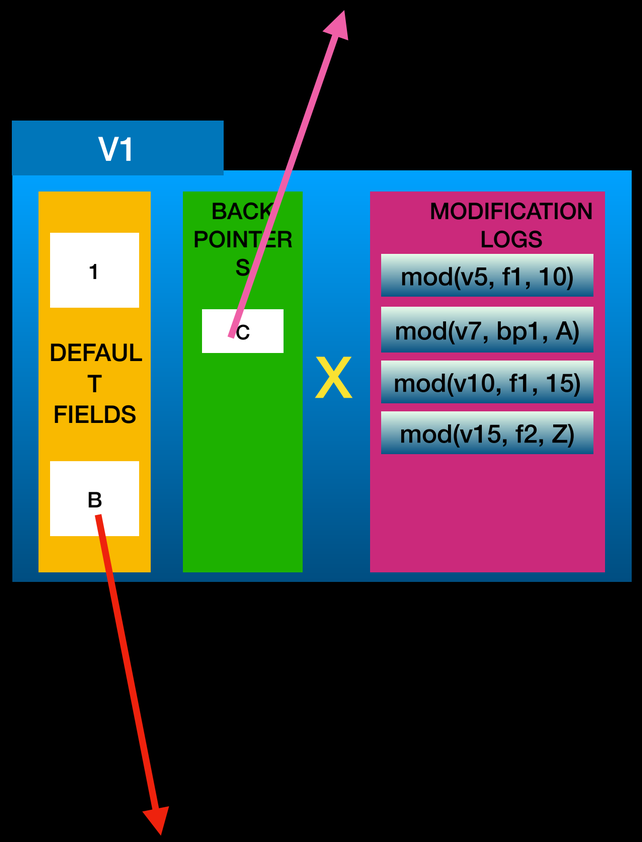
CONCERN:

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



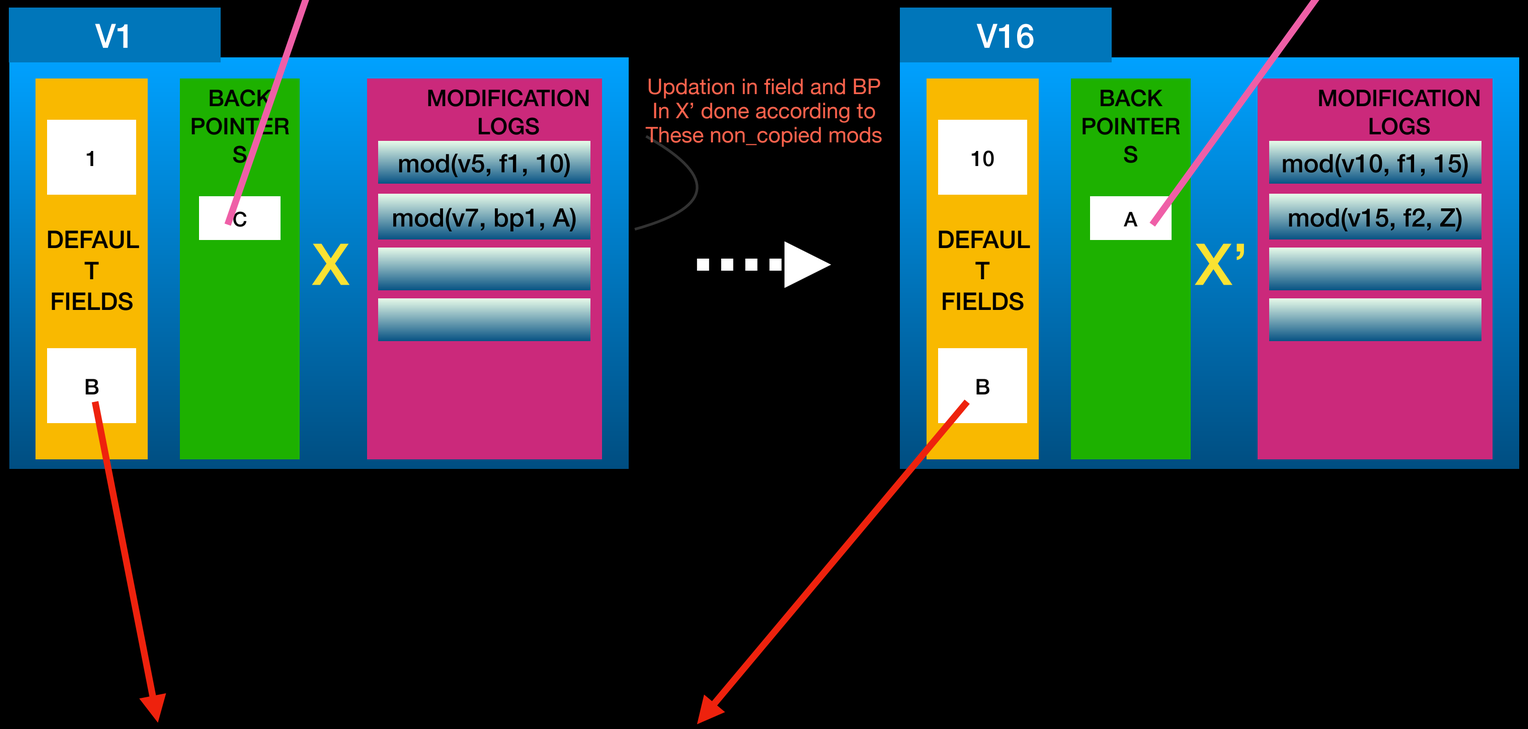
Suppose Node in any version (cur t >=15) looks like this

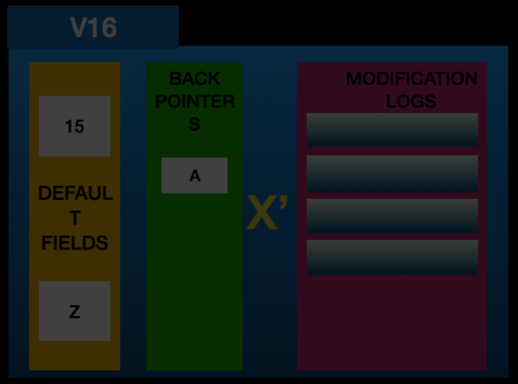
WE ARE GOING TO PRESENT 1:1 SPLIT

NOTE: THIS IS DONE IN FULL PERSISTENT STRATEGY

OPTIMISATION with 1:1 Approach

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



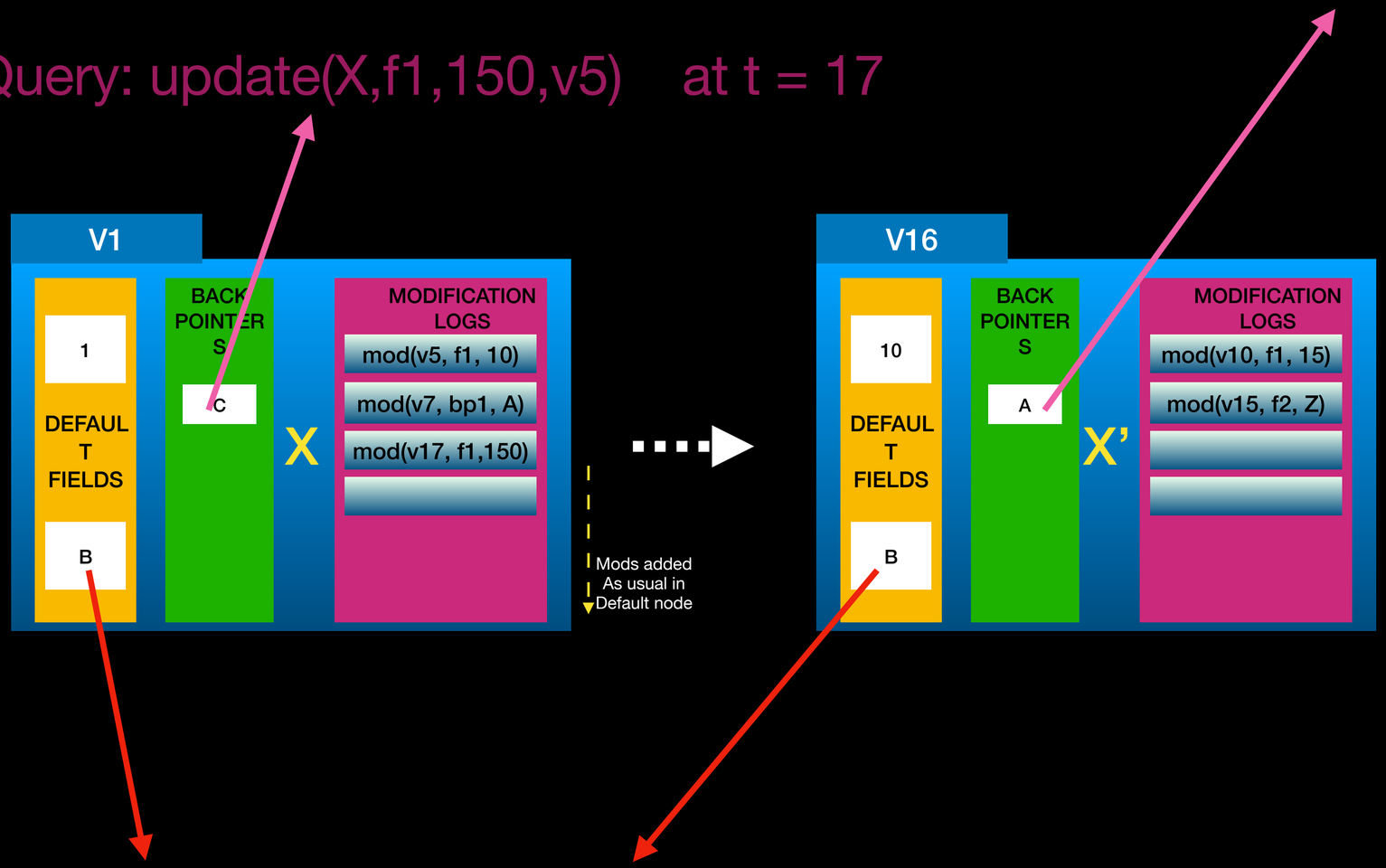


IN 1:0 MODE

NOTE: THIS IS DONE IN FULL PERSISTENT STRATEGY

OPTIMISATION with 1:1 Approach

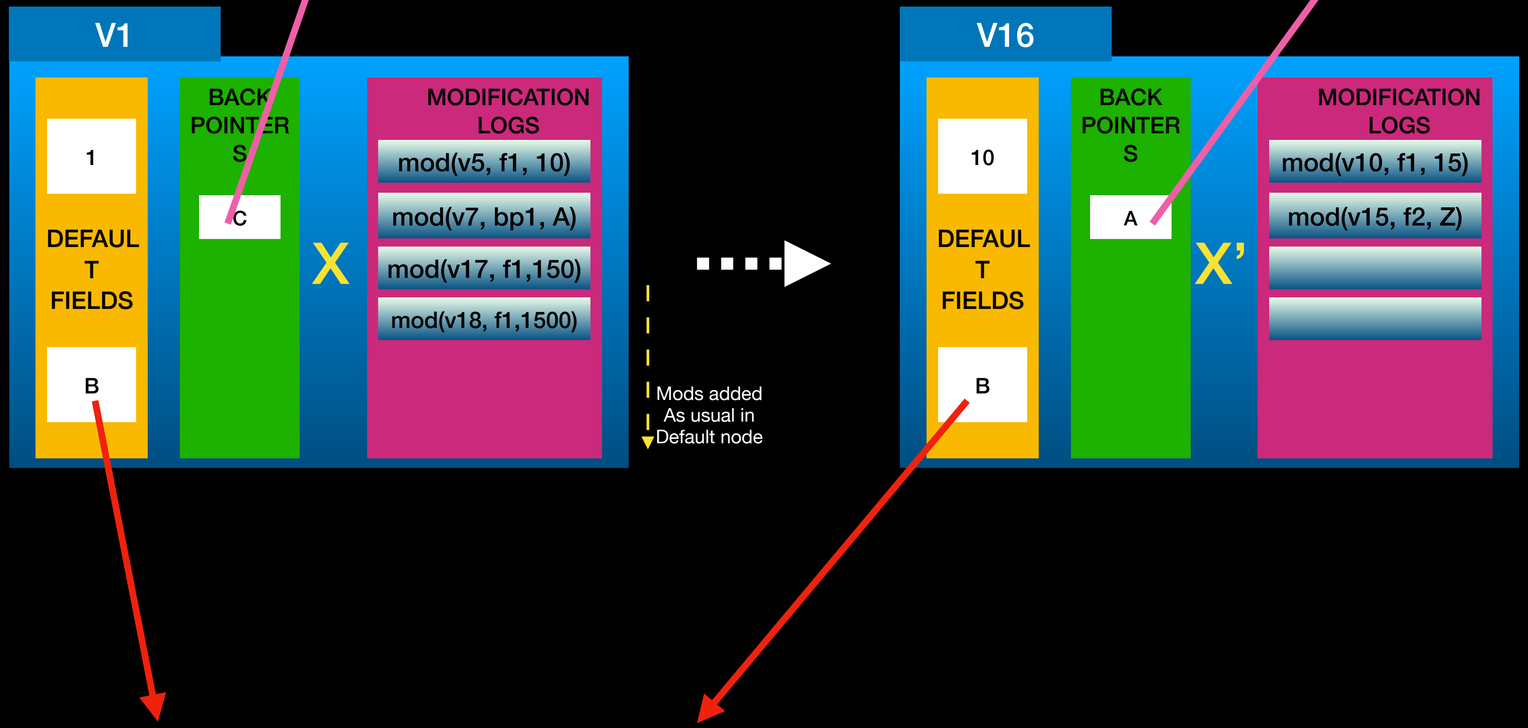
Query: update(X,f1,150,v5) at t = 17

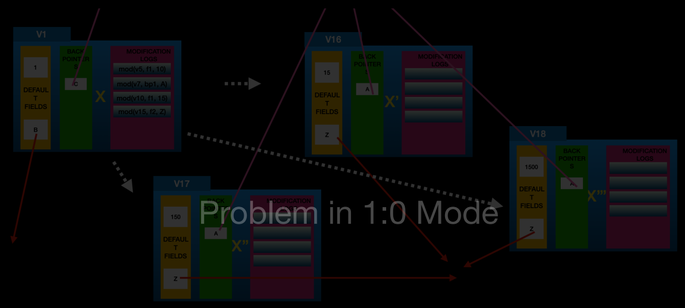


NOTE: THIS IS DONE IN FULL PERSISTENT STRATEGY

OPTIMISATION with 1:1 Approach

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





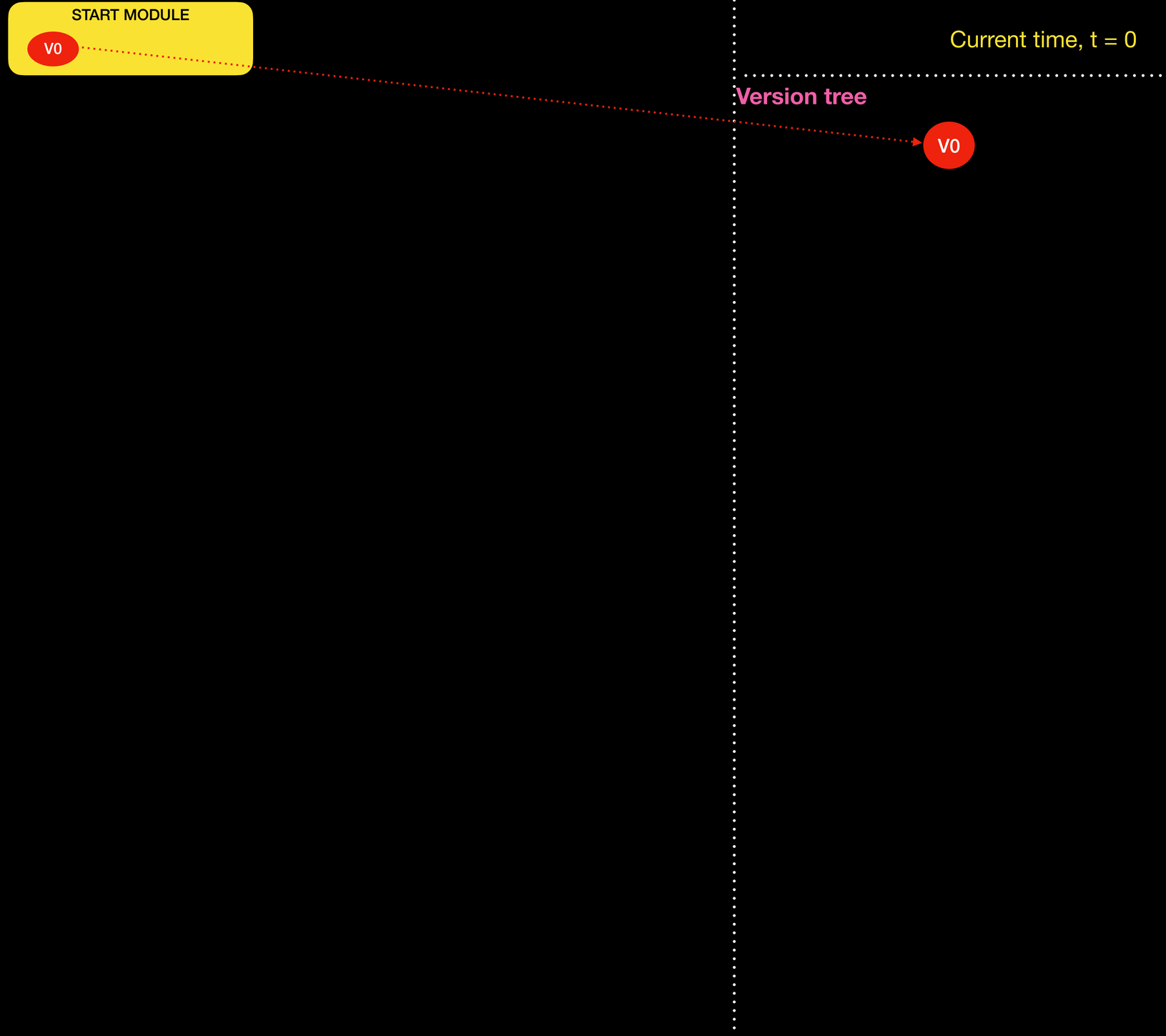
CONCERN:

HERE WE ARE NOT CREATING SUCCESSIVE EMPTY_MOD NODE-copies

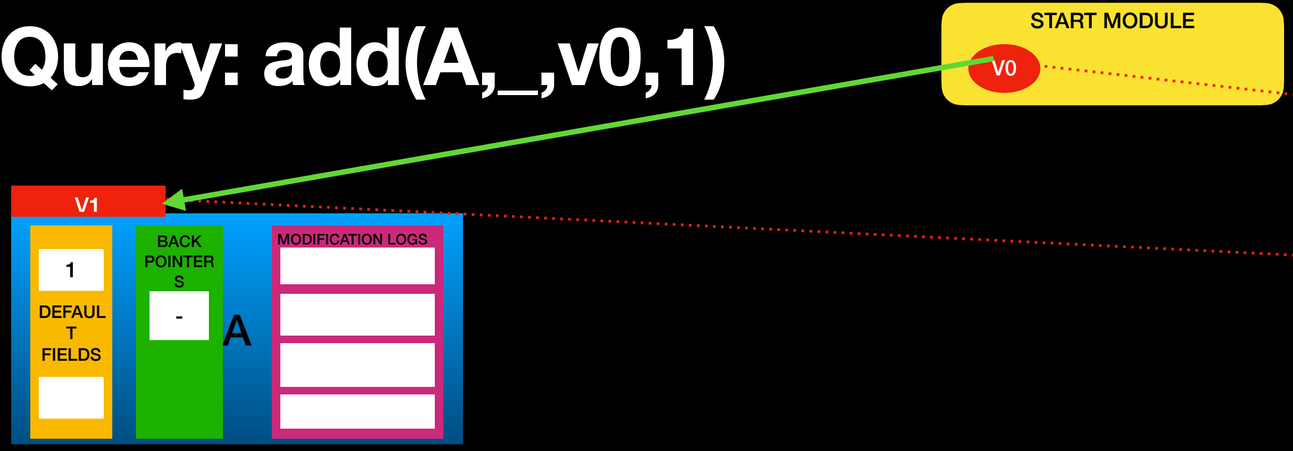
NOTE: THIS IS DONE IN FULL PERSISTENT STRATEGY

Simulation Using Full Persistent Model

Query: init_LL()

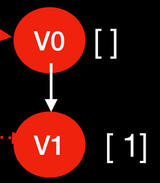


Query: add(A,_,v0,1)



Current time, t = 1

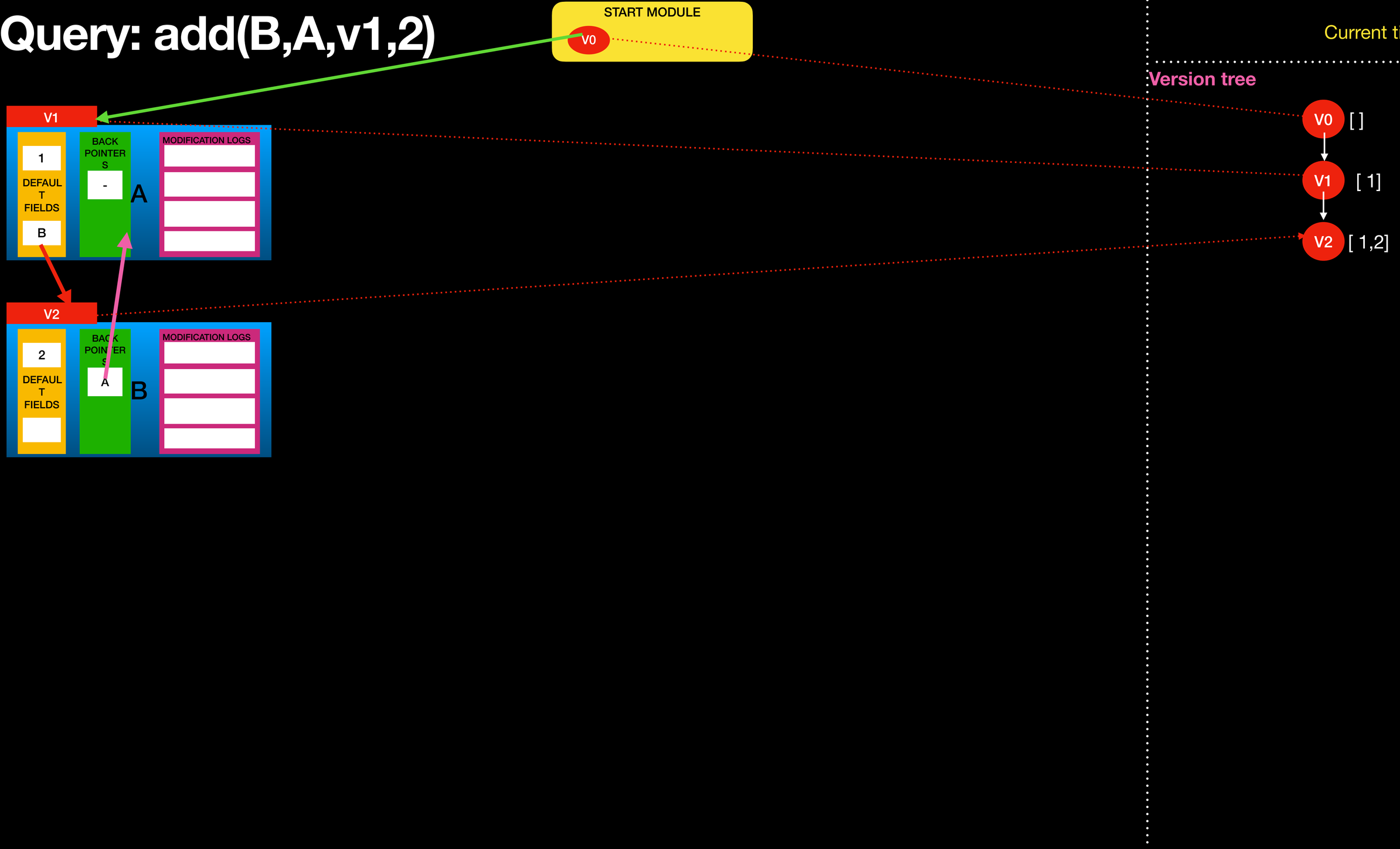
Version tree



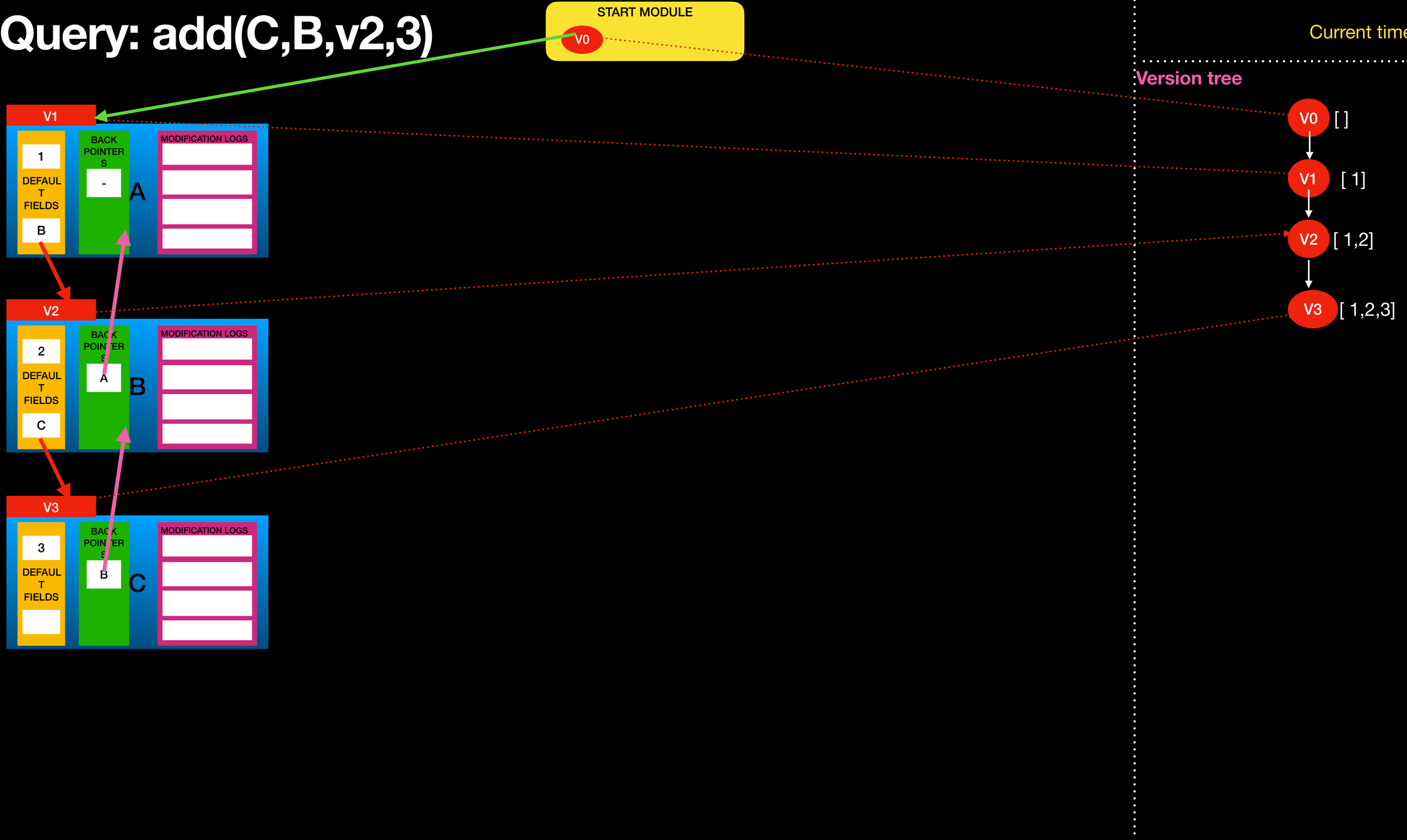
add(X,W,v90,1000) at time = 95 Includes

Create new node X with f1 = 1000, defaultVersion = v95
update(W,f2,X,v90)
update(X, bp, W, v90) [here, update means set at default level]
Add v95 under v90 in version tree

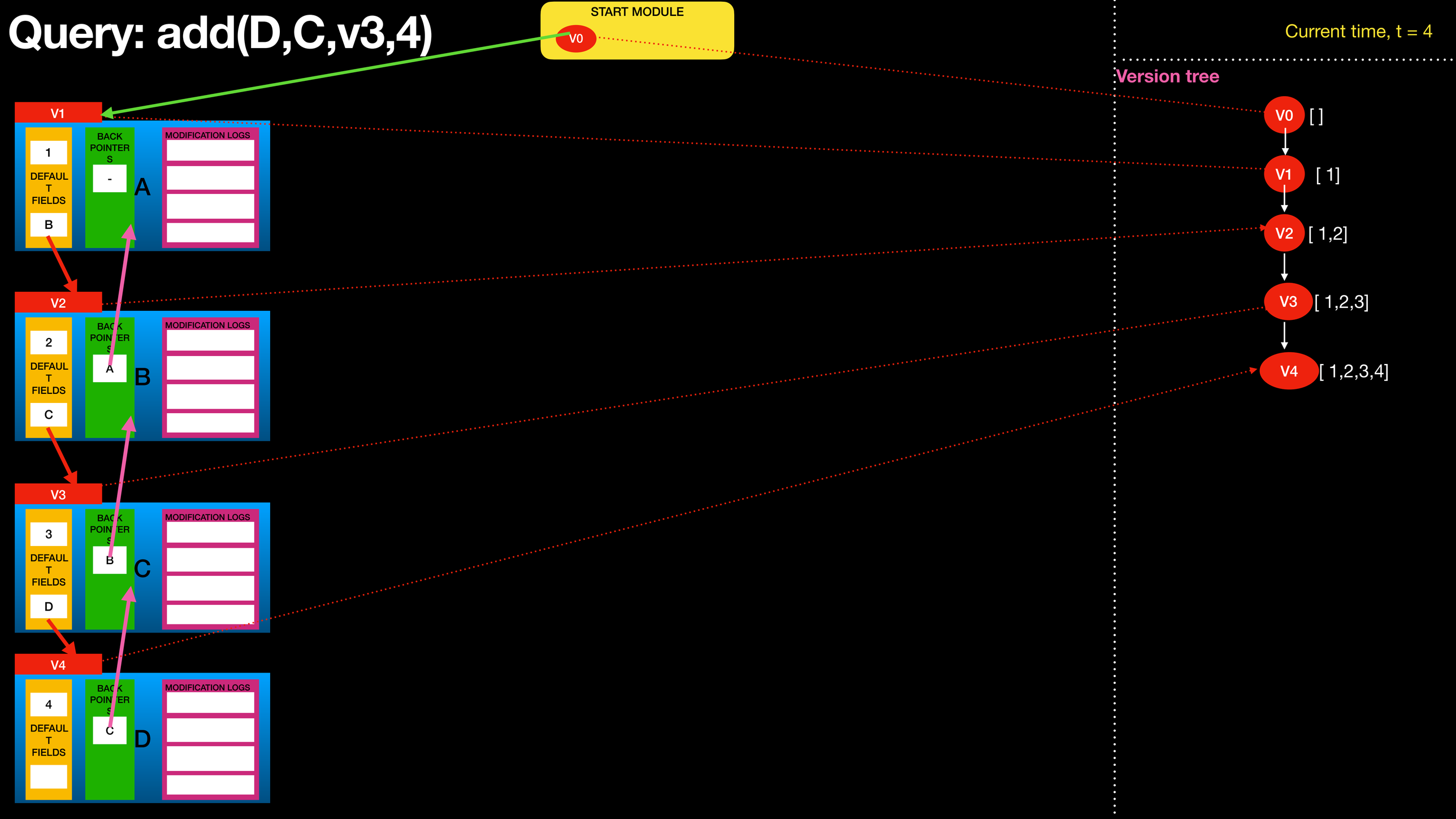
Query: add(B,A,v1,2)



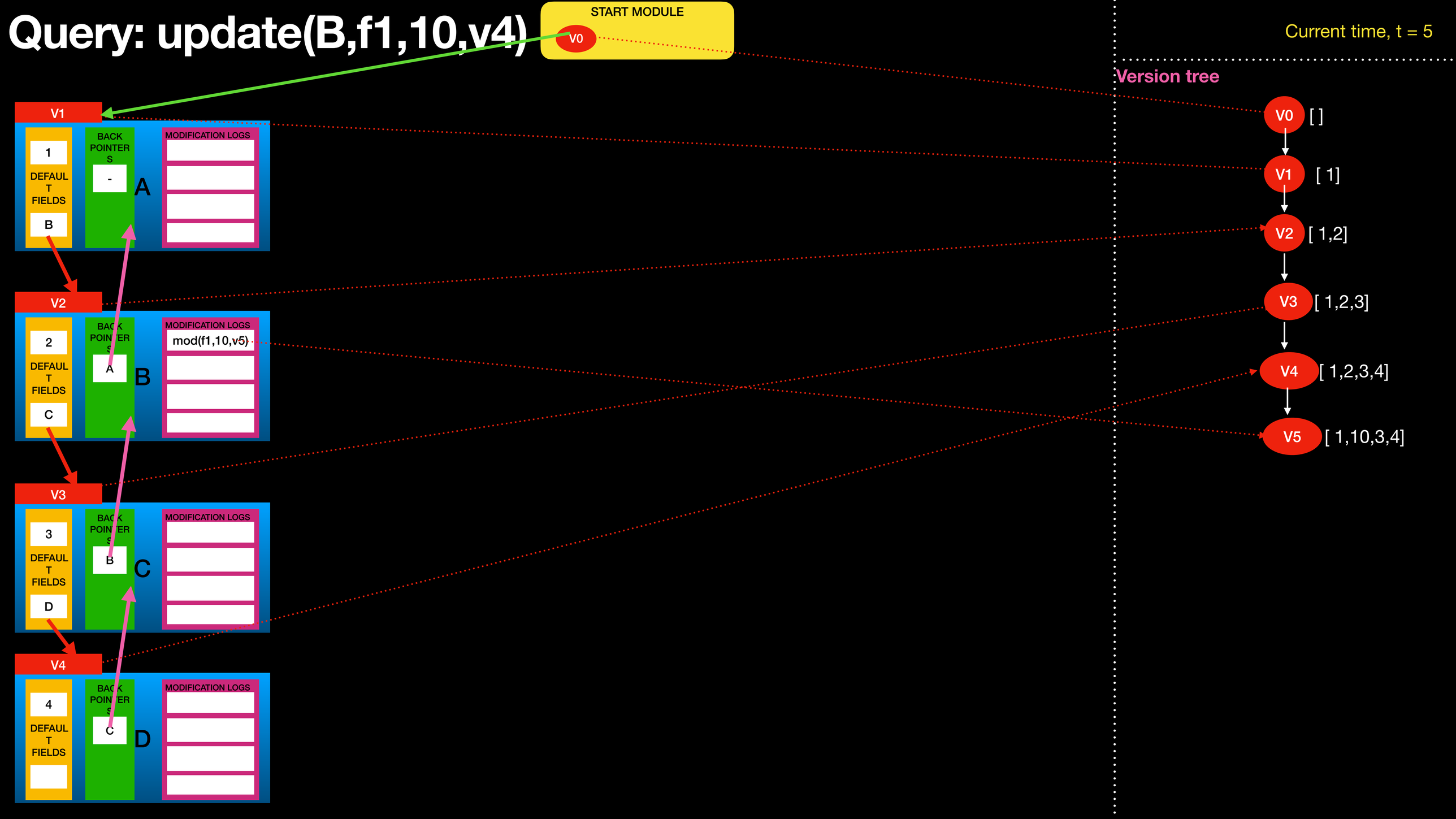
Query: add(C,B,v2,3)



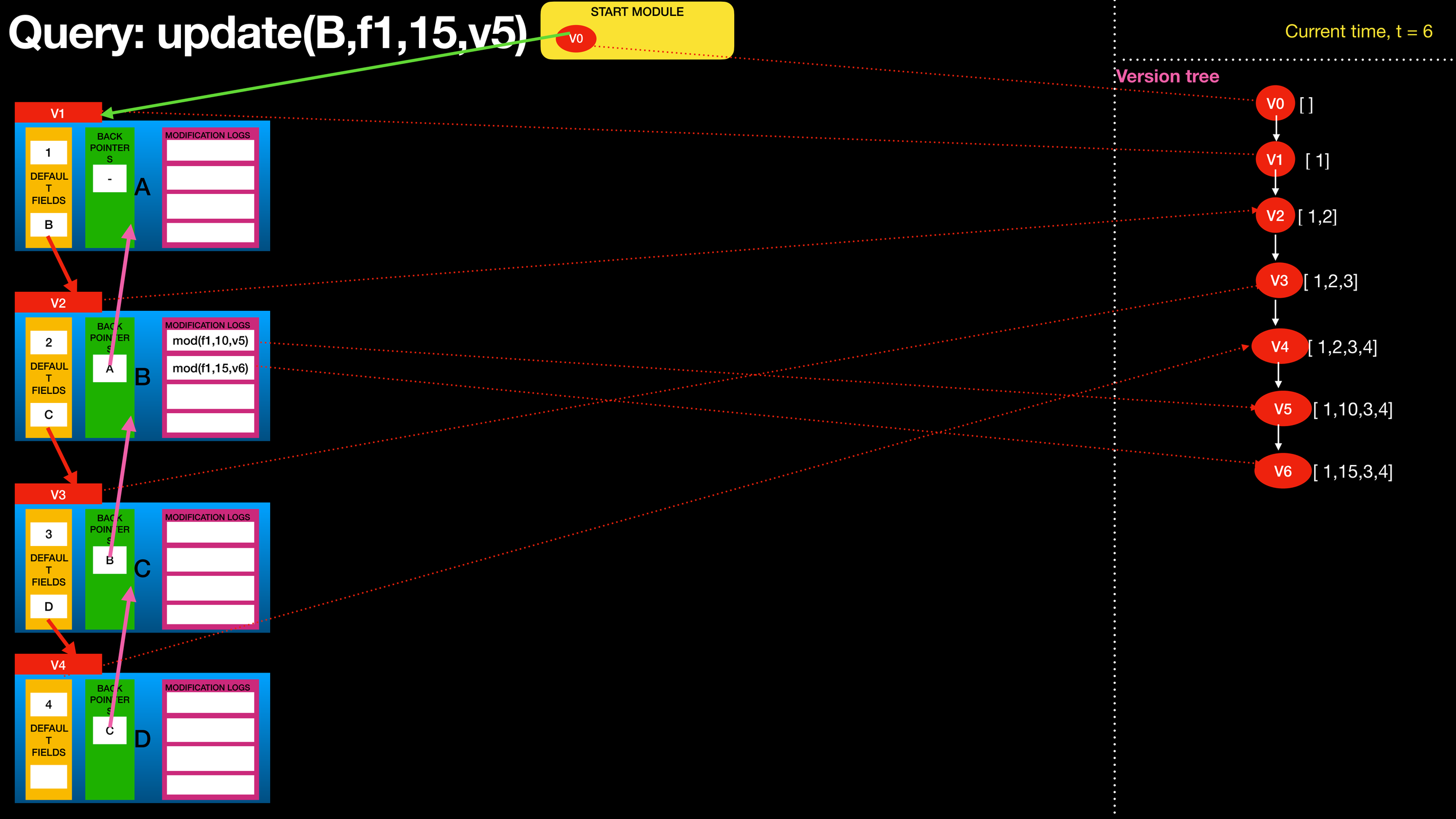
Query: add(D,C,v3,4)



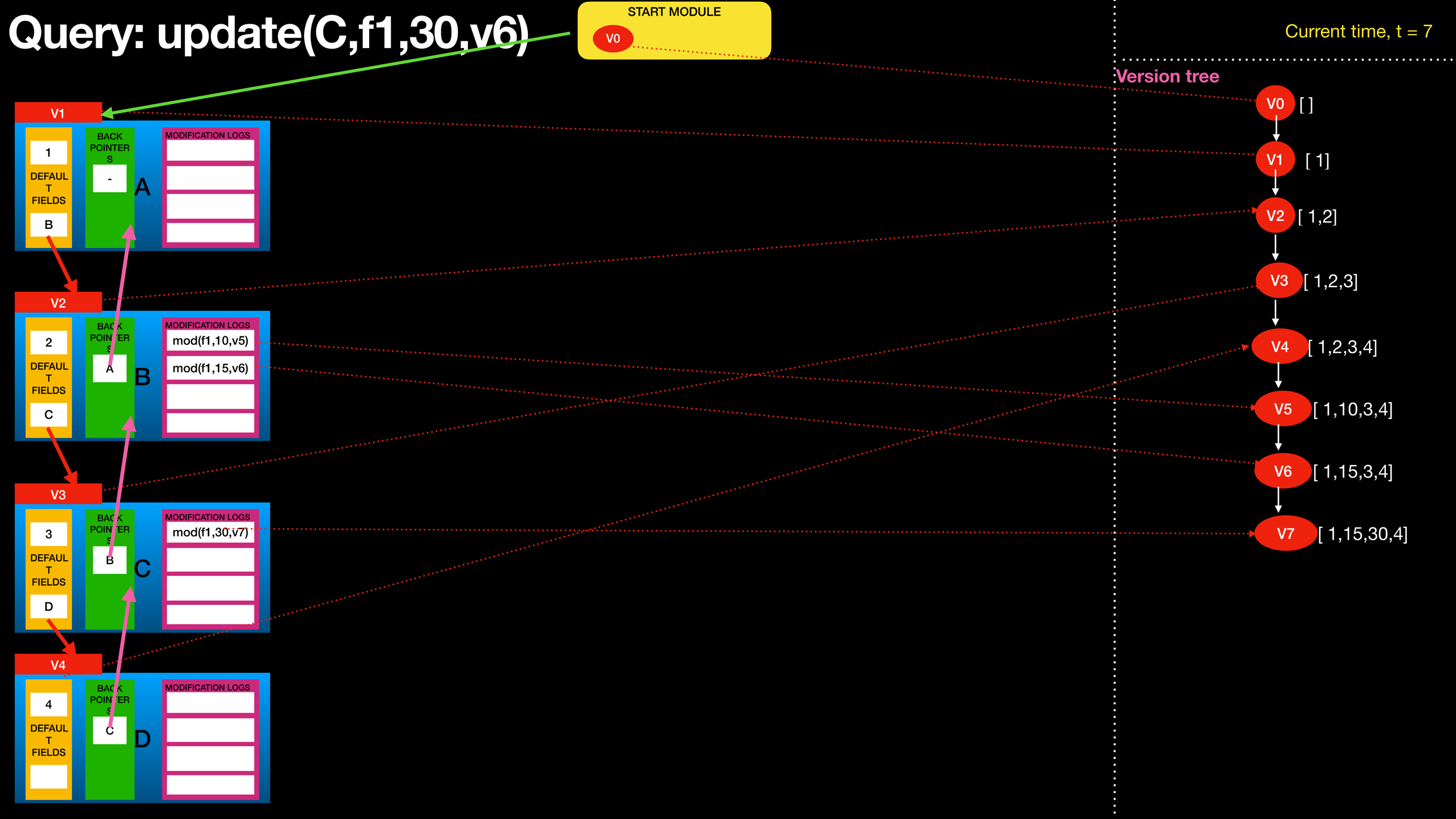
Query: update(B,f1,10,v4)



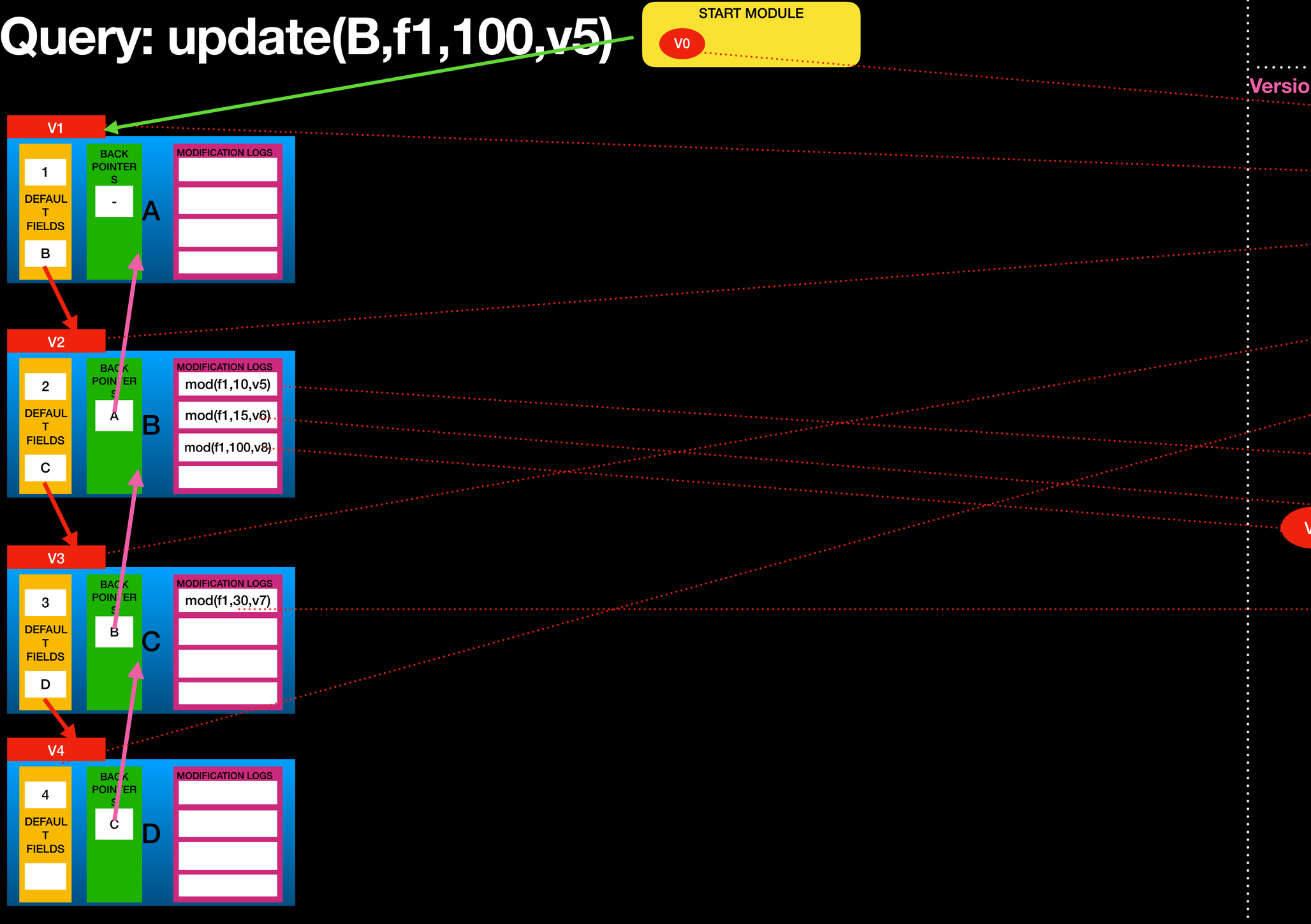
Query: update(B,f1,15,v5)



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

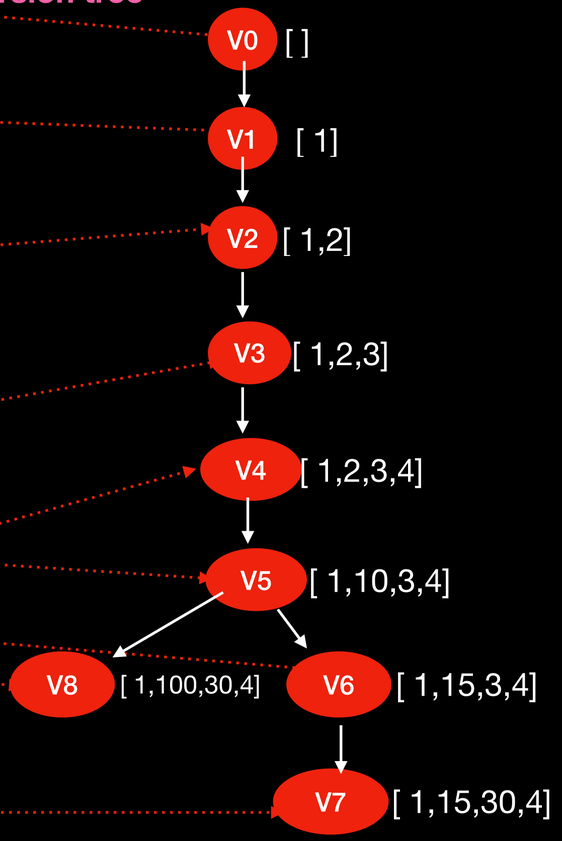


Query: update(B,f1,100,v5)

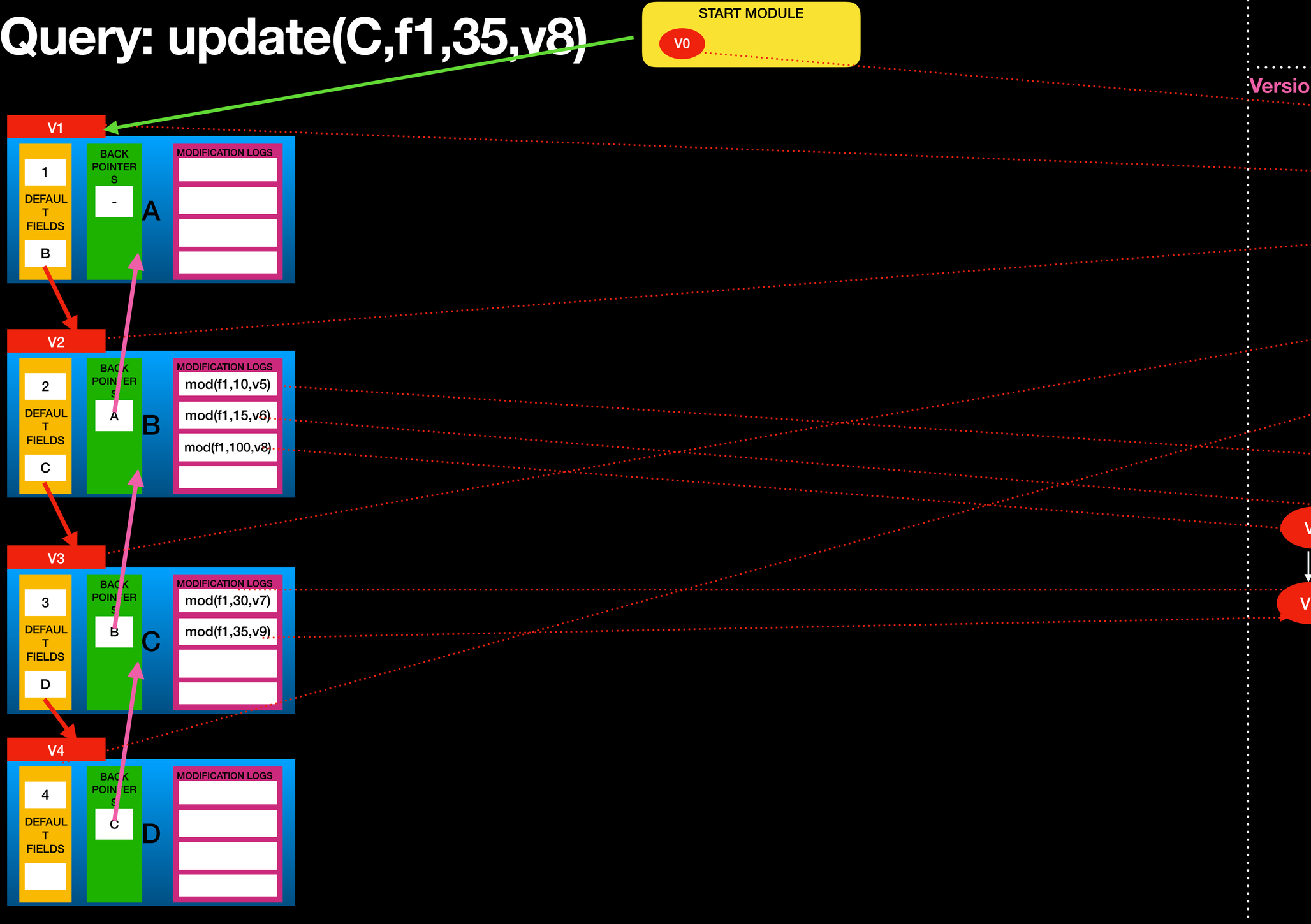


Current time, t = 8

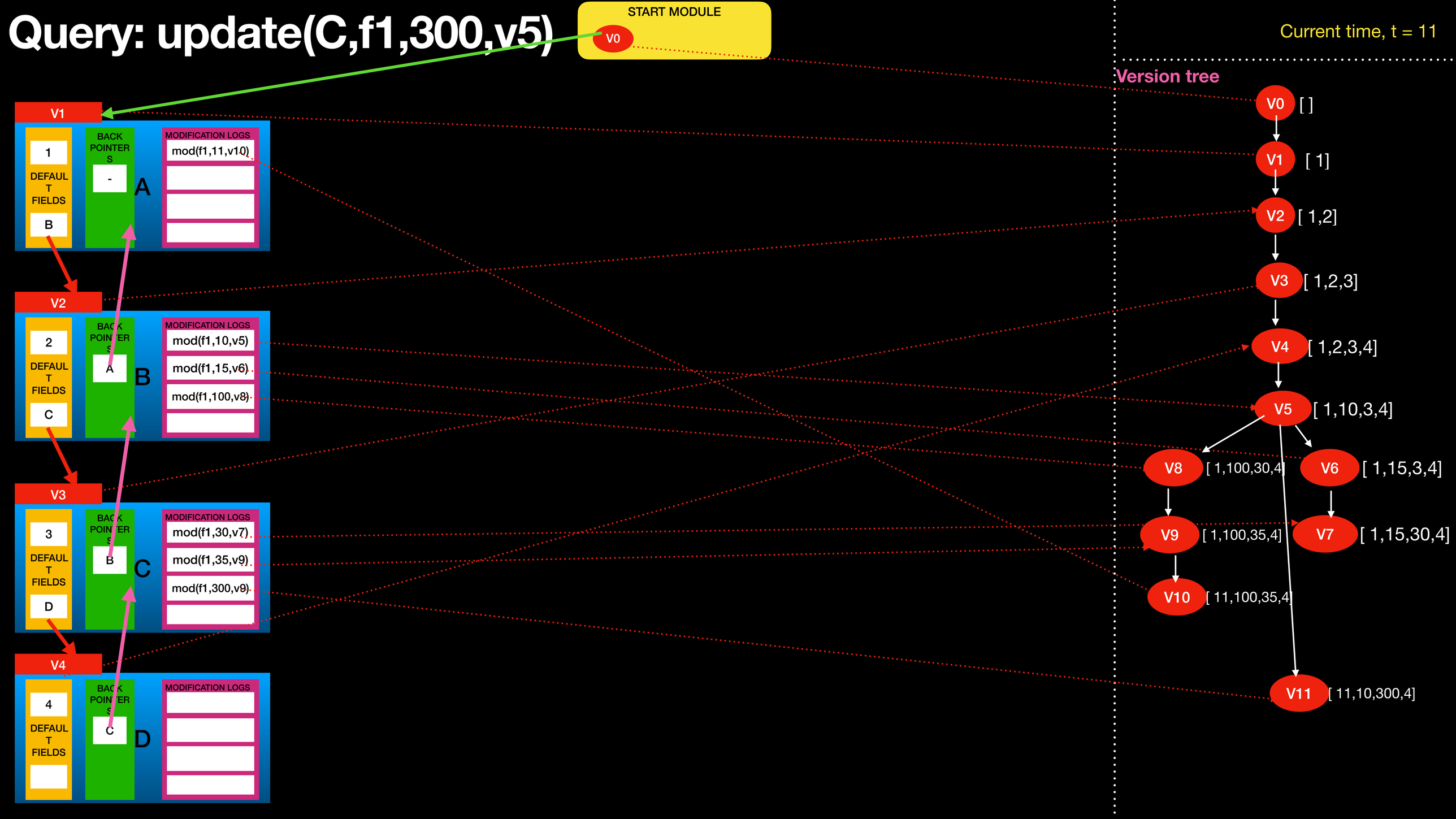
Version tree



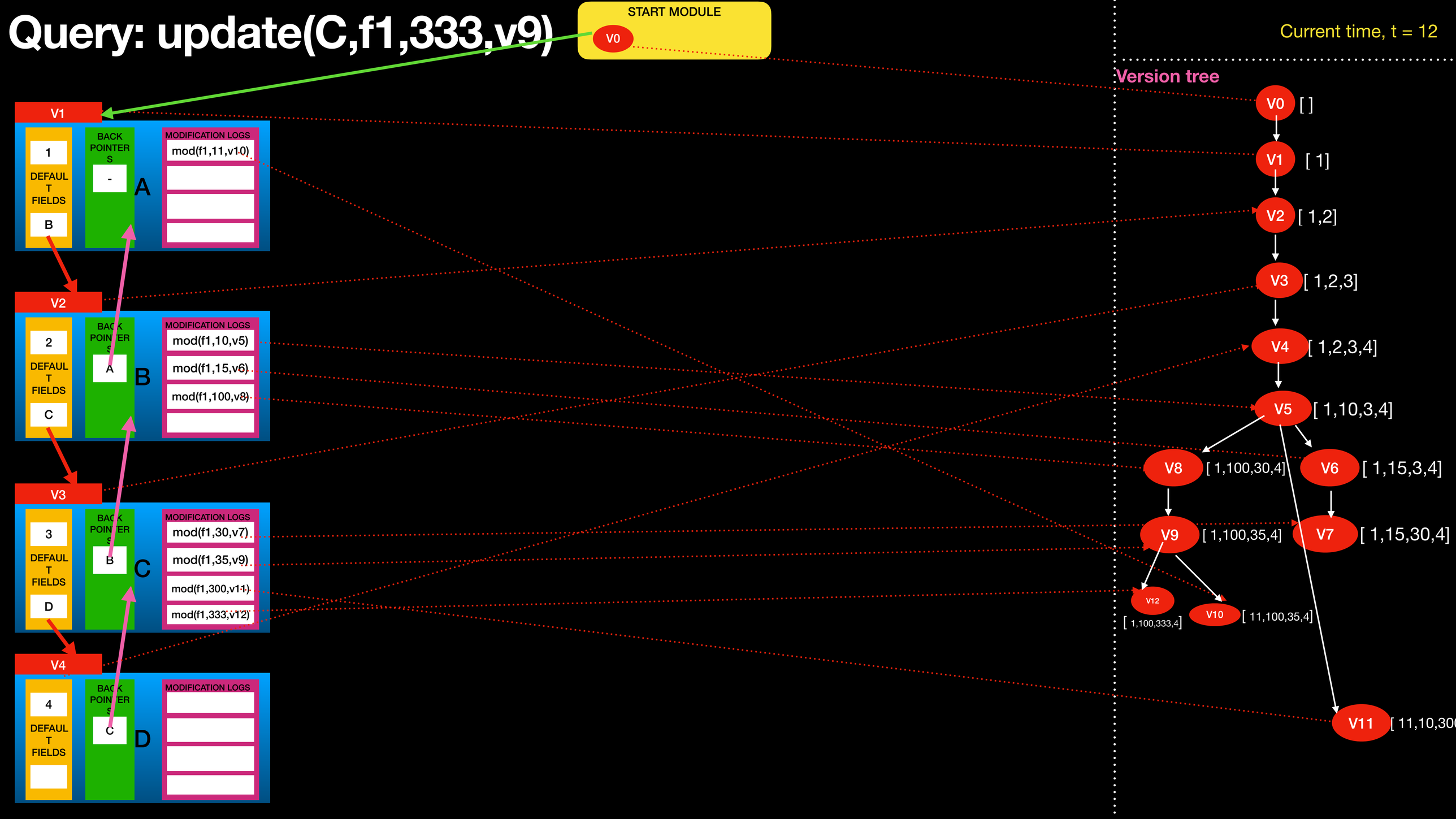
Query: update(C,f1,35,v8)



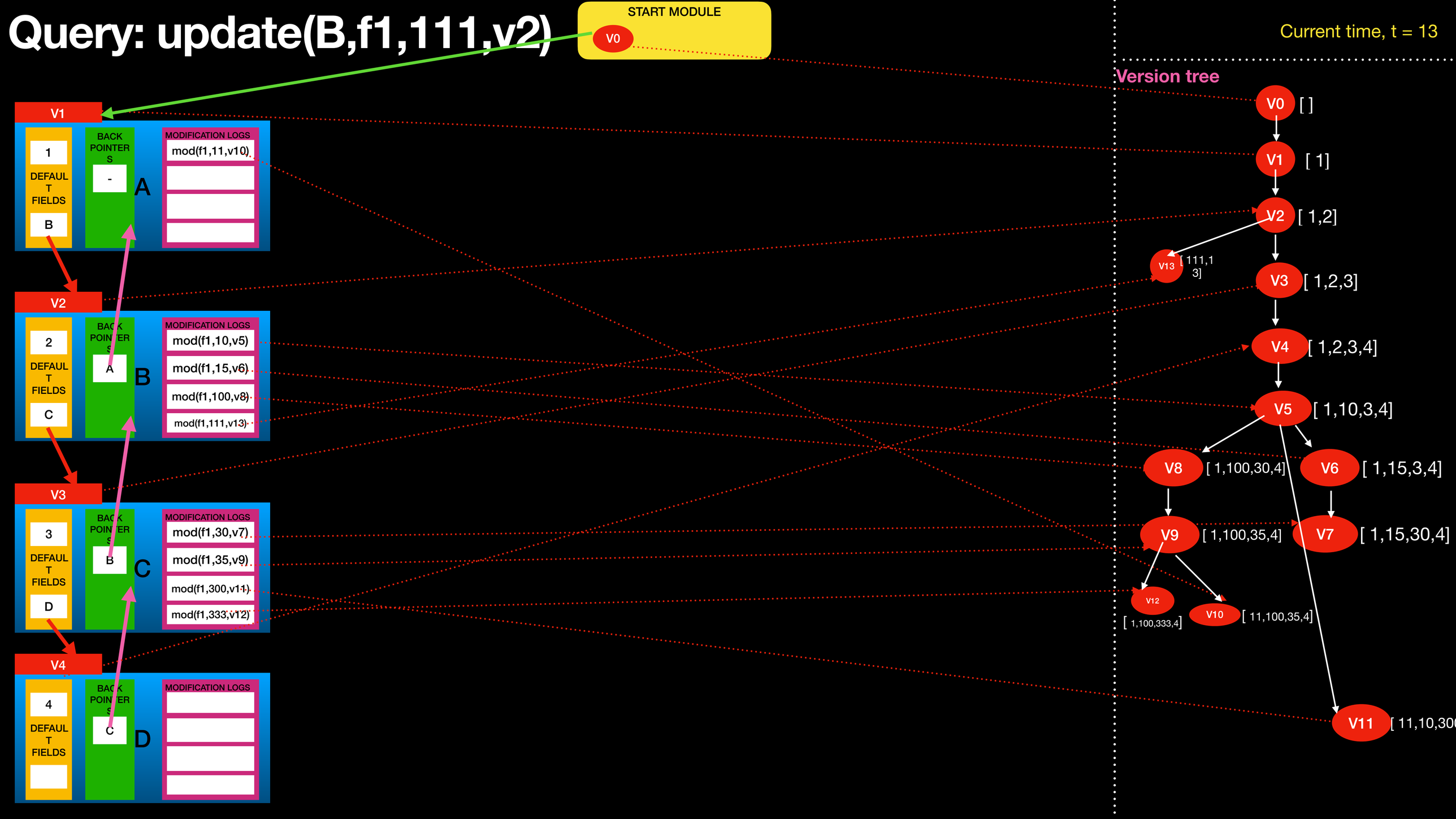
Query: update(C,f1,300,v5)



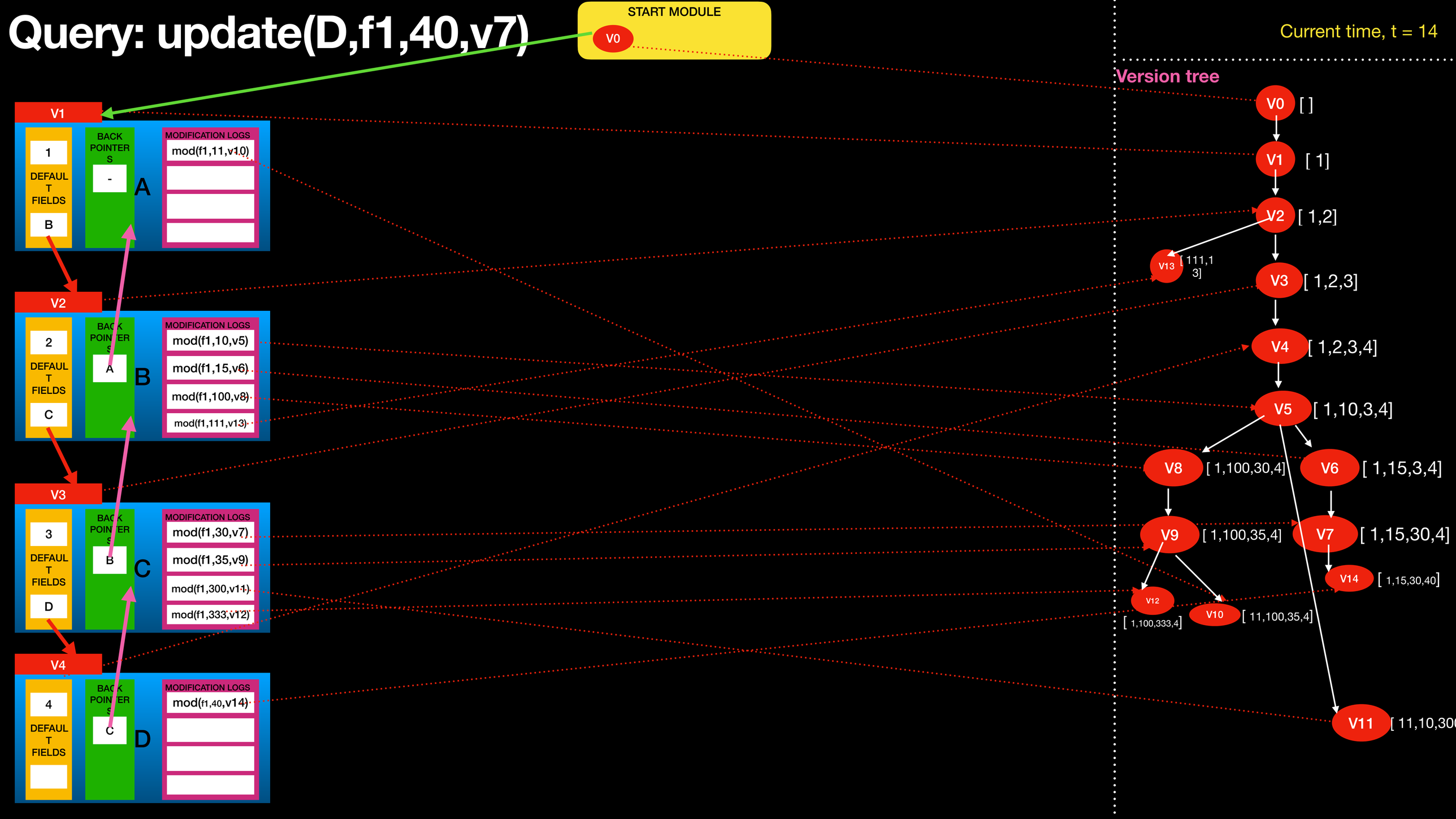
Query: update(C,f1,333,v9)



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



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

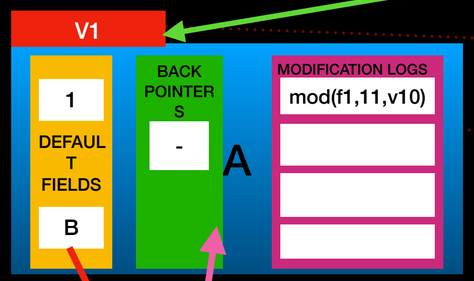


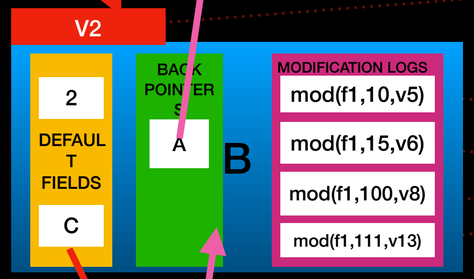
Query: update(C,f1,321,v11)

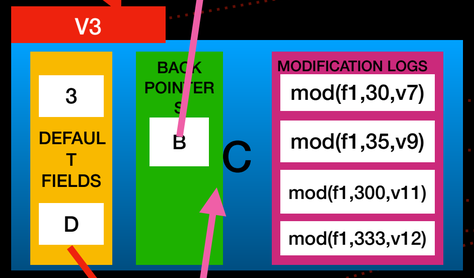
START MODULE

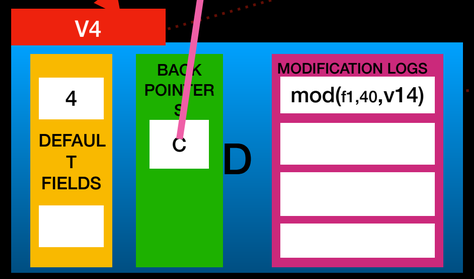
V0

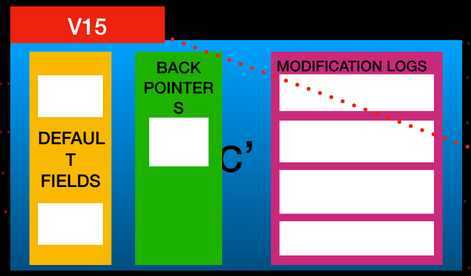
Current time, t = 15









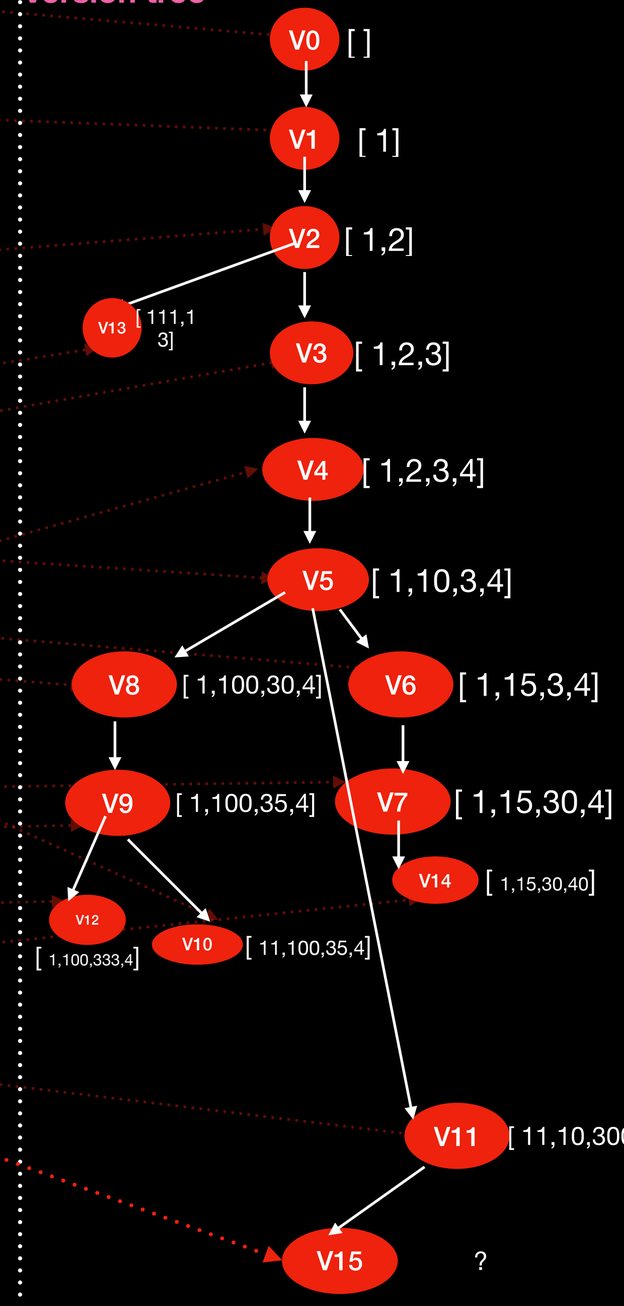


STEP 1

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

PTO->

Version tree



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

START MODULE

V0

V1

1

DEFAULT FIELDS

B

BACK POINTERS

-

A

MODIFICATION LOGS

mod(f1,11,v10)

V2

2

DEFAULT FIELDS

C

BACK POINTERS

A

B

MODIFICATION LOGS

mod(f1,10,v5)

mod(f1,15,v6)

mod(f1,100,v8)

mod(f1,111,v13)

V3

3

DEFAULT FIELDS

D

BACK POINTERS

B

C

MODIFICATION LOGS

mod(f1,30,v7)

mod(f1,35,v9)

mod(f1,300,v11)

mod(f1,333,v12)

V4

4

DEFAULT FIELDS

BACK POINTERS

C

D

MODIFICATION LOGS

mod(f1,40,v14)

DEFAULT FIELDS

BACK POINTERS

C'

MODIFICATION LOGS

STEP 2

*TRANSFER **AROUND 50%** MODS TO C' FROM C*

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

PTO->

Current time, t = 15

Version tree

V0 [ ]

V1 [1]

V2 [1,2]

V13 [111,13]

V3 [1,2,3]

V4 [1,2,3,4]

V5 [1,10,3,4]

V8 [1,100,30,4]

V6 [1,15,3,4]

V9 [1,100,35,4]

V7 [1,15,30,4]

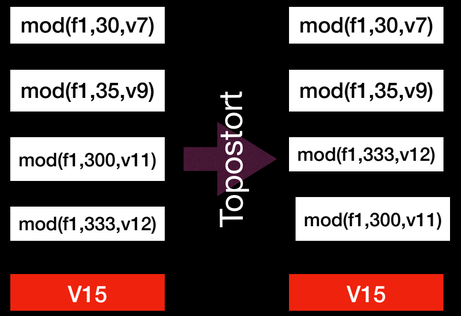
V14 [1,15,30,40]

V12 [1,100,333,4]

V10 [11,100,35,4]

V11 [11,10,30

V15 ?



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

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

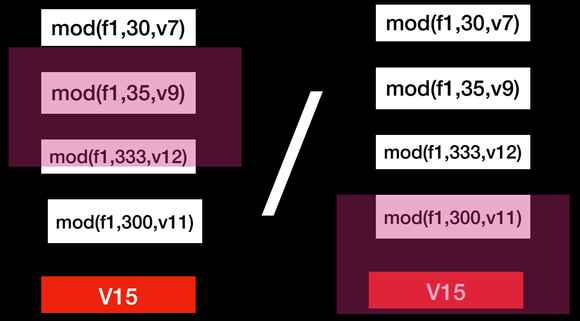
Open problem!!
I am still thinking for an optimisation
Here. Any one help!!

Who will go to New Copy??

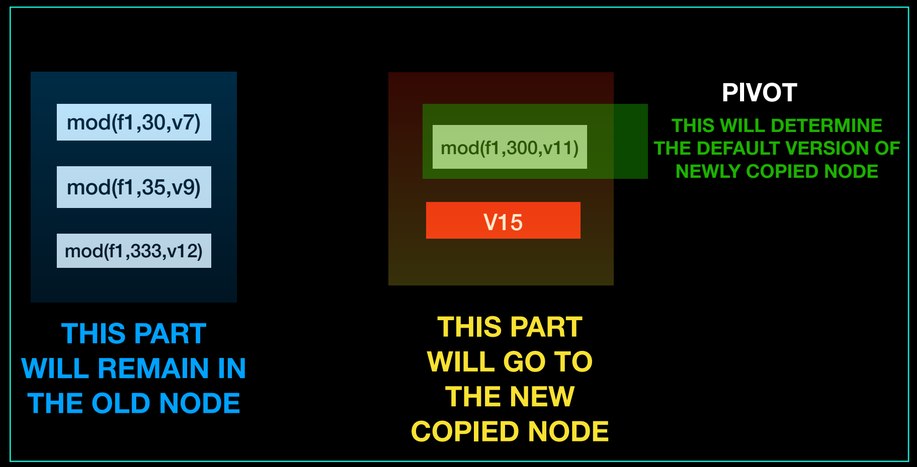
Choose
A Candidate Mod as pivot

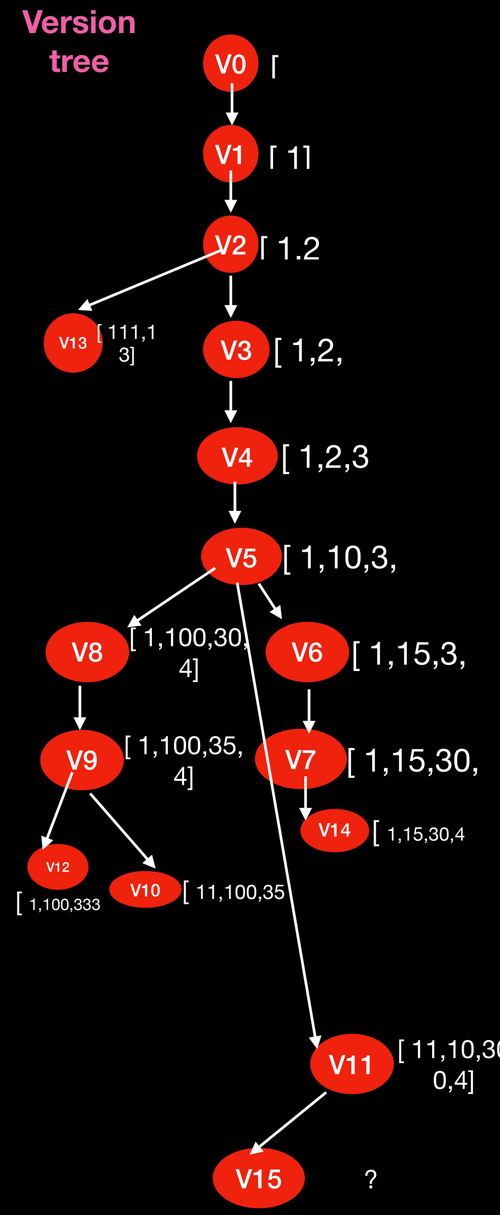
Which has most number of mods who are strictly successors of that pivot
In Topological Order

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



Suppose Option 2



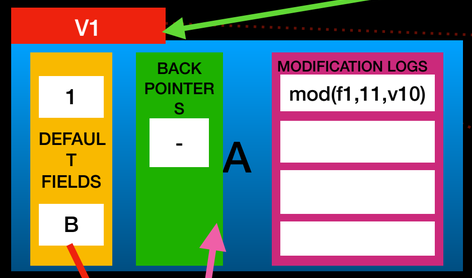


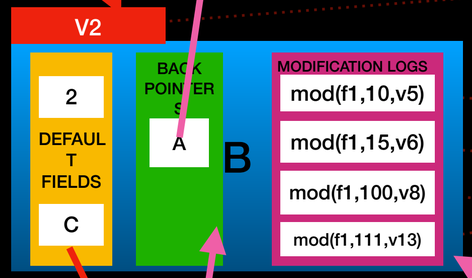
STEP 2, INTERMEDIATE DISCUSSION

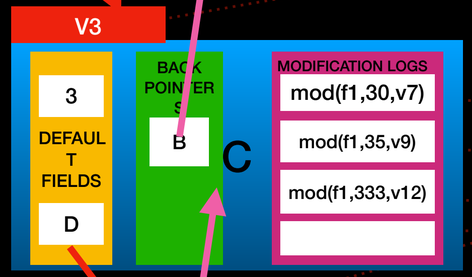
Query: update(C,f1,321,v11)

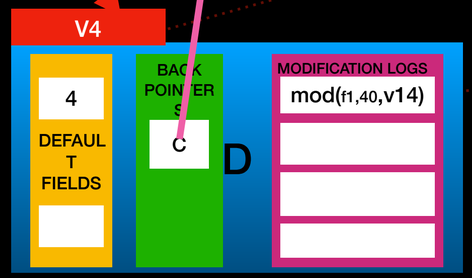
START MODULE

V0

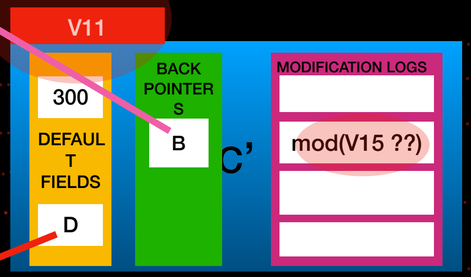


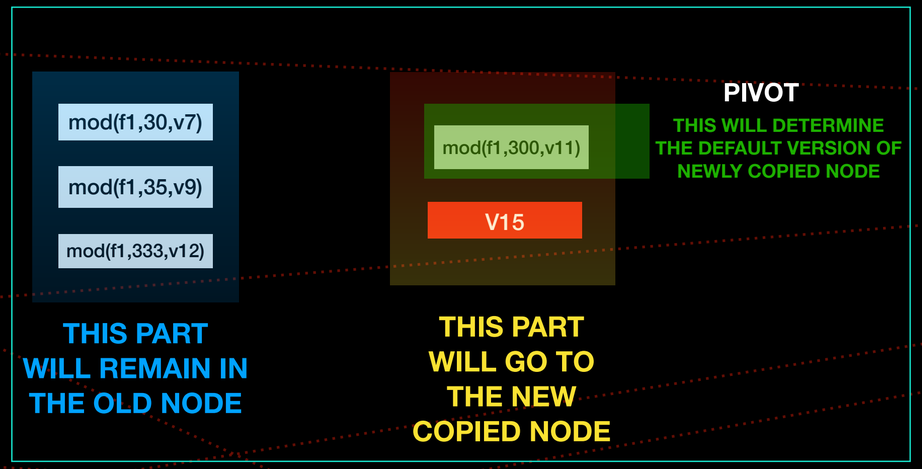






NOTE: IT IS NOT V15





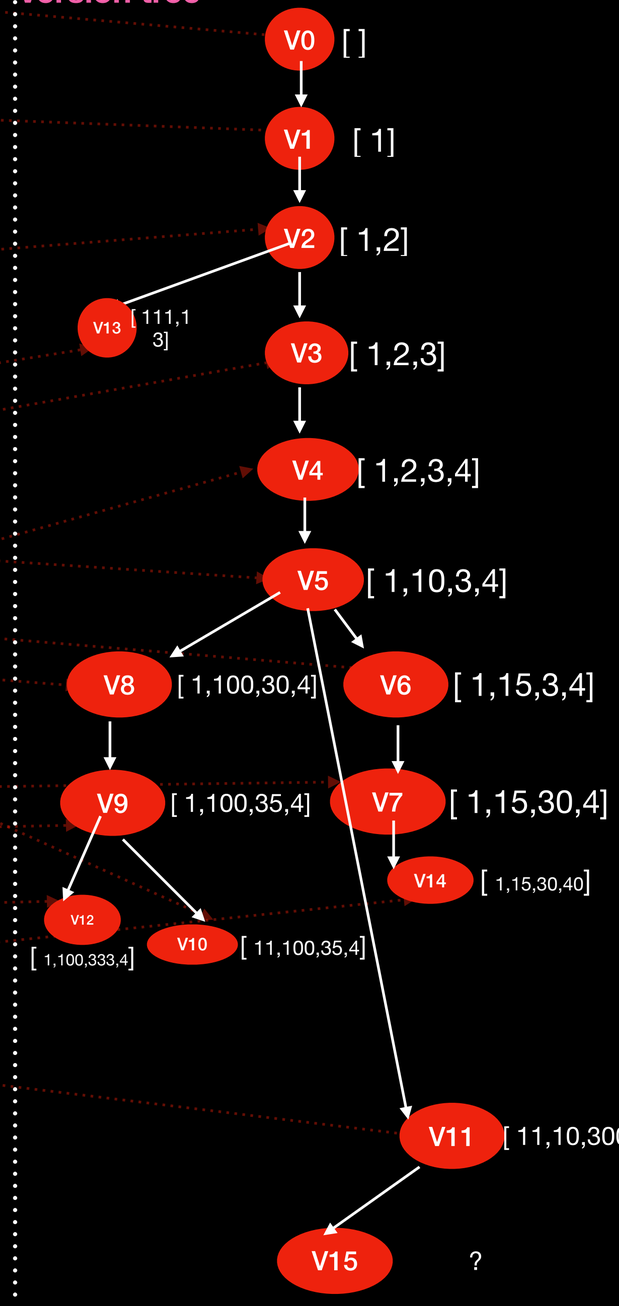
STEP 2, INTERMEDIATE DISCUSSION

*TRANSFER **AROUND 50%** MODS TO C' FROM C*

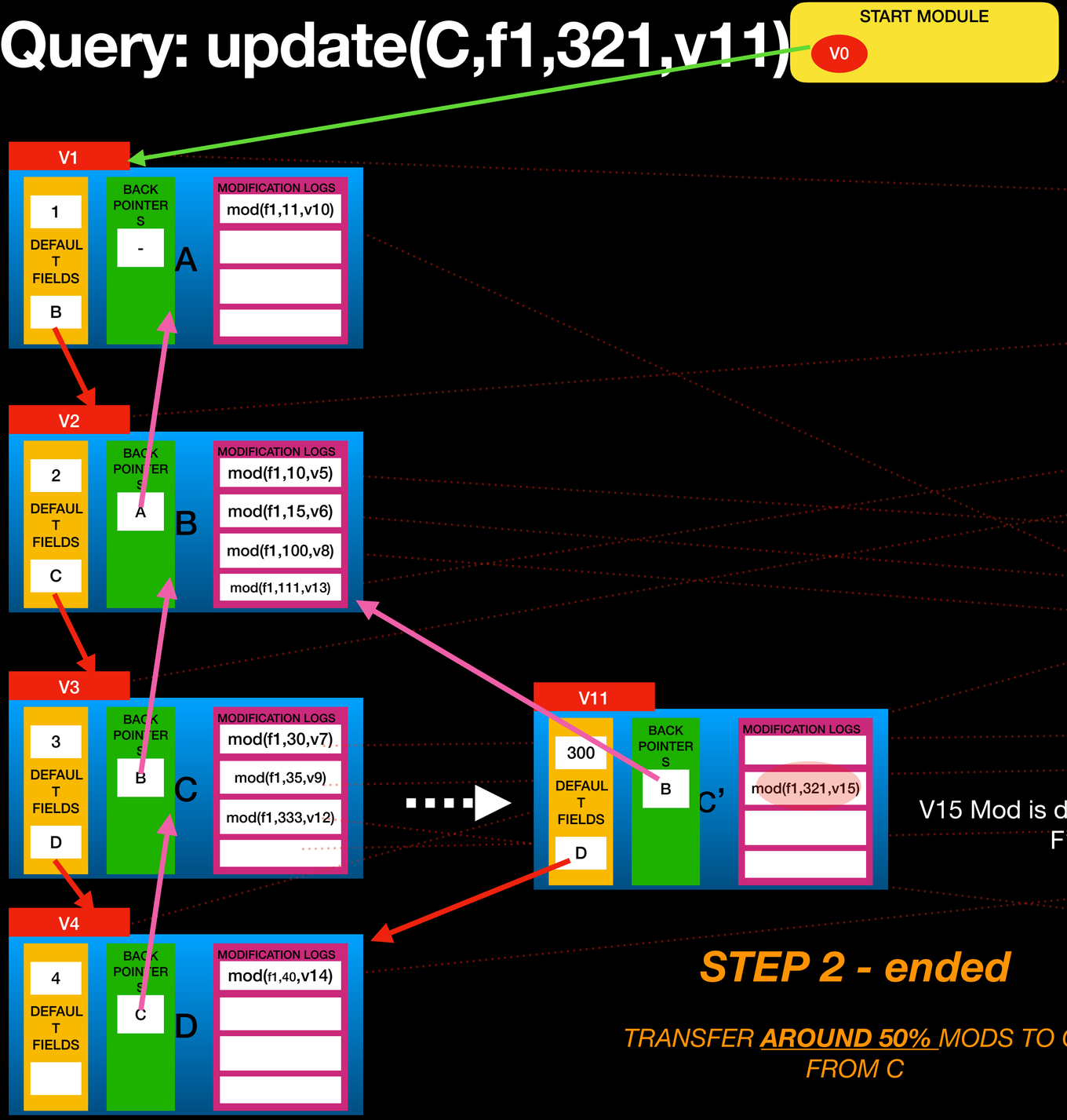
PTO->

Current time, t = 15

Version tree



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



V15 Mod is defined as user input
F1 -> 321

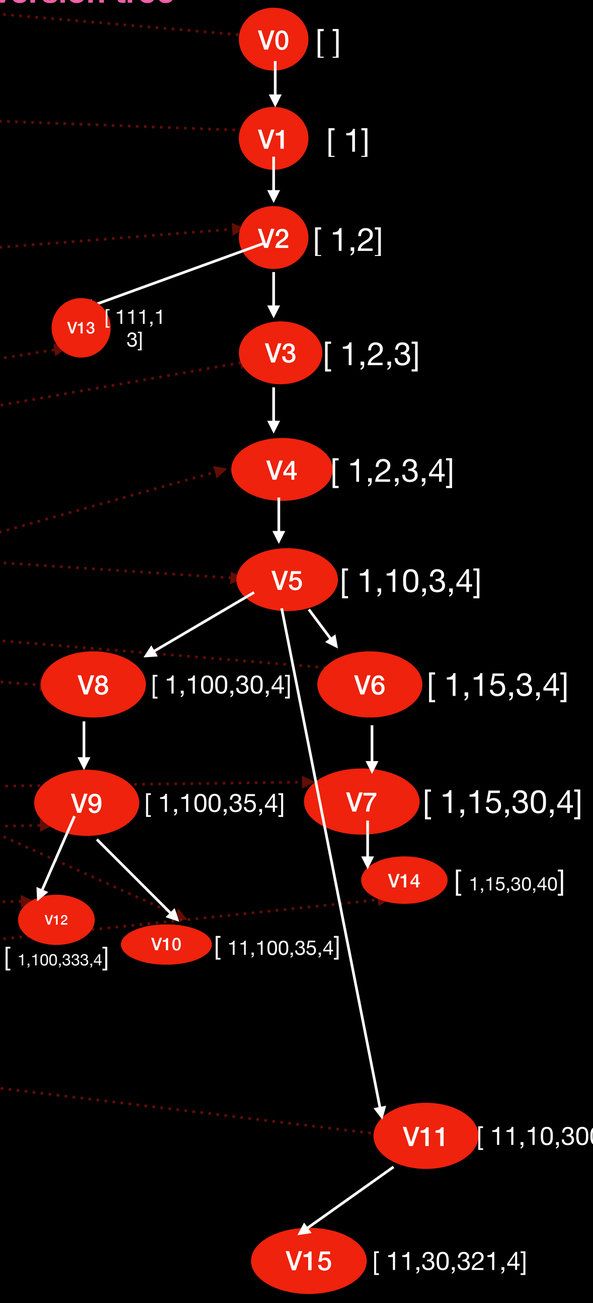
STEP 2 - ended

*TRANSFER **AROUND 50%** MODS TO C'*
FROM C

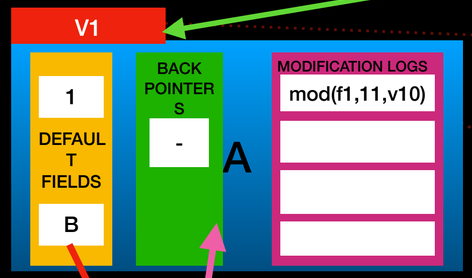
PTO->

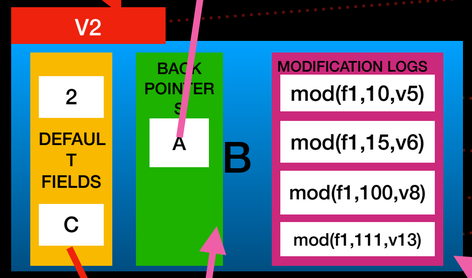
Current time, t = 15

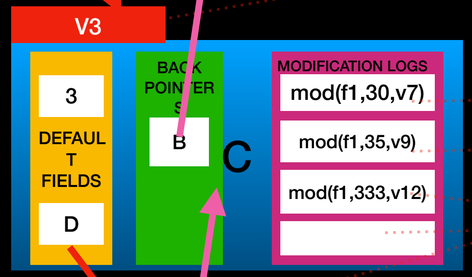
Version tree

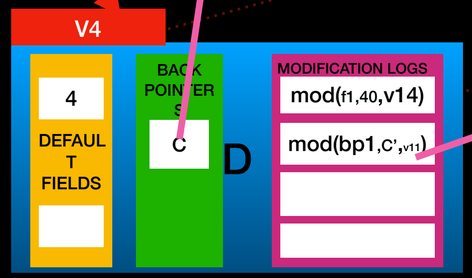


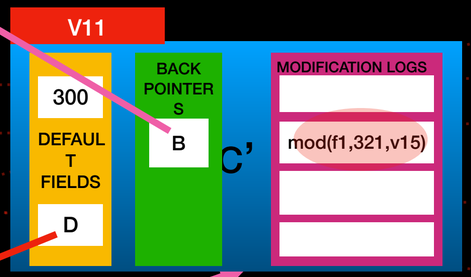
Query: update(C,f1,321,v11)











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

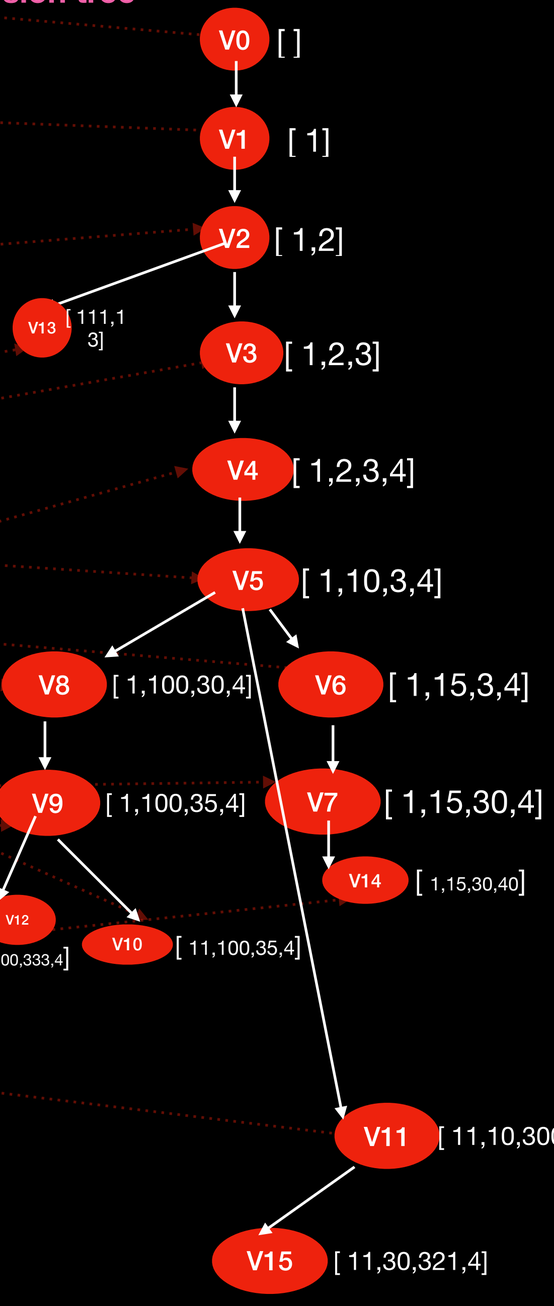
update(D,bp1,C',v11)

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->

Current time, t = 15

Version tree

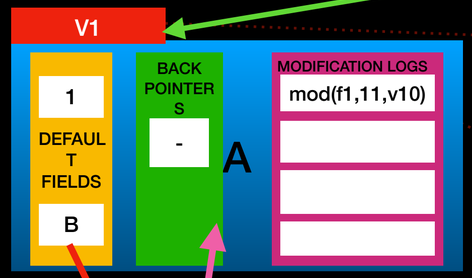


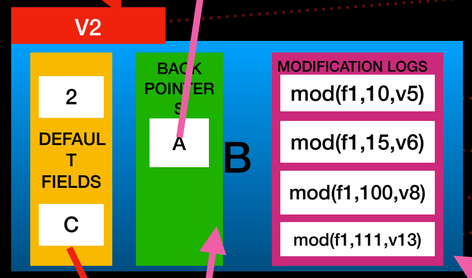
Query: update(C,f1,321,v11)

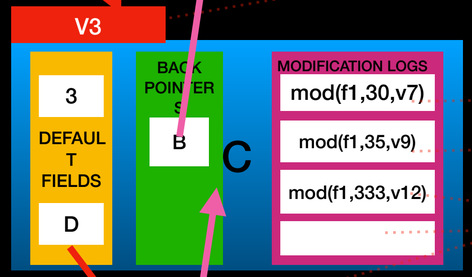
START MODULE

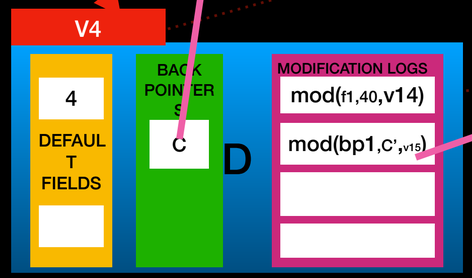
V0

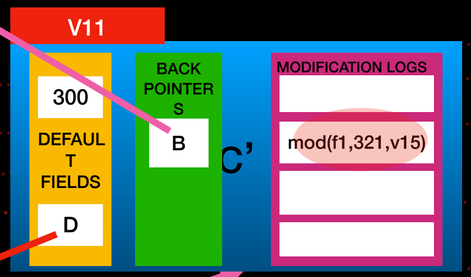
Current time, t = 15











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

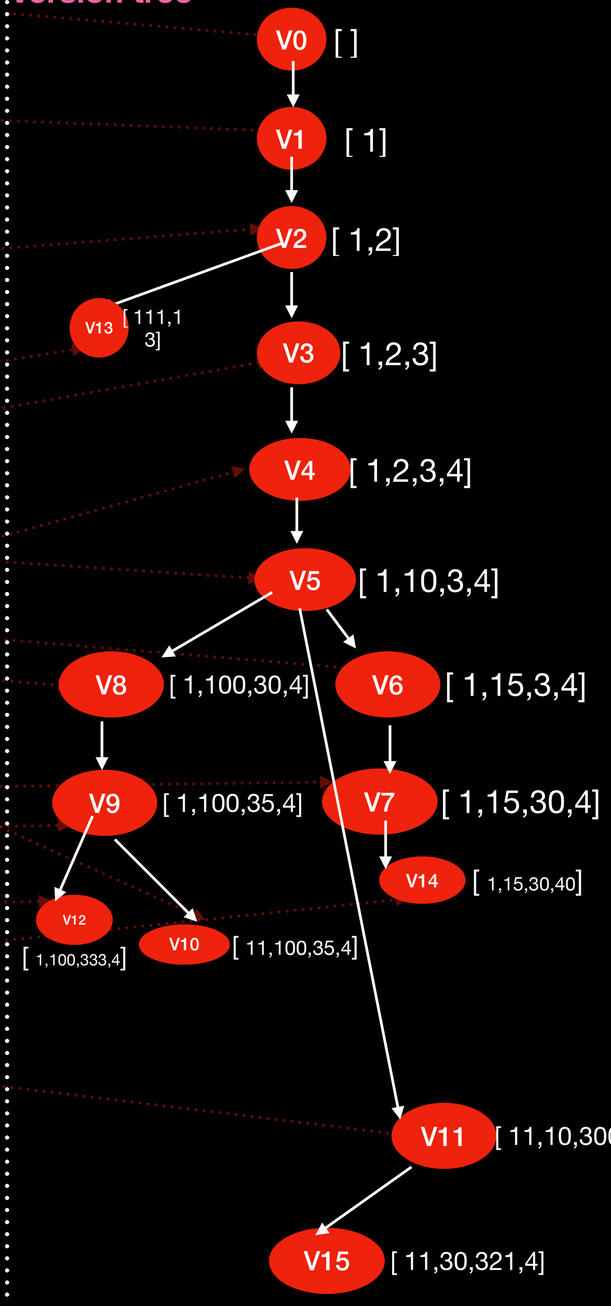
STEP 4

RECURSIVELY
MODIFY THE FORWARD POINTERS
OF ANCESTORS

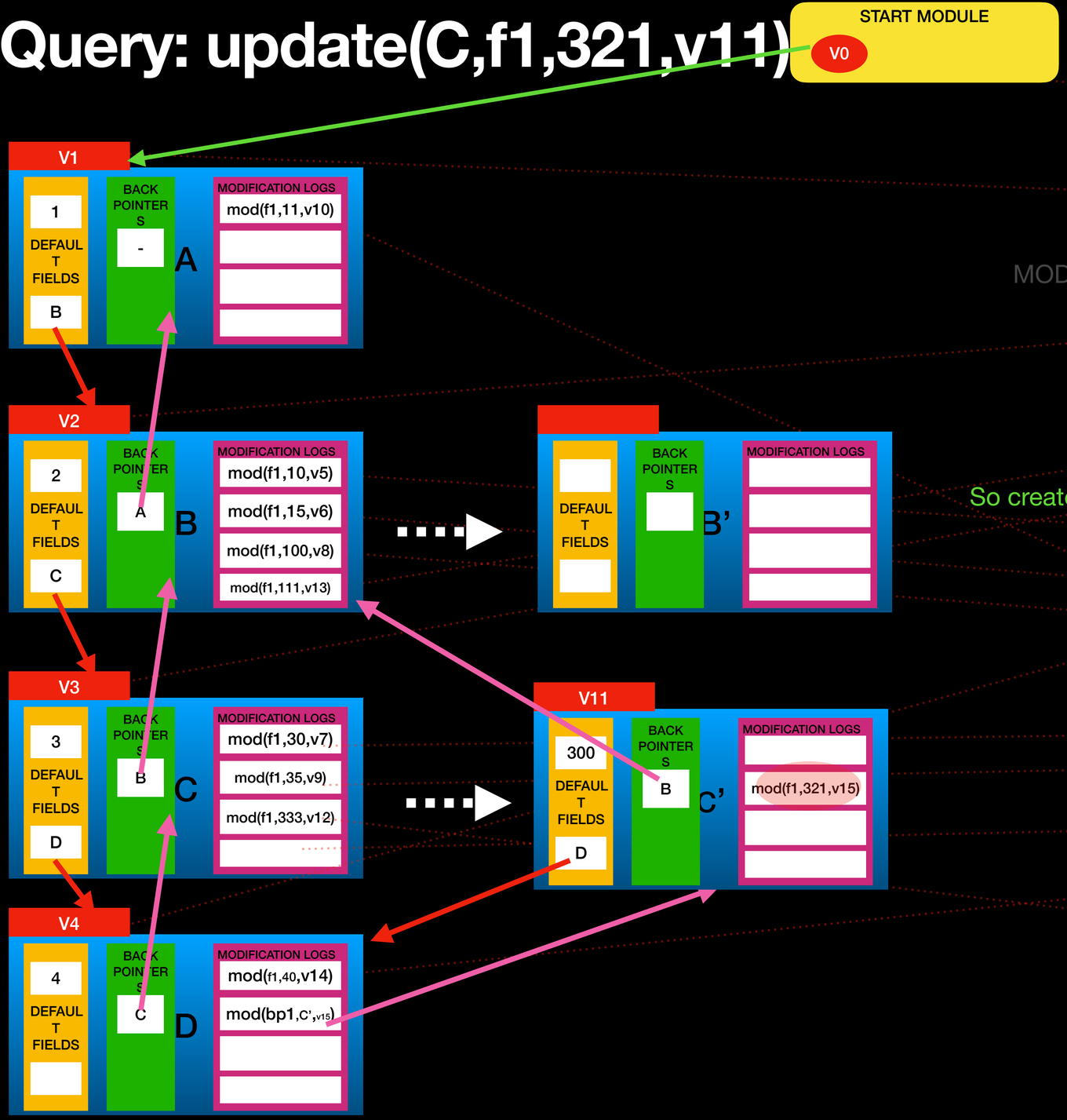
update(B,f2,C',*v11*)

PTO->

Version tree



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



STEP 4

RECURSIVELY
MODIFY THE FORWARD POINTERS
OF ANCESTORS

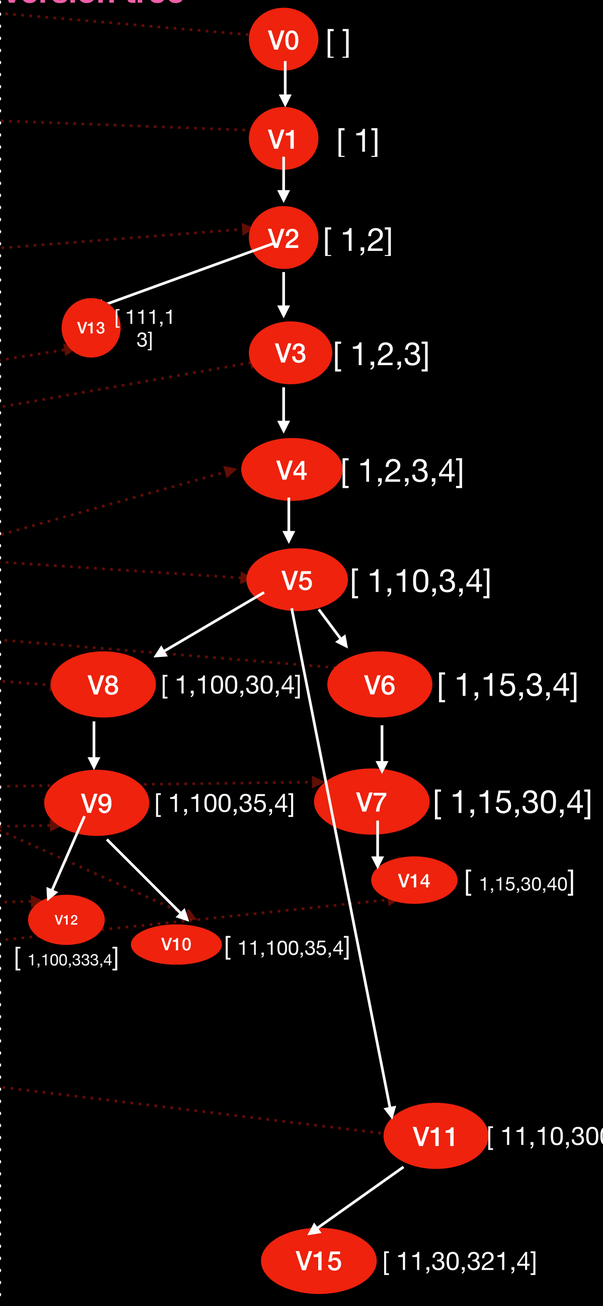
update(B,f2,C',v11)

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

PTO->

Current time, t = 15

Version tree



mod(f1,10,v5)
mod(f1,15,v6)
mod(f1,100,v8)
mod(f1,111,v13)
mod(f2,C',v11)

Topostort

mod(f1,10,v5)
mod(f1,15,v6)
mod(f1,100,v8)
mod(f2,C',v11)
mod(f1,111,v13)

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

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

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

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.

mod(f1,10,v5)
mod(f1,15,v6)
mod(f1,100,v8)
mod(f2,C',v11)
mod(f1,111,v13)

mod(f1,111,v13)

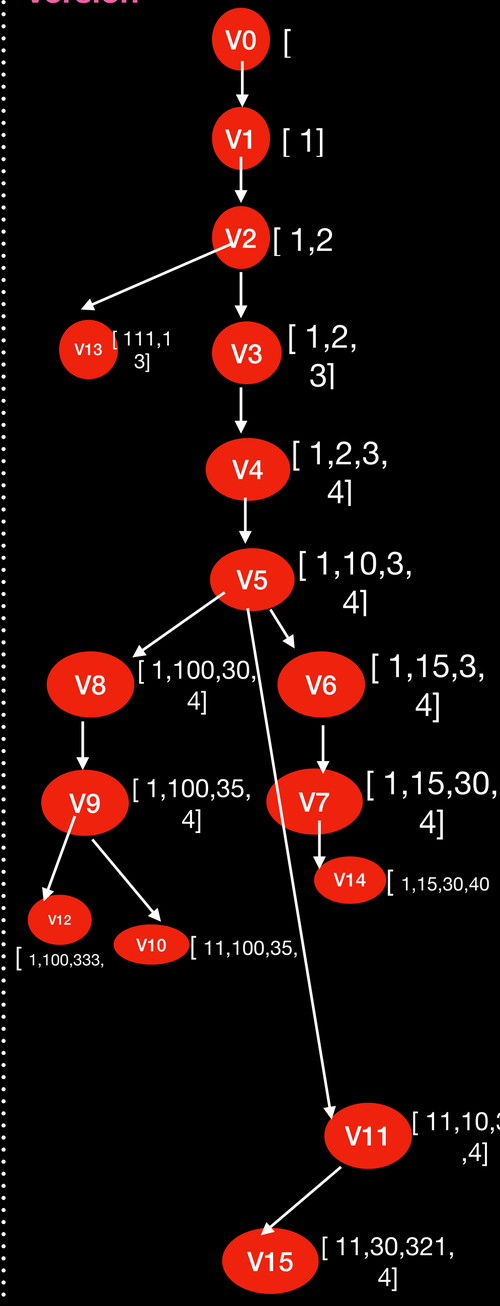
THIS PART WILL REMAIN IN THE OLD NODE

mod(f1,10,v5)
mod(f1,15,v6)
mod(f1,100,v8)
mod(f2,C',v11)

THIS PART WILL GO TO THE NEW COPIED NODE

PIVOT
THIS WILL DETERMINE THE DEFAULT VERSION OF NEWLY COPIED NODE

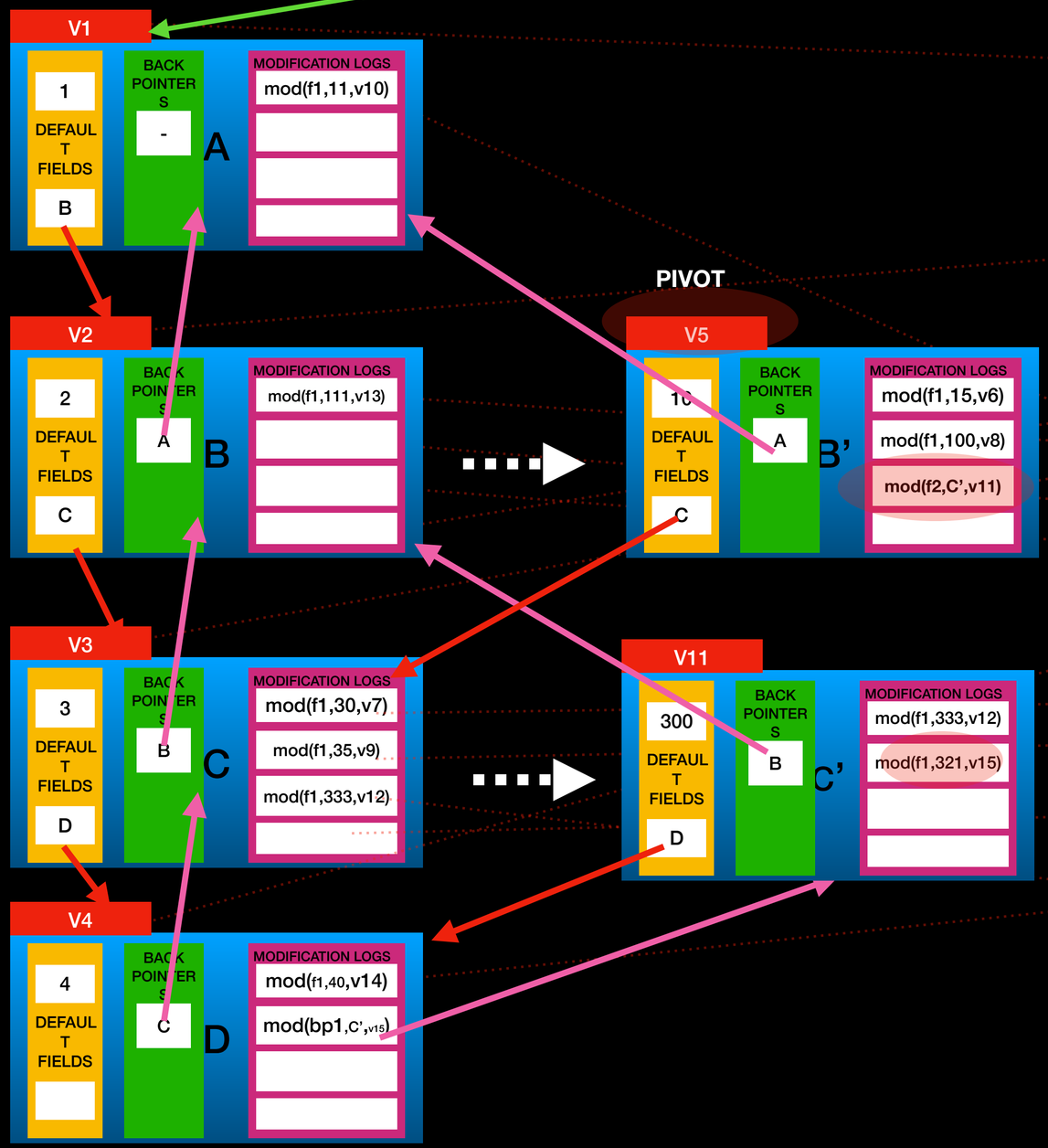
Version



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

START MODULE

V0



STEP 4

RECURSIVELY
MODIFY THE FORWARD POINTERS
OF ANCESTORS

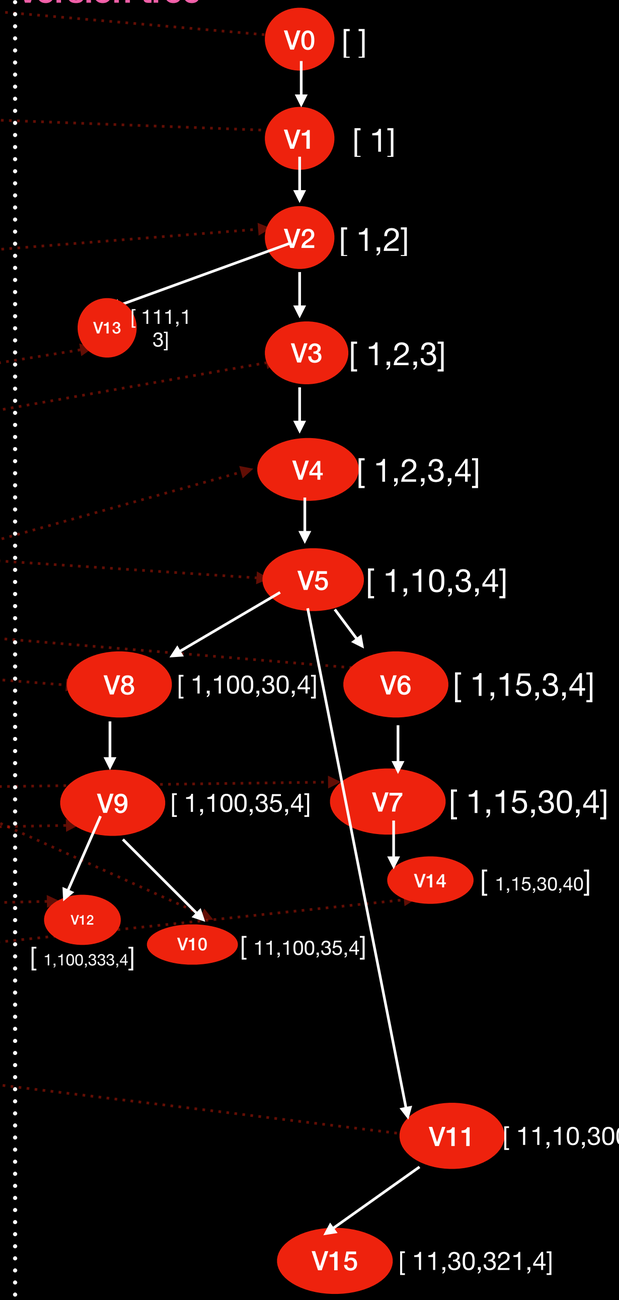
update(B,f2,C',v11)

ModLogs of B is full!
So create a copy of B at v15 and so on …

PTO->

Current time, t = 15

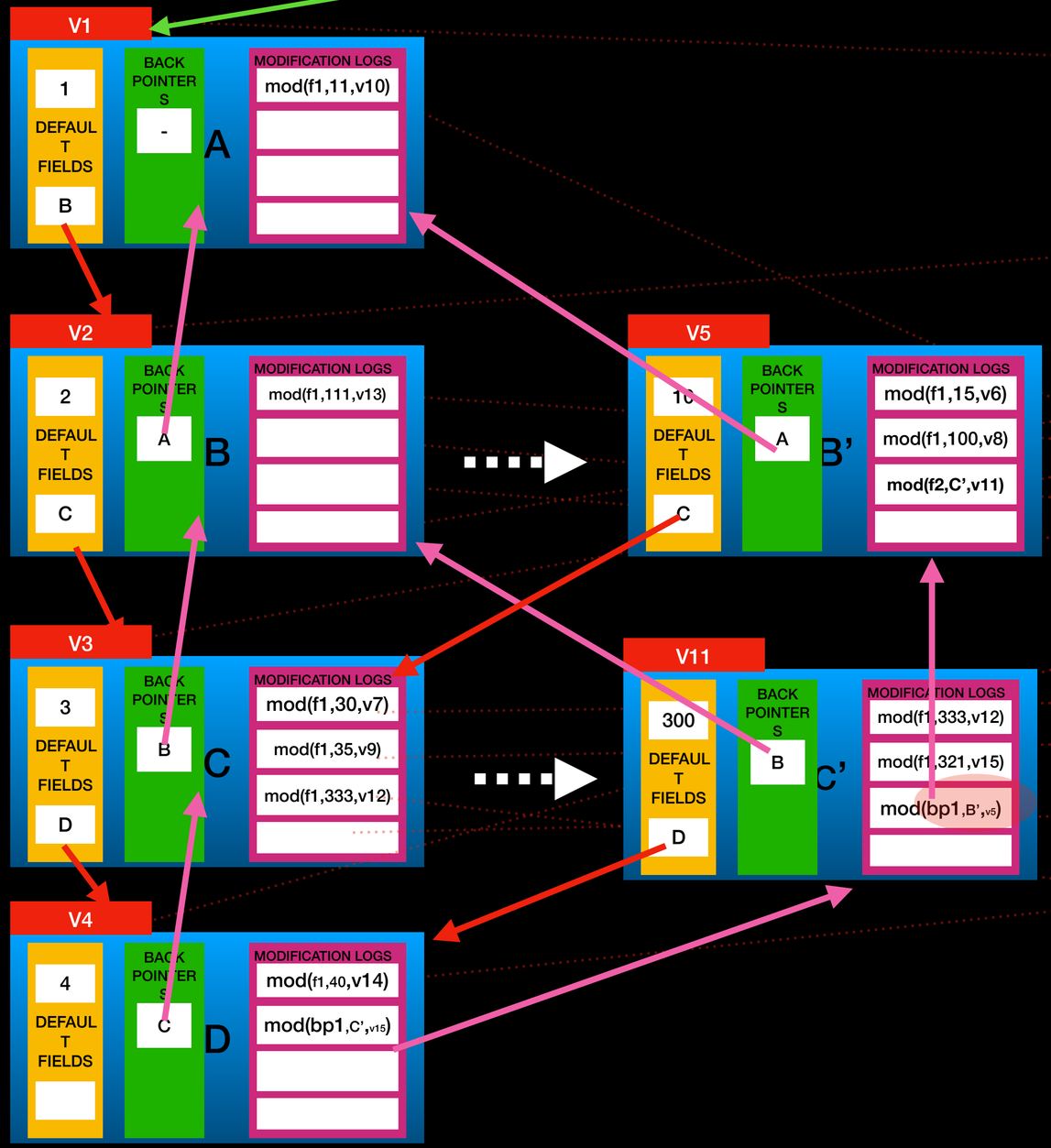
Version tree



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

START MODULE

V0



STEP 4

RECURSIVELY
MODIFY THE FORWARD POINTERS
OF ANCESTORS

update(B,f2,C',v11)

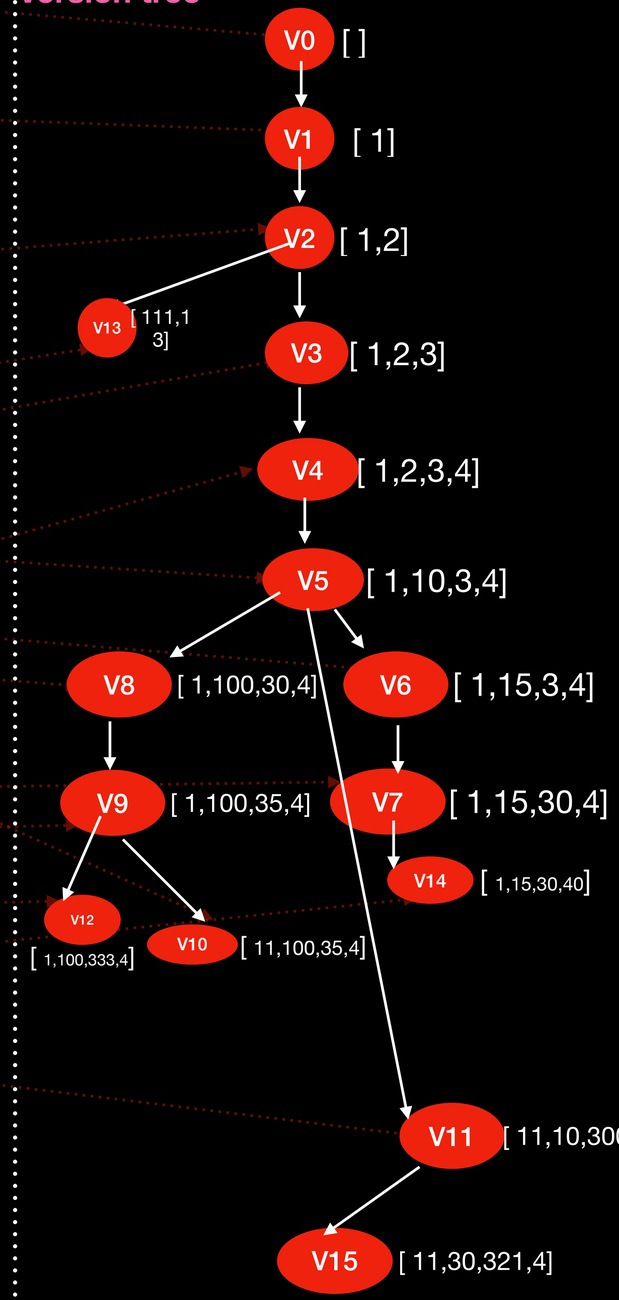
RECURSIVELY
MODIFY THE BACK POINTERS
OF LATEST SUCCESSOR

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

PTO->

Current time, t = 15

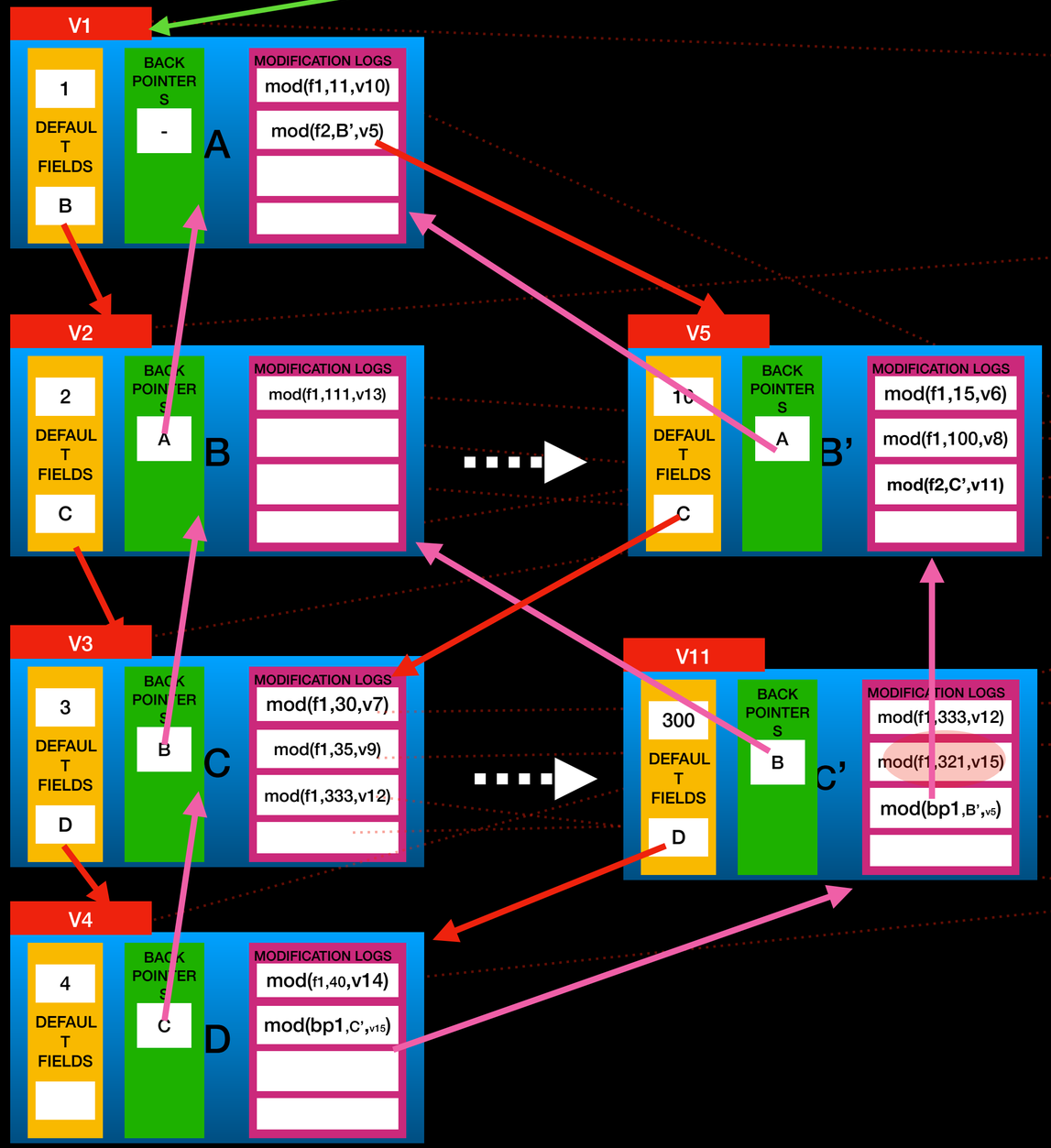
Version tree



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

START MODULE

V0



STEP 4

RECURSIVELY
MODIFY THE FORWARD POINTERS
OF ANCESTORS

update(B,f2,C',v11)

RECURSIVELY
MODIFY THE FORWARD POINTERS
OF ANCESTORS

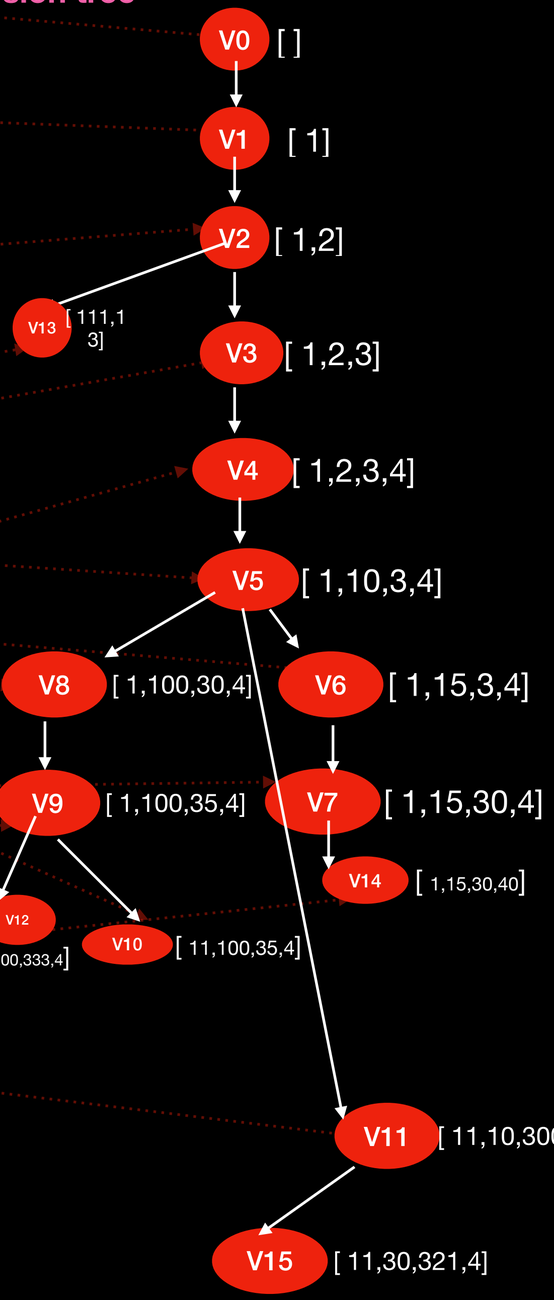
update(A,f2,B',v5)

Here, we face no problem
As A has sufficient mod logs
Left

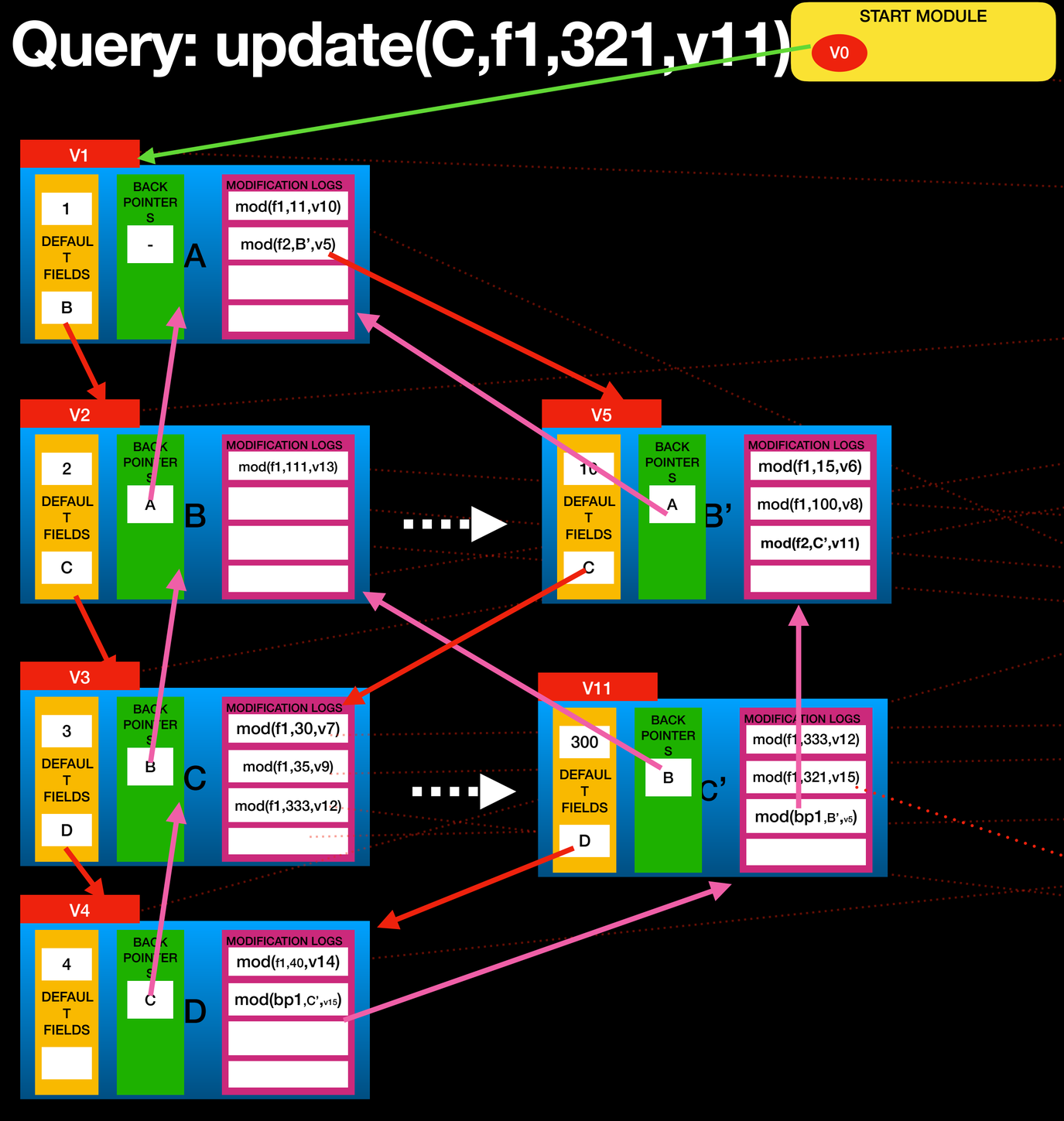
PTO->

Current time, t = 15

Version tree



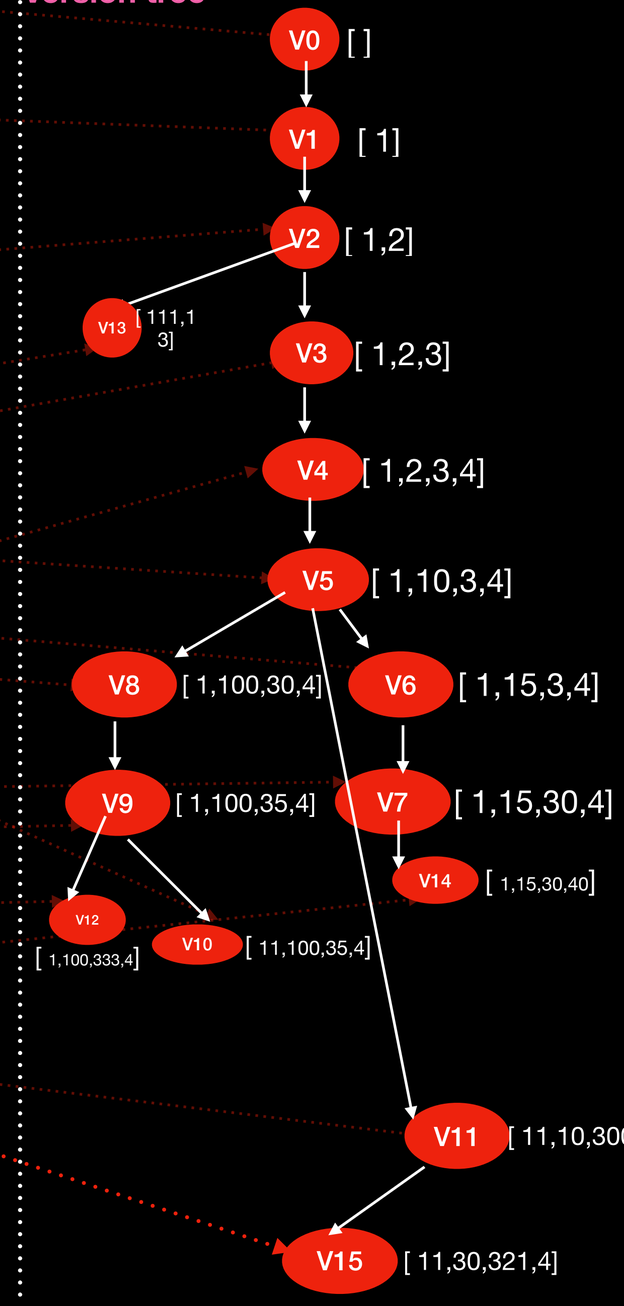
Query: update(C,f1,321,v11)



FINAL STATE

Current time, t = 15

Version tree



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

START MODULE

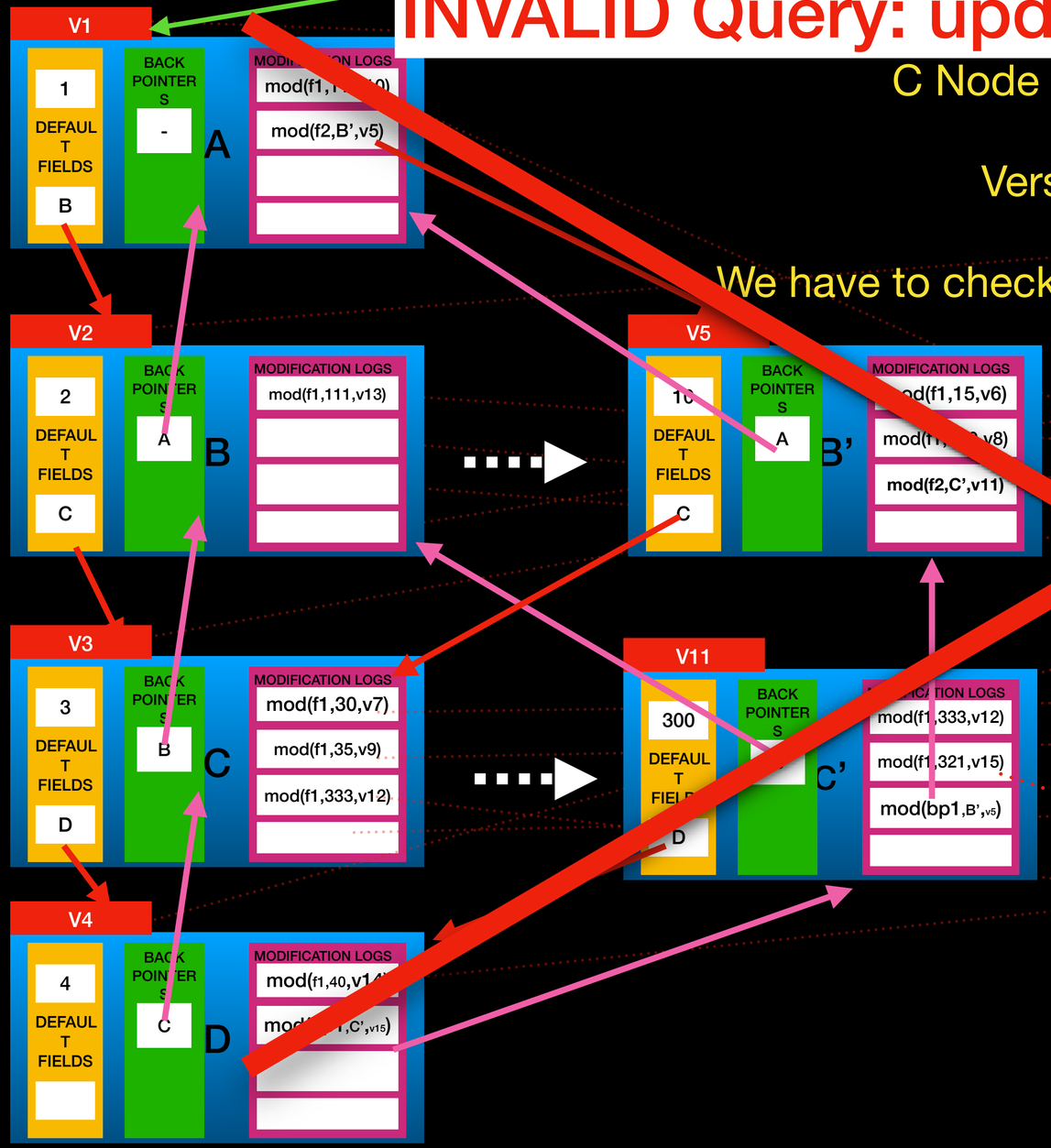
V0

INVALID Query: update(C,f1,3000,v13)

C Node was absent
In
Version 13

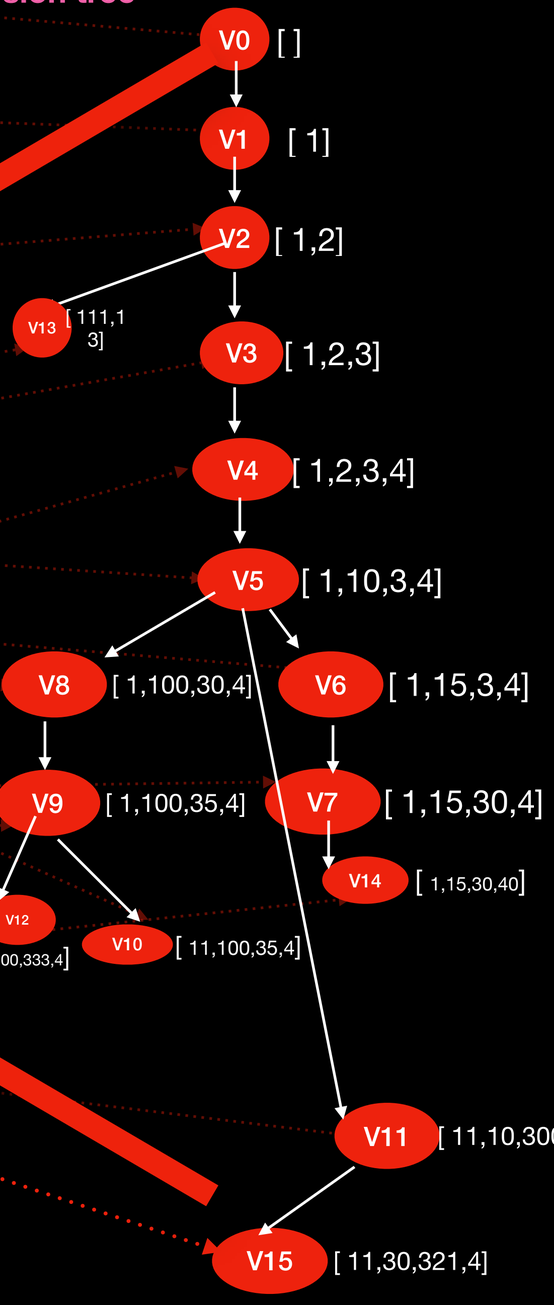
We have to check it in implementation

Current time, t = 15



FINAL STATE

Version tree



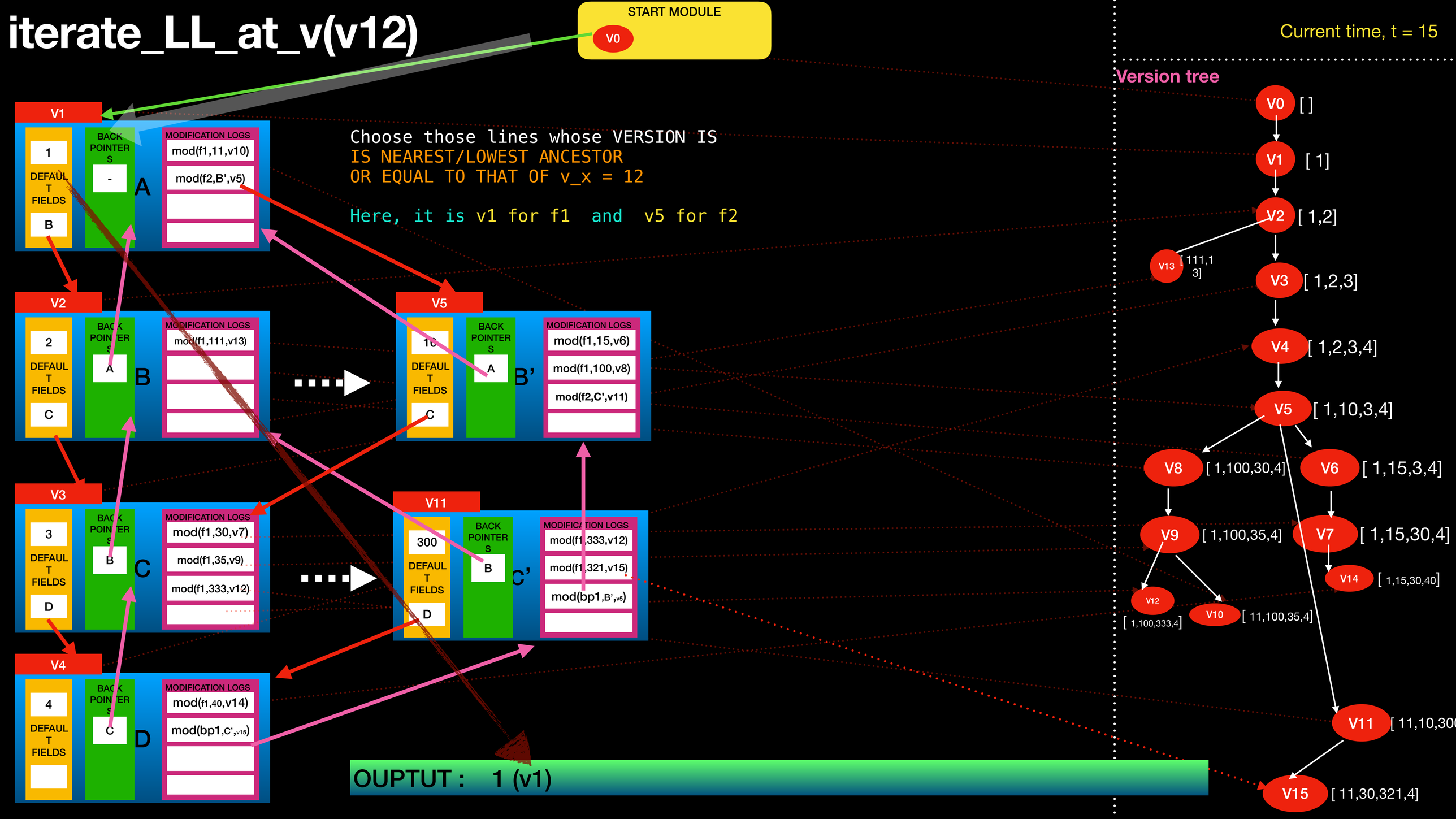
Same Way we can show

-> Deletion of Node remove(x,v)
-> Insertion of Node add(x, y, v)

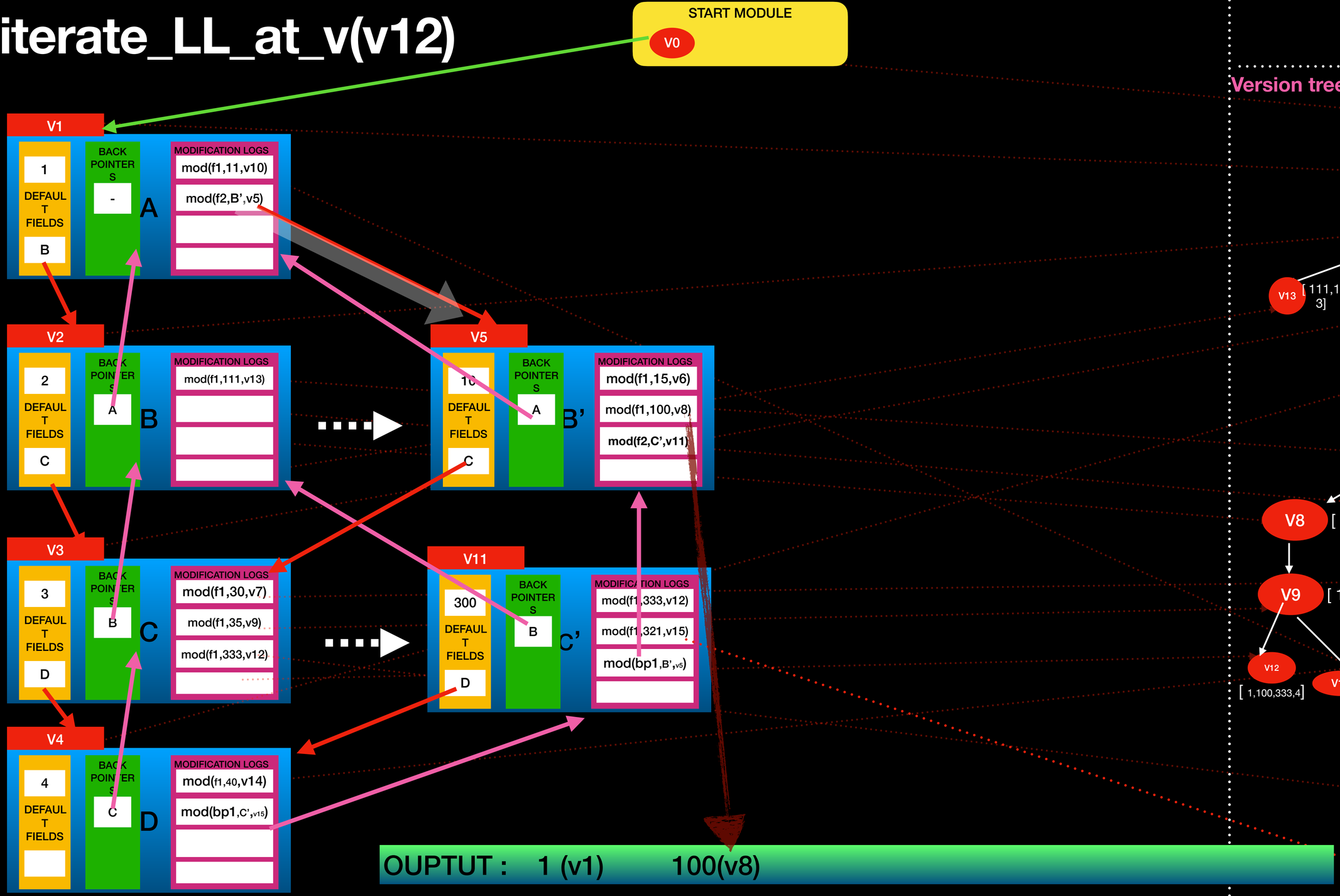
Shown in partial persistent mode

iterate_LL_at_v(v12)

iterate_LL_at_v(v12)

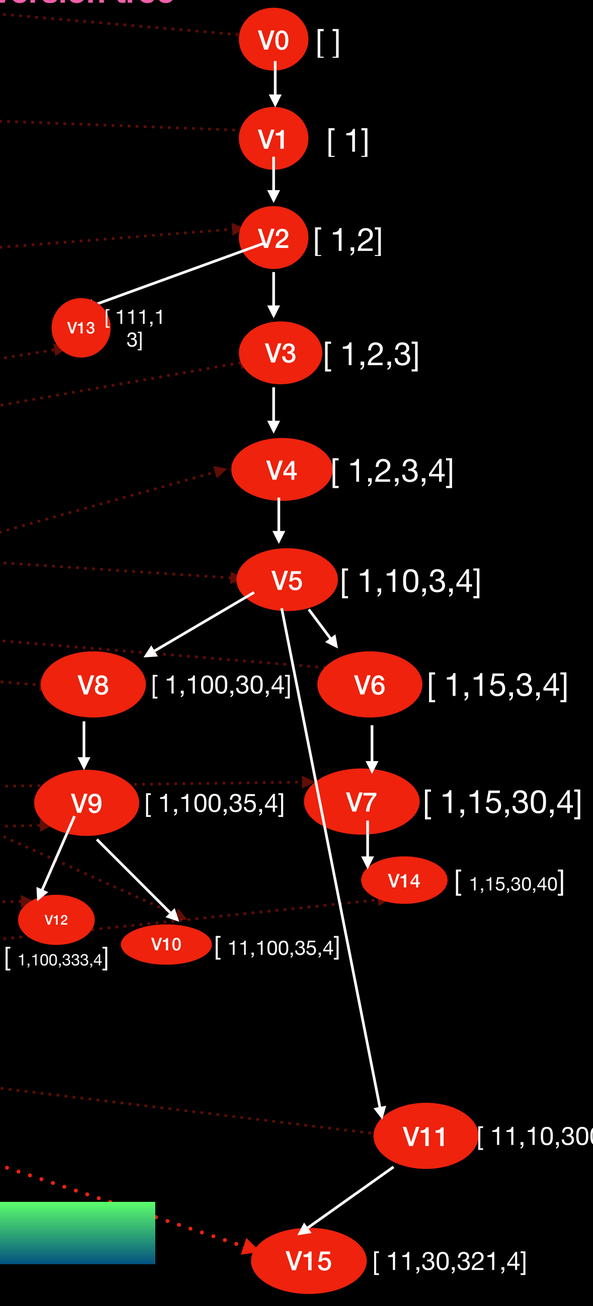


iterate_LL_at_v(v12)

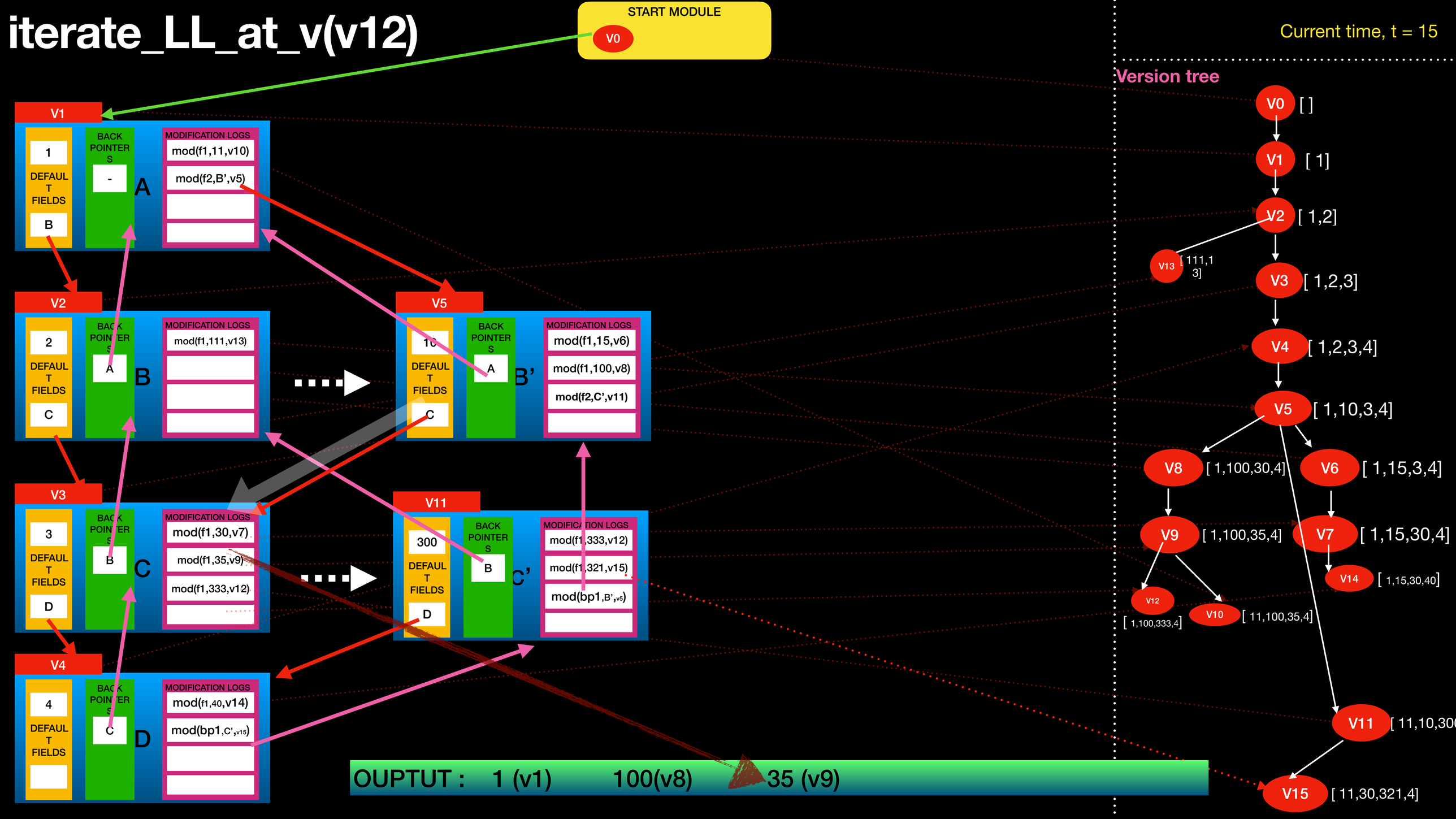


Current time, t = 15

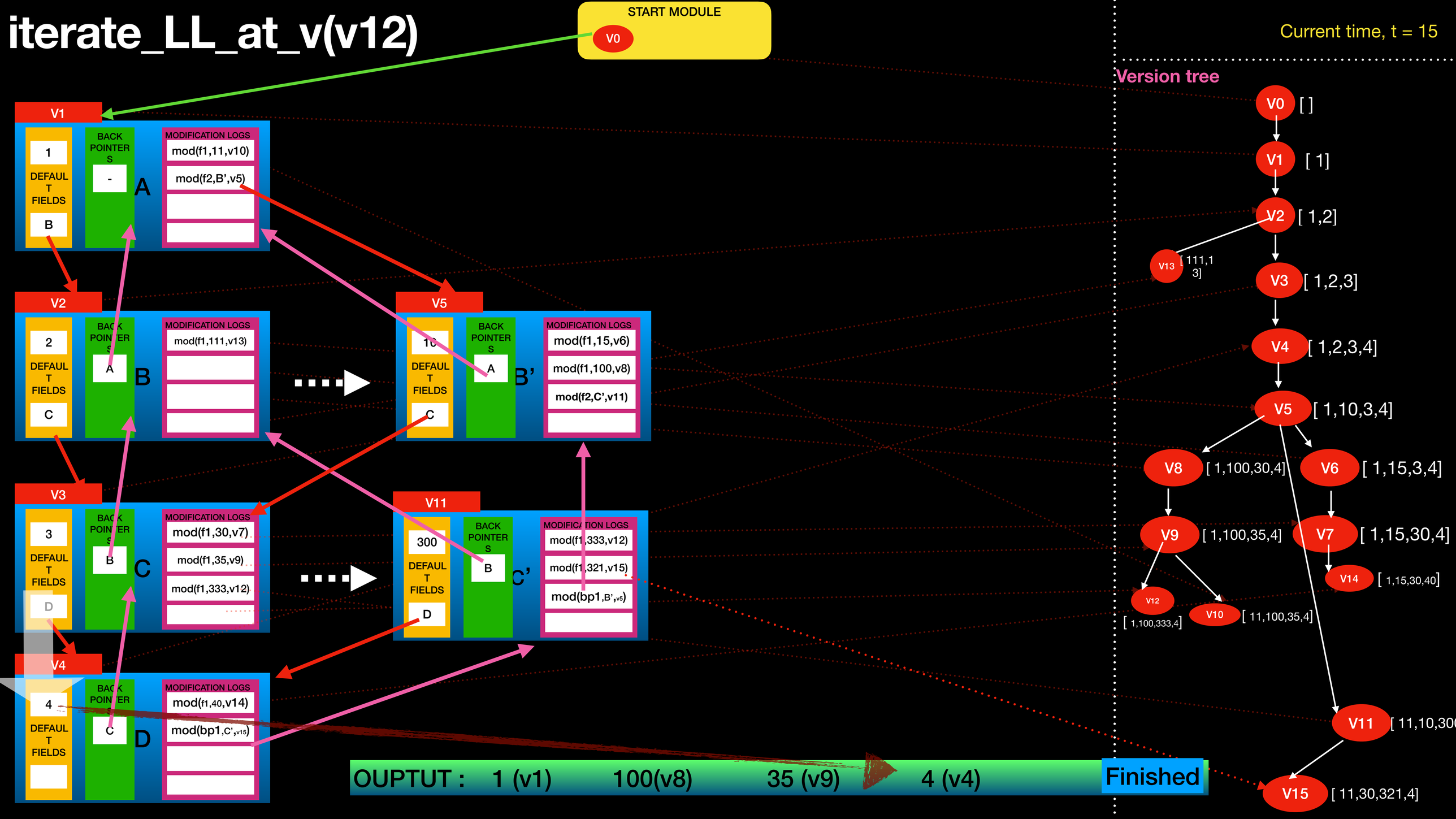
Version tree



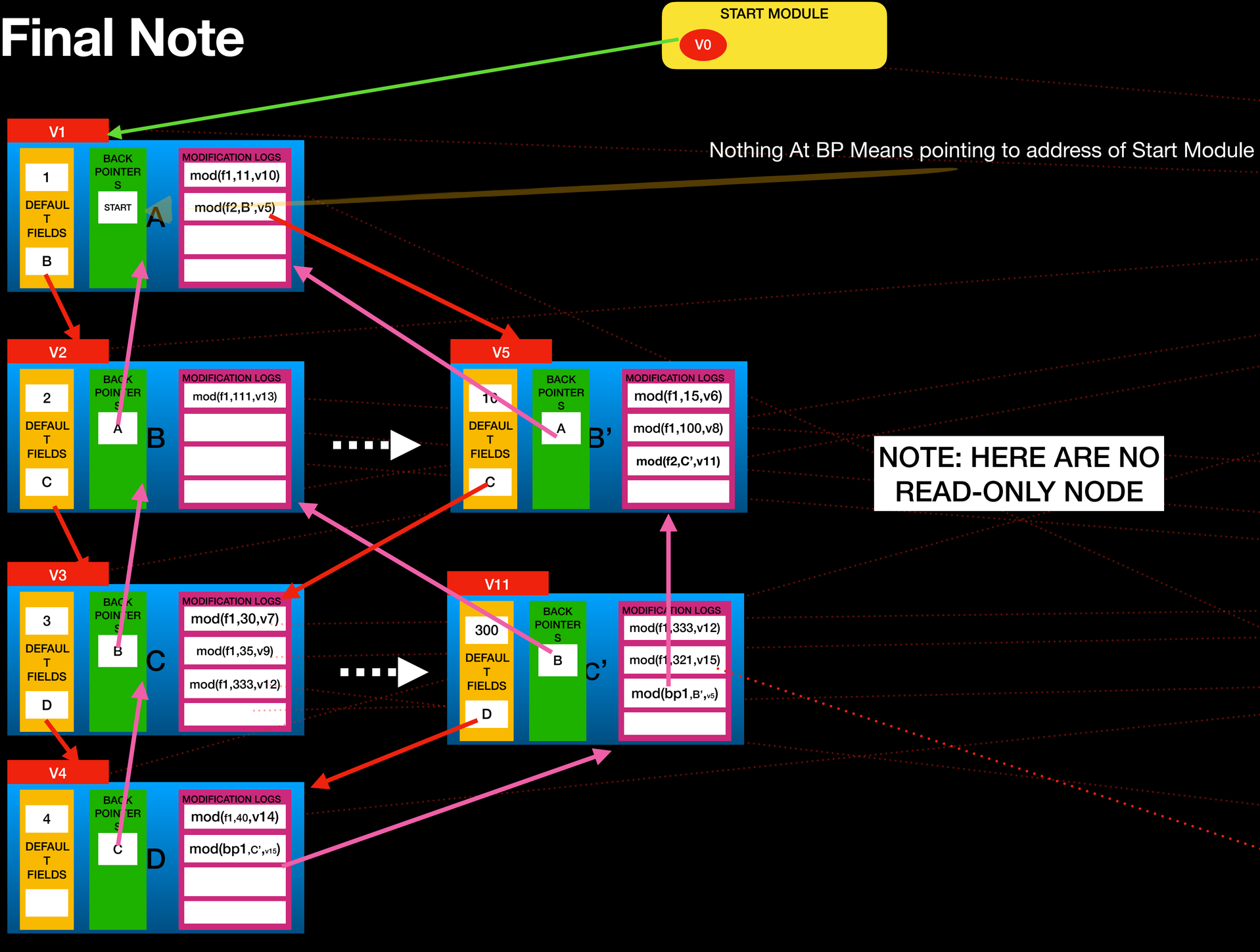
iterate_LL_at_v(v12)



iterate_LL_at_v(v12)

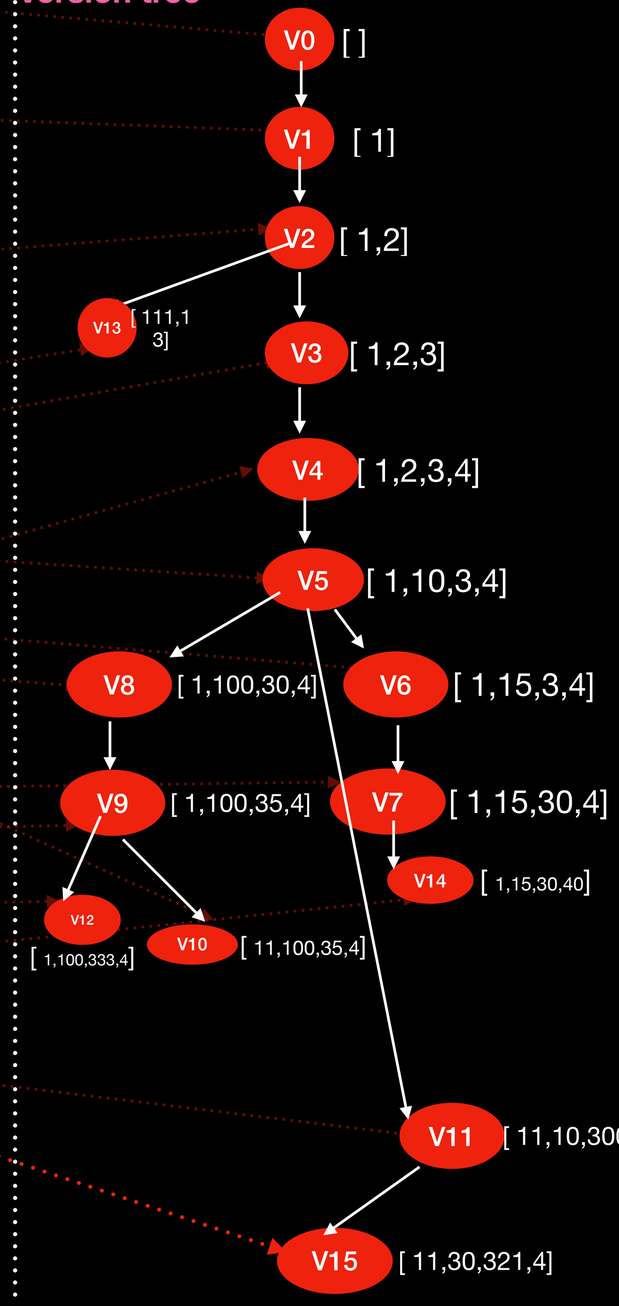


Final Note



Current time, t = 15

Version tree



Persistent Stack

Main Concept in Persistent Stack

Kushal Das 30/12/2021

Here I have used pointer machine to implement my fully persistent Stack

I have taken some basic functions of stack

- Push
- Pop
- IsEmpty
- Print

I have taken the print function just to see what the stack was in a particular version.

In this I have implemented a list using pointer machine in which each node has:

- Data
- Next pointer
- Version
- Back pointer
- Modifications log
    - Data
    - Next and Back pointers
    - Version

Then I implemented an insert at an index and delete at an index in any version functions.

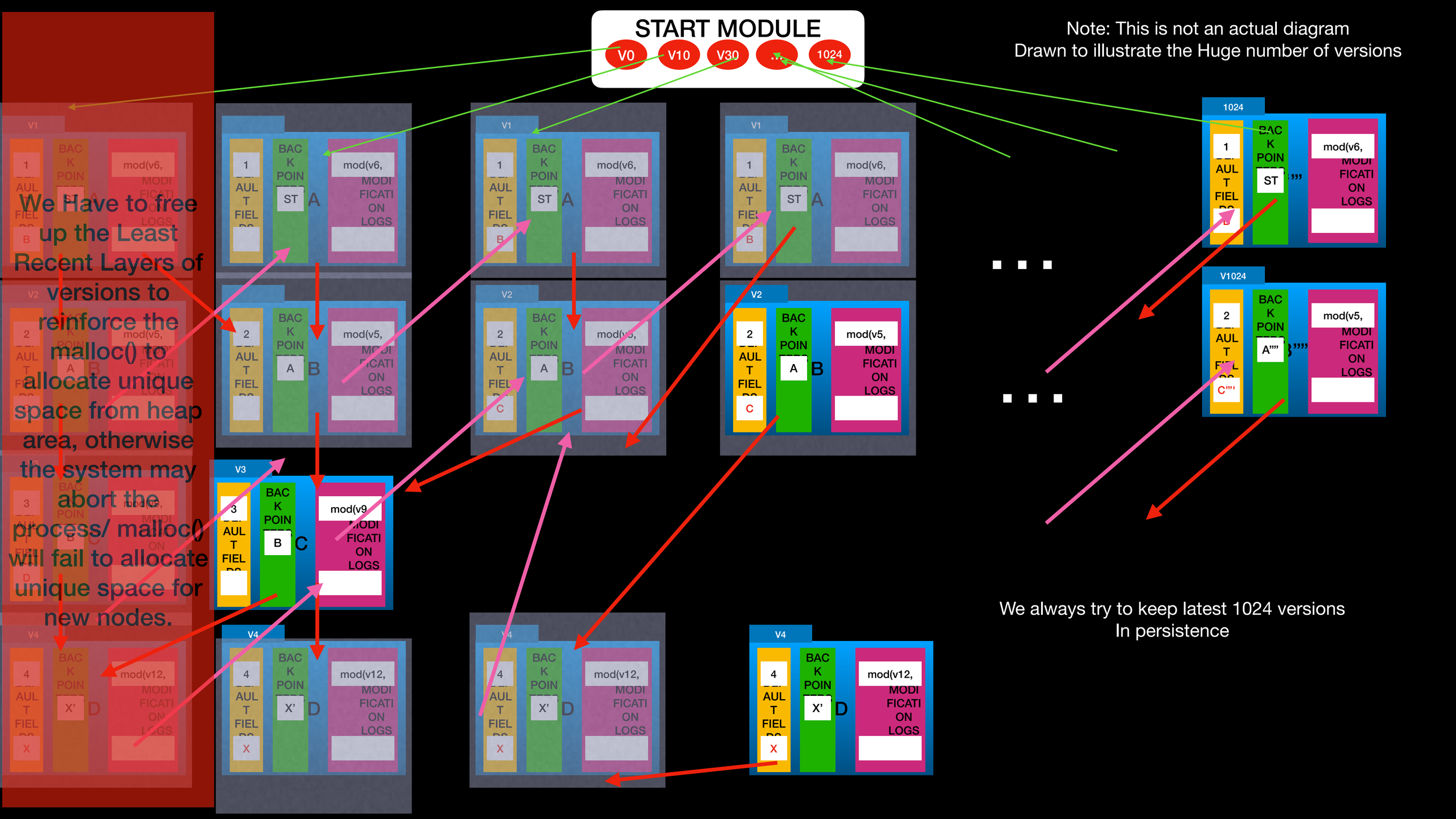
Then for stack the index is 0 i.e. insert at start and delete at start.

Persistent Data structure

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

1. Try To Keep A Limited Number of Latest Version in Persistence

Because, the PPDS are very memory consuming.


START MODULE
V0
V10
V30
...
1024
Note: This is not an actual diagram
Drawn to illustrate the Huge number of versions
We Have to free up the Least Recent Layers of versions to reinforce the malloc() to allocate unique space from heap area, otherwise the system may abort the process/ malloc() will fail to allocate unique space for new nodes.
We always try to keep latest 1024 versions
In persistence
V1
V2
V3
V4
1024
V1024
BACK POINTER
DEFAULT FIELDS
MODIFICATION LOGS
mod(v6,
mod(v5,
mod(v9
mod(v12,
ST
A
B
C
D
X
X'
1
2
3
4
A""
C""'

Supporting Code

Unit Test To Validate The Expected Result And Actual Result

```cpp
#include "gtest/gtest.h"
#include <iostream>
#include <vector>
#include "src/lib/headers_preprocessors.h"
#include "src/lib/PPDS_LinkedList.h"
#include "src/lib/test_with_Vector_and_PDSLL.h"

using namespace std;

const int max_mod = 3;
const int max_size = 200;

TEST(VECT_AND_PDSLL_VALIDATION_TEST, COMPARE_VECTOR_VS_PDSLL) {

    vector<int>vect(max_size,SENTINEL);
    PPDS_LINKED_LIST<int>list(max_mod);
    VECT_AND_PDSLL vp_test(max_mod,max_size);
    vector<int>actualResult,expectedResult;


    vp_test.randomized_Insert(list,vect);
    expectedResult = vp_test.trimVect(vect);
    actualResult = list.iterate_and_print_F1_at_ver_Vect(list.getCurTime());

    EXPECT_EQ(expectedResult, actualResult);  // random insert
    expectedResult.clear(); actualResult.clear();


    vp_test.randomized_Update(list,vect);
    expectedResult = vp_test.trimVect(vect);
    actualResult = list.iterate_and_print_F1_at_ver_Vect(list.getCurTime());

    EXPECT_EQ(expectedResult, actualResult); // random update

}

```

max_size = 200

```
Target //tests:VALIDATION_TEST up-to-date:
  bazel-bin/tests/VALIDATION_TEST
INFO: Elapsed time: 2.258s, Critical Path: 2.05s
INFO: 3 processes: 1 internal, 2 darwin-sandbox.
INFO: Build completed successfully, 3 total actions
INFO: Running command line: external/bazel_tools/tools/test/test-setup.sh tests/VALIDATIONINFO: Build completed successfully, 3 total actions
exec ${PAGER:-/usr/bin/less} "$0" || exit 1
Executing tests from //tests:VALIDATION_TEST
-------------------------------------------------------------------------------
Running main() from gmock_main.cc
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from VECT_AND_PDSLL_VALIDATION_TEST
[ RUN      ] VECT_AND_PDSLL_VALIDATION_TEST.COMPARE_VECTOR_VS_PDSLL
[       OK ] VECT_AND_PDSLL_VALIDATION_TEST.COMPARE_VECTOR_VS_PDSLL (208 ms)
[----------] 1 test from VECT_AND_PDSLL_VALIDATION_TEST (208 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (208 ms total)
[  PASSED  ] 1 test.
```

max_size = 2000

```
Target //tests:VALIDATION_TEST up-to-date:
  bazel-bin/tests/VALIDATION_TEST
INFO: Elapsed time: 1.622s, Critical Path: 1.49s
INFO: 3 processes: 1 internal, 2 darwin-sandbox.
INFO: Build completed successfully, 3 total actions
INFO: Running command line: external/bazel_tools/tools/test/test-setup.sh tests/VALIDATIONINFO: Build completed successfully, 3 total actions
exec ${PAGER:-/usr/bin/less} "$0" || exit 1
Executing tests from //tests:VALIDATION_TEST
-------------------------------------------------------------------------------
Running main() from gmock_main.cc
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from VECT_AND_PDSLL_VALIDATION_TEST
[ RUN      ] VECT_AND_PDSLL_VALIDATION_TEST.COMPARE_VECTOR_VS_PDSLL
[       OK ] VECT_AND_PDSLL_VALIDATION_TEST.COMPARE_VECTOR_VS_PDSLL (5 ms)
[----------] 1 test from VECT_AND_PDSLL_VALIDATION_TEST (5 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (5 ms total)
[  PASSED  ] 1 test.
```

These Results Passed

Supporting Code

max_size = 3000

```
2 warnings generated.
Target //tests:VALIDATION_TEST up-to-date:
  bazel-bin/tests/VALIDATION_TEST
INFO: Elapsed time: 2.383s, Critical Path: 2.16s
INFO: 3 processes: 1 internal, 2 darwin-sandbox.
INFO: Build completed successfully, 3 total actions
INFO: Running command line: external/bazel_tools/tools/test/test-setup.sh tests/VALIDATIONINFO: Build completed successfully, 3 total actions
exec ${PAGER:-/usr/bin/less} "$0" || exit 1
Executing tests from //tests:VALIDATION_TEST
-------------------------------------------------------------------------------
Running main() from gmock_main.cc
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from VECT_AND_PDSLL_VALIDATION_TEST
[ RUN      ] VECT_AND_PDSLL_VALIDATION_TEST.COMPARE_VECTOR_VS_PDSLL
Assertion failed: (0), function NodeAtIndex, file PPDS_LinkedList.h, line 385.
```

Failed To Validate The Result

```
Current number Of Nodes Are Used: 3511
Current Length of LL: 2999
Total Update Request: 6000
Current Memory Usage For Allocation Of Nodes: 646024 Byte
Is the LL stable? 0
```

We can see that the length of the LL is not 3000 anymore

In-depth analysis of this failure , we shall do after exam

2. Application in JAVA

The Java programming language is not particularly functional. Despite this, the core JDK package java.util.concurrent includes CopyOnWriteArrayList and CopyOnWriteArraySet which are **partial persistent** structures, implemented using copy-on-write techniques

Class declaration

```
public class CopyOnWriteArrayList
   extends Object
implements List, RandomAccess, Cloneable, Serializable
```

1. CopyOnWriteArrayList is a thread-safe variant of ArrayList where operations which can
change the ArrayList (add, update, set methods) creates a clone of the underlying array.
2. CopyOnWriteArrayList is to be used in a Thread based environment where read operations
are very frequent and update operations are rare.
3. Iterator of CopyOnWriteArrayList will never throw ConcurrentModificationException.
4. Any type of modification to CopyOnWriteArrayList will not reflect during iteration since the iterator was created.
5. List modification methods like remove, set and add are not supported in the iteration.
This method will throw UnsupportedOperationException.
null can be added to the list.

```java
/**
 * Appends the specified element to the end of this list.
 *
 * @param e element to be appended to this list
 * @return {@code true} (as specified by {@link Collection#add})
 */
public boolean add(E e) {
    synchronized (lock) {
        Object[] es = getArray();
        int len = es.length;
        es = Arrays.copyOf(es, len + 1);
        es[len] = e;
        setArray(es);
        return true;
    }
}

```



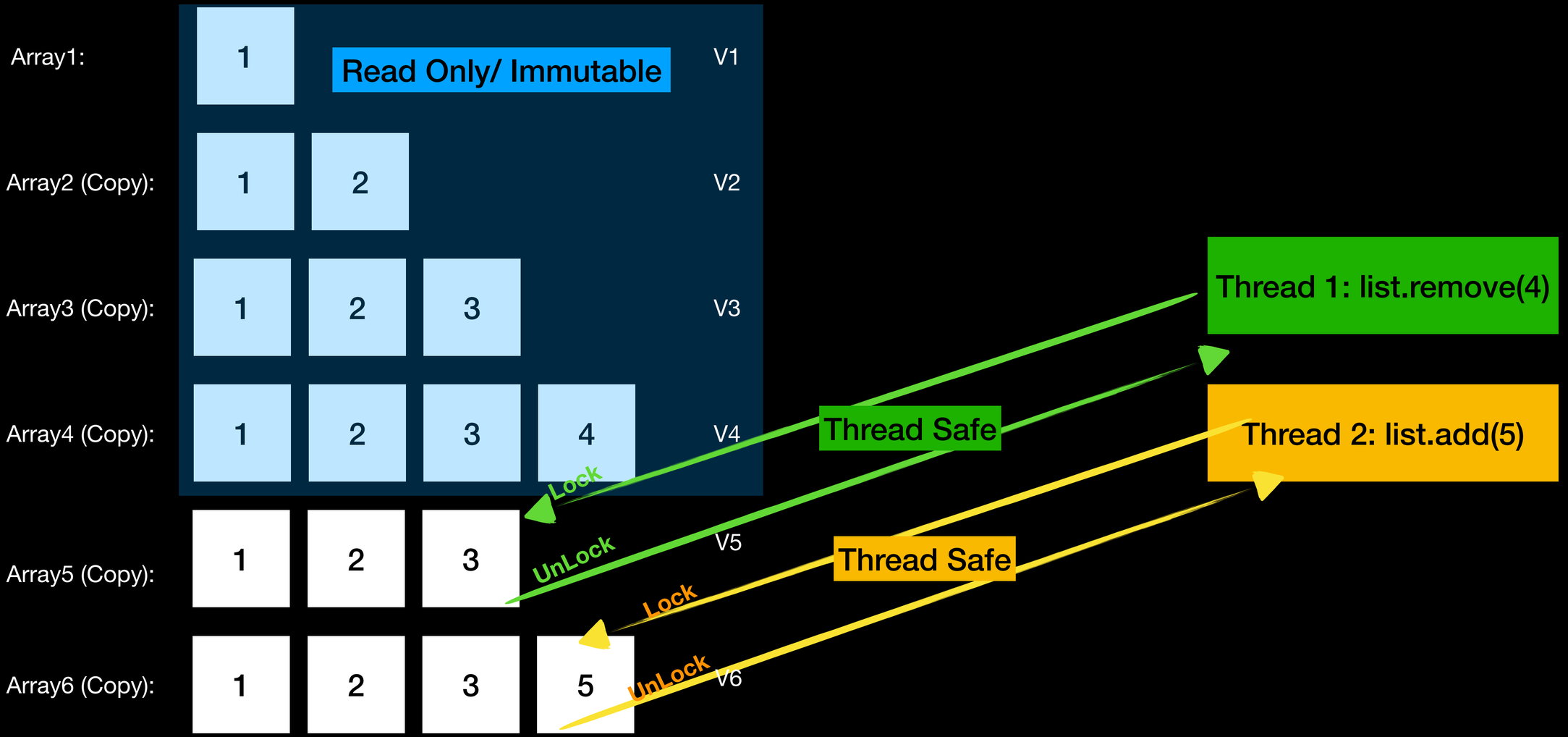
```java
/**
 * Inserts the specified element at the specified position in this
 * list. Shifts the element currently at that position (if any) and
 * any subsequent elements to the right (adds one to their indices).
 *
 * @throws IndexOutOfBoundsException {@inheritDoc}
 */
public void add(int index, E element) {
    synchronized (lock) {
        Object[] es = getArray();
        int len = es.length;
        if (index > len || index < 0)
            throw new IndexOutOfBoundsException(outOfBounds(index, len));
        Object[] newElements;
        int numMoved = len - index;
        if (numMoved == 0)
            newElements = Arrays.copyOf(es, len + 1);
        else {
            newElements = new Object[len + 1];
            System.arraycopy(es, 0, newElements, 0, index);
            System.arraycopy(es, index, newElements, index + 1,
                             numMoved);
        }
        newElements[index] = element;
        setArray(newElements);
    }
}

```


Code Taken From underlying source code of “CopyOnWriteArrayList” class

Note: We have tried to implement the same Functionality in C++.
[Code Attached]

Idea:

```
1       CopyOnWriteArrayList<String> list
2               = new CopyOnWriteArrayList<>();
3
```

3. Application In Clojure

Clojure

Like many programming languages in the Lisp family, Clojure contains an implementation of a linked list, but unlike other dialects its implementation of a Linked List has enforced persistence instead of being persistent by convention.[19] Clojure also has efficient implementations of persistent vectors, maps, and sets based on persistent hash array mapped tries. These data structures implement the mandatory read-only parts of the Java collections framework.[20]

The designers of the Clojure language advocate the use of persistent data structures over mutable data structures because they have value semantics which gives the benefit of making them freely shareable between threads with cheap aliases, easy to fabricate, and language independent.[21]

These data structures form the basis of Clojure's support for parallel computing since they allow for easy retries of operations to sidestep data races and atomic compare and swap semantics.[22]

[Slides in Clojure are attached Separately]

Extract Taken from wikipedia

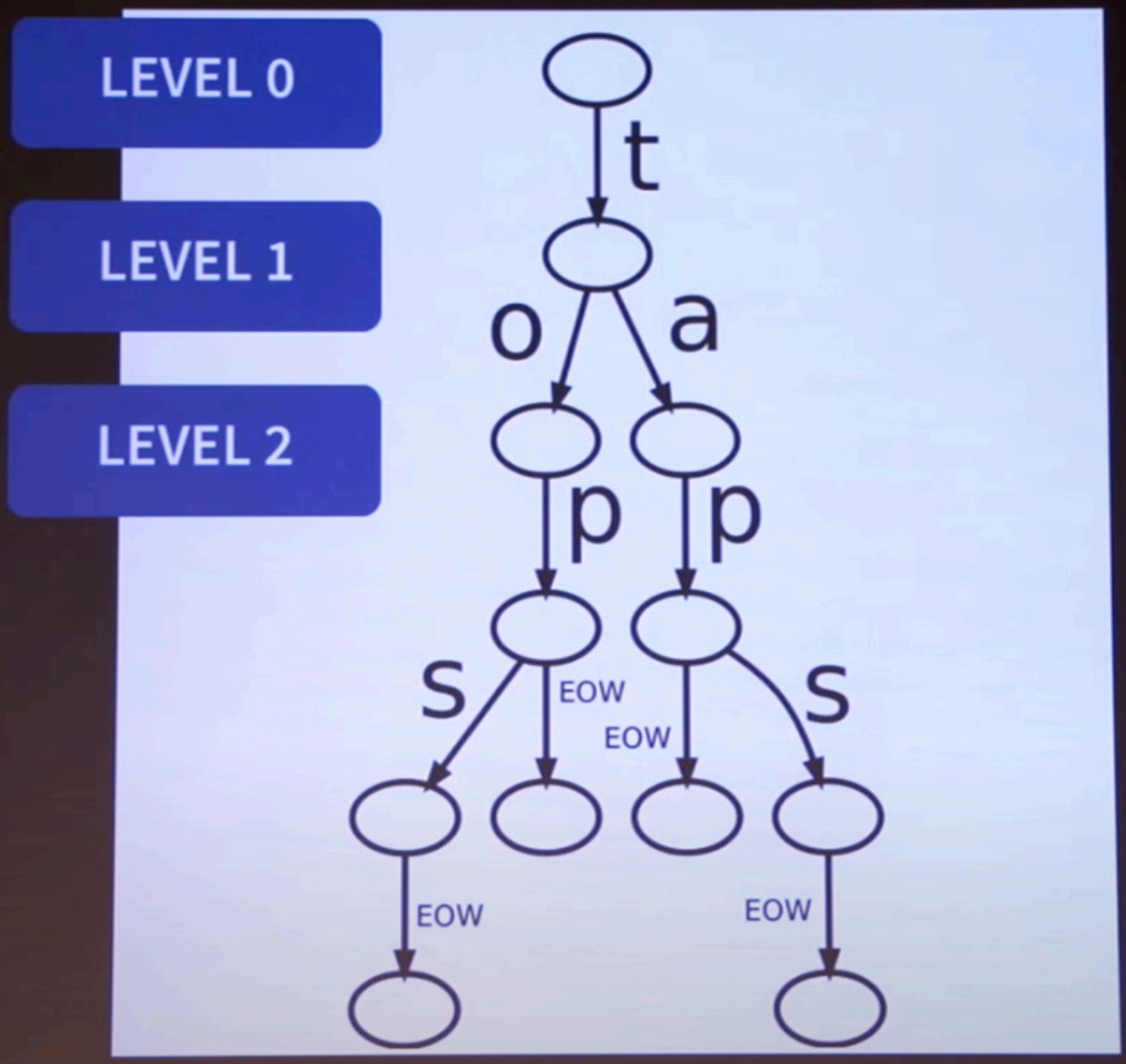
Persistent Data Structures

In Clojure Language

A Better Presentation On Clojure Will be made after Exam

- Composite values - immutable
- 'Change' is merely a function, takes one value and returns another, 'changed' value
- Collection maintains its performance guarantees
  - Therefore new versions are not full copies
- Old version of the collection is still available after 'changes', with same performance
- Example - hash map/set and vector based upon array mapped hash tries (Bagwell)

A Better Presentation On Clojure Will be made after Exam



00000 01011 01001 11001 10111 00010 11100

0 11 9 25

ArrayNode
shift = 0

28

ArrayNode
shift = 5

2

23

ArrayNode
shift = 10

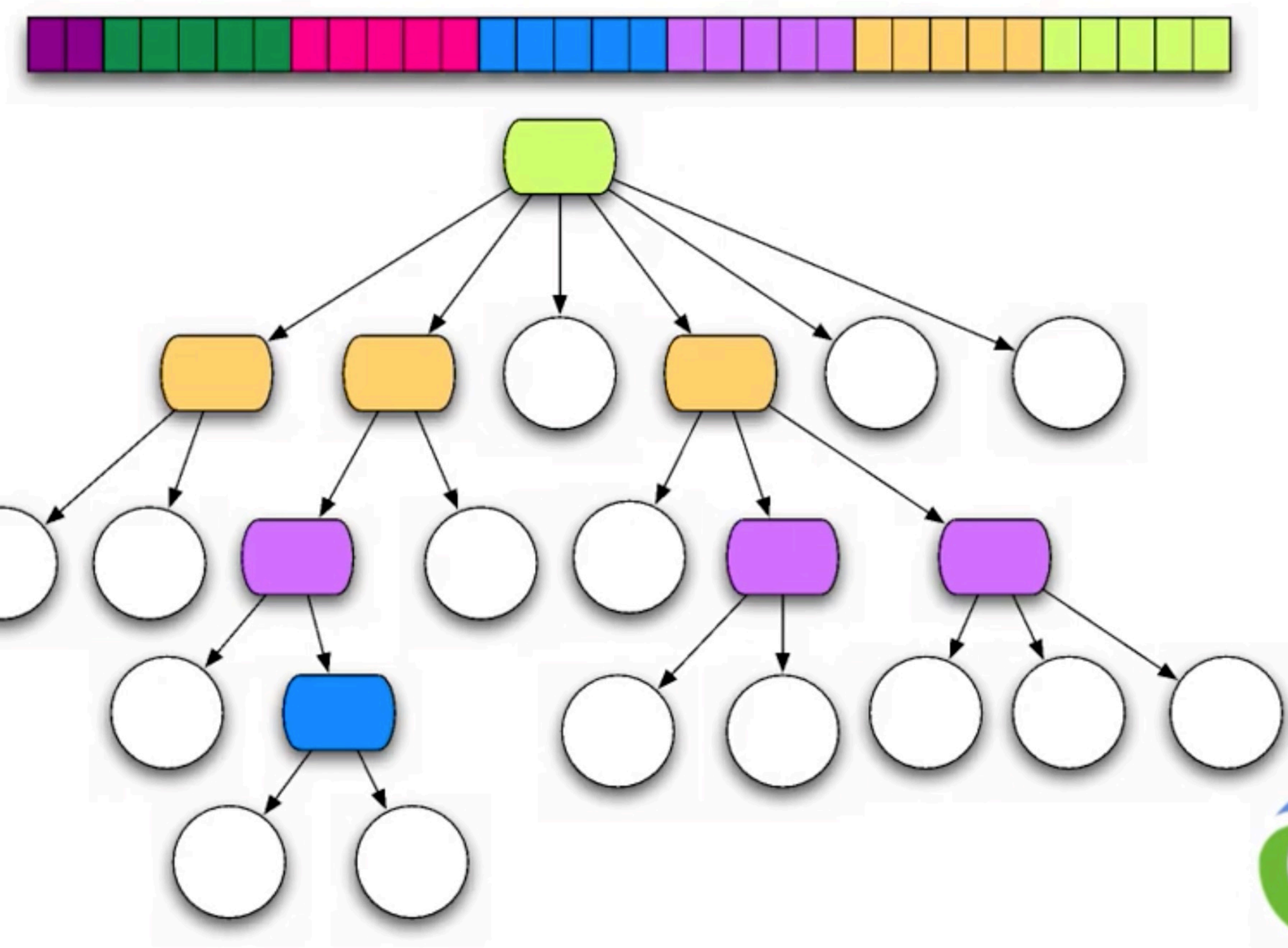
BitmapIndexedNode
shift = 15

A Better Presentation On Clojure Will be made after Exam

… and then follow the AMT down

A Better Presentation On Clojure Will be made after Exam

Bit-partitioned hash tries



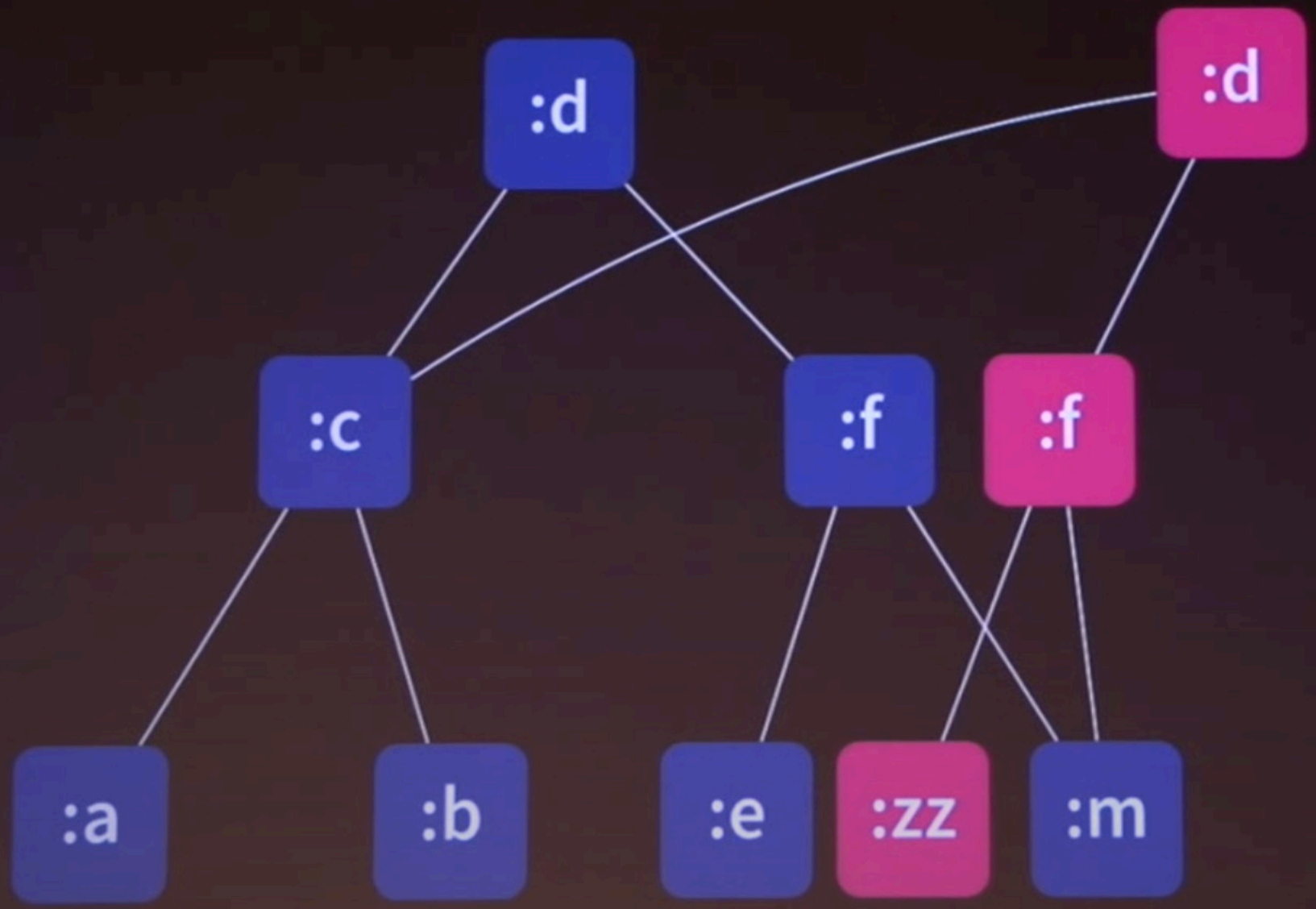
A Better Presentation On Clojure Will be made after Exam

Structural Sharing

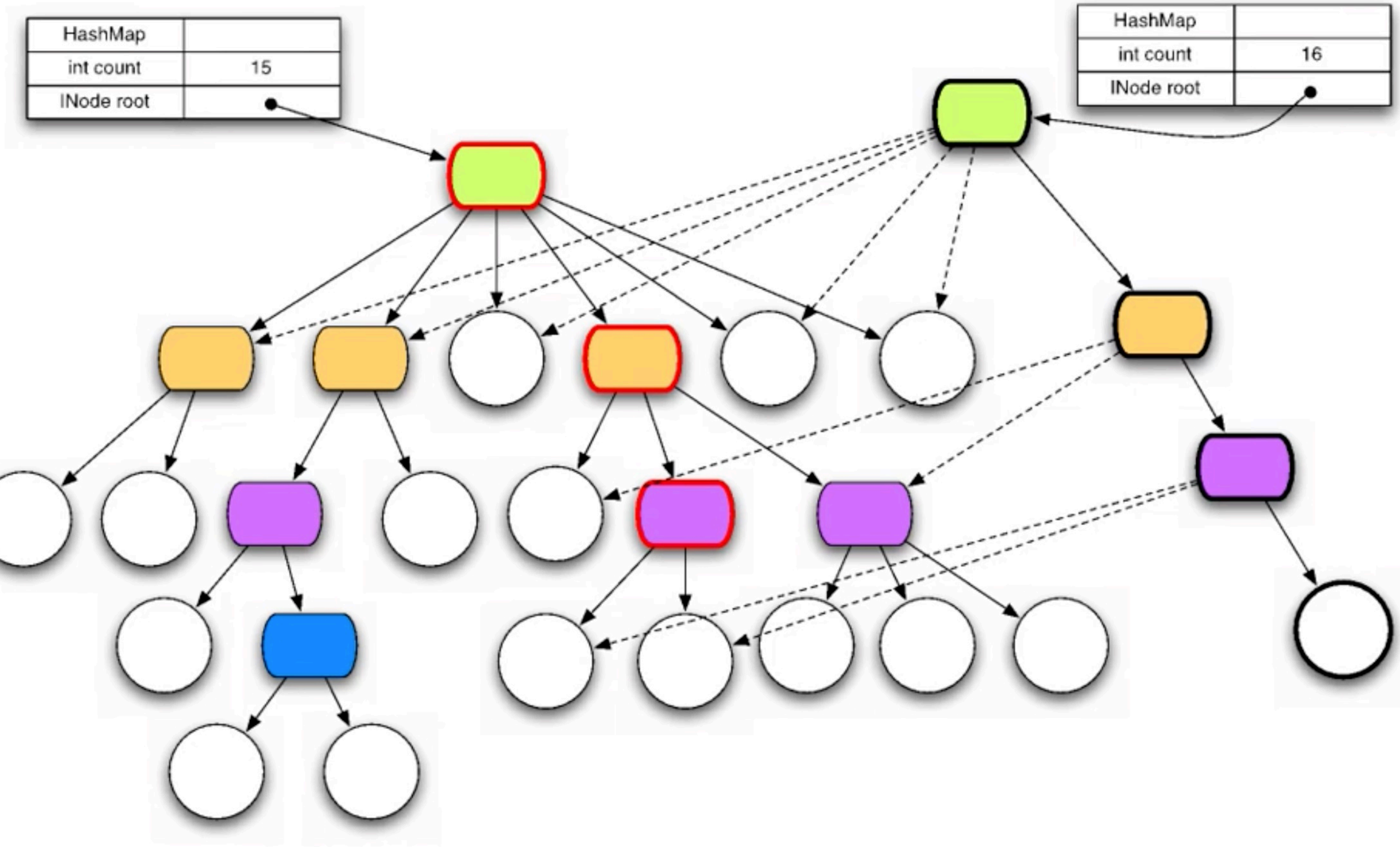
- Key to efficient 'copies' and therefore persistence
- Everything is immutable so no chance of interference
- Thread safe
- Iteration safe

STRUCTURAL SHARING

(assoc v 4 :zz)



Path Copying

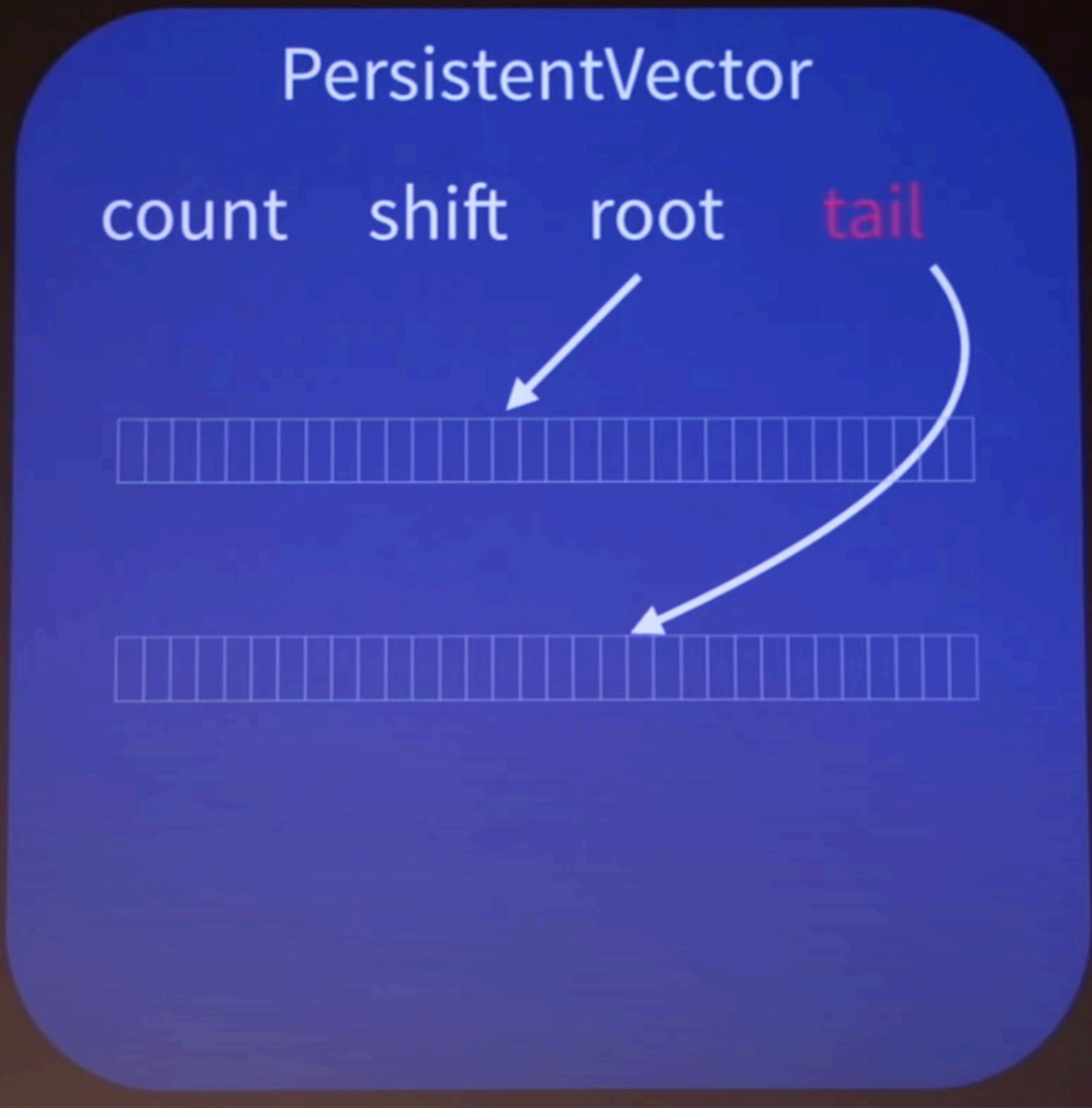


REGULAR HASH TABLE?

NEED ROOT RESIZING

NOT AMENABLE TO STRUCTURAL SHARING

THE TAIL OPTIMIZATION



Slides are taken from, the seminars:

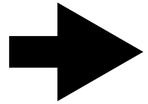
1. What Lies Beneath - A Deep Dive Into Clojure's Data Structures - Mohit Thatte
2. Persistent Data Structures and Managed References - Rich Hickey

Now lets come to Javascript

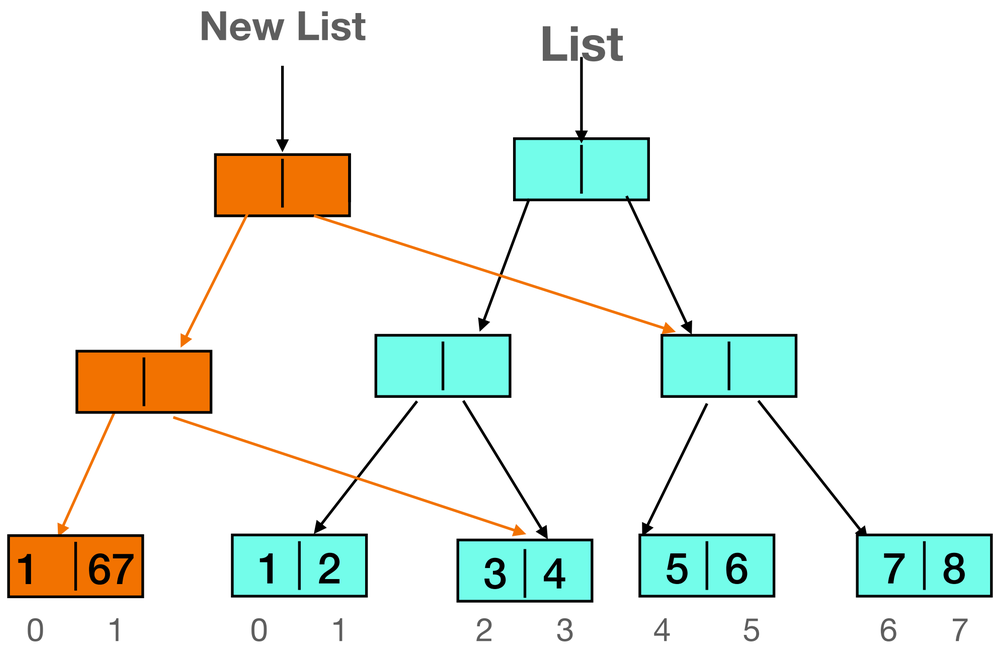
Suppose we want to make a list of some numbers

list=[1,2,3,4,5,6,7,8]

Now suppose we want to change the number at index 1 i.e. 2 to 67.

Ok so now lets break our list of numbers in size of 2  1,2 3,4 5,6 7,8

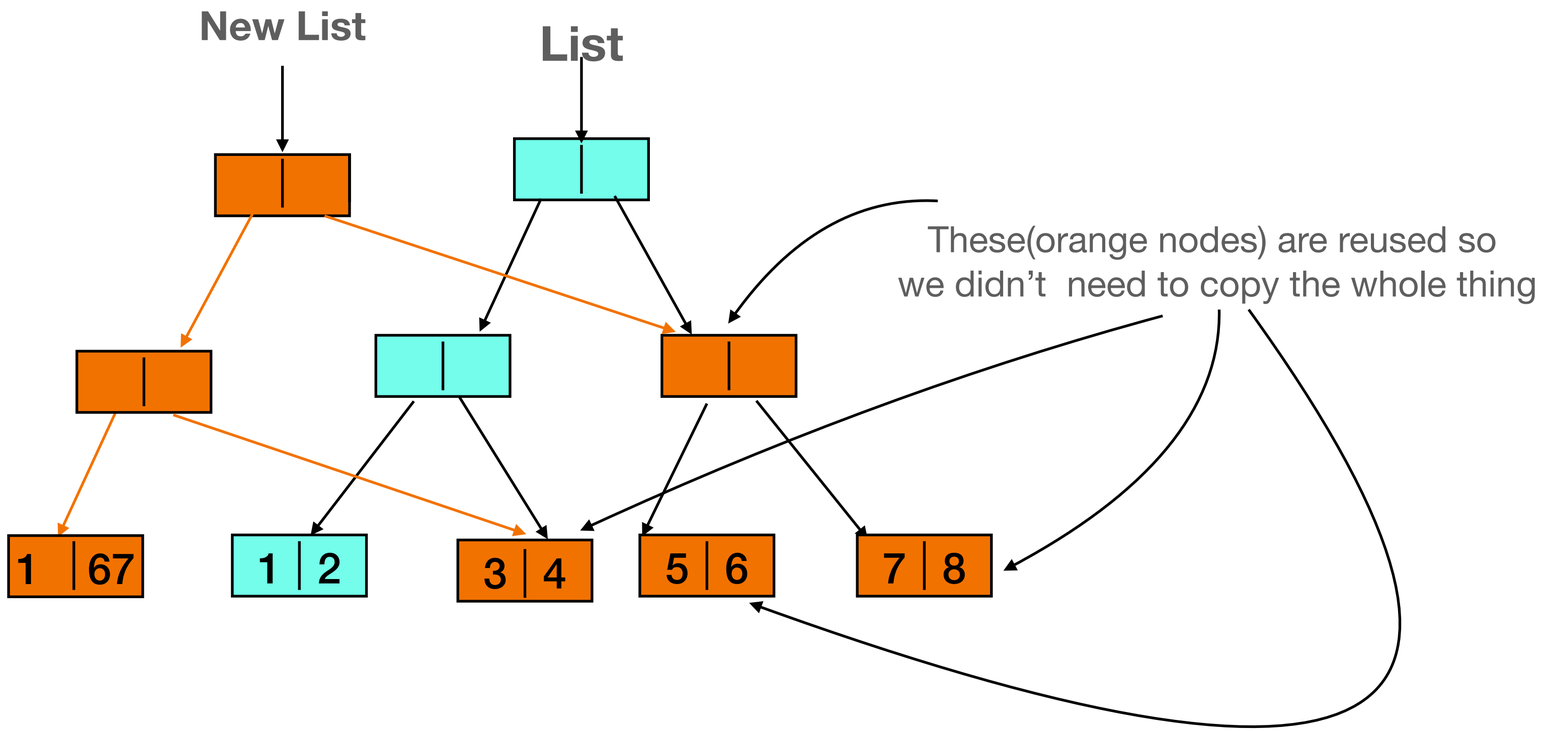
So now we will update the value 2 to 67



1st the node is copied

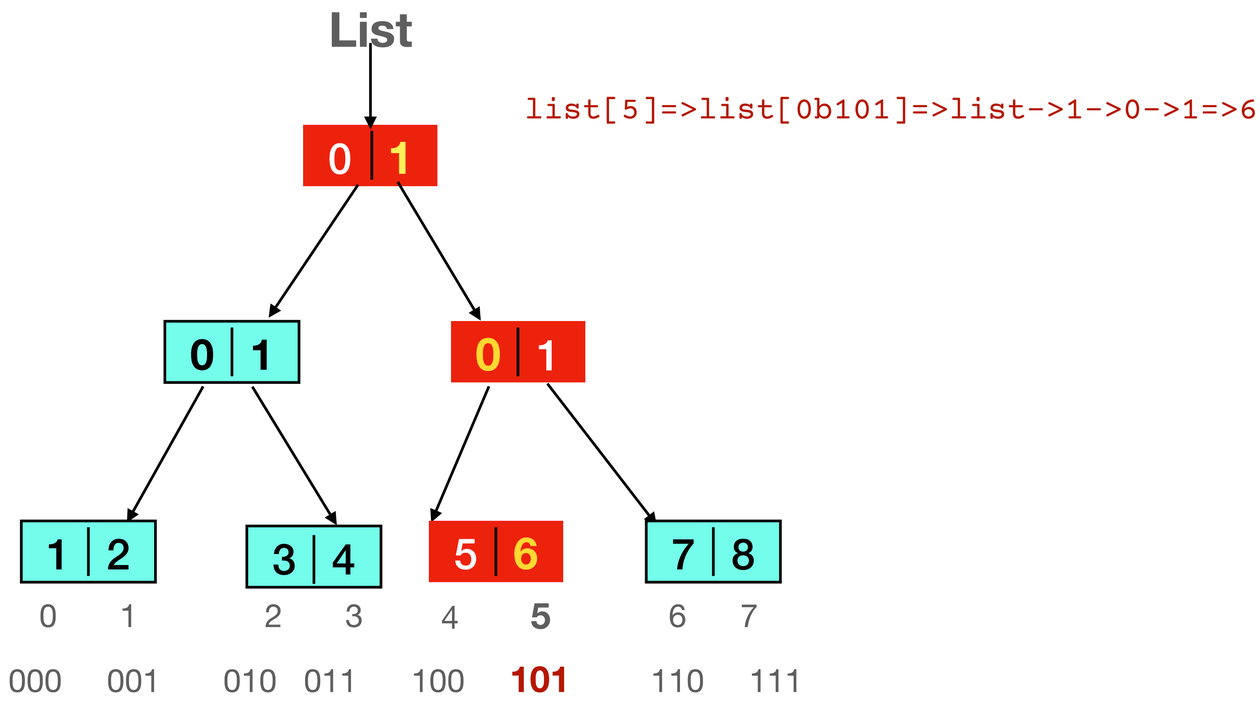
Then path copying happens

So now the list is



So now if we want to get a value at any index say 5
We can do that easily using hashing 5 to its binary value 0b101

We will go from 1 → 0 → 1 to get our value I.e. 5



In this way we can reach to any node quickly and return the value in a less time using hashing