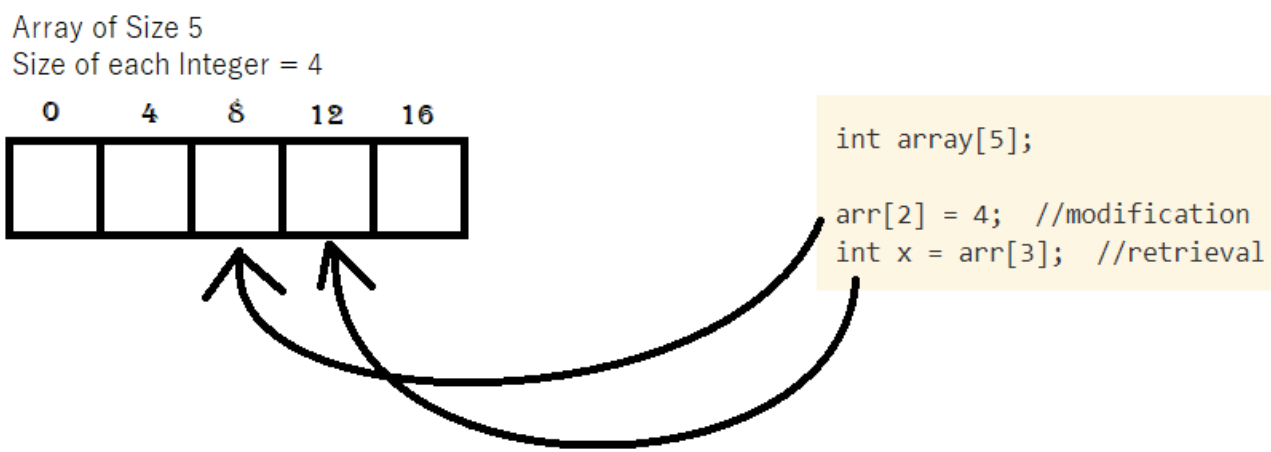
Persistent Array

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

Ephimeral Array

- Collection of items of same type stored in contiguous memory location
- An element from a particular position can be retrieved or modified directly by calculating address of a location
- Eg.



- Required address of a position in an array = starting address+ (required index * size of data type)

Operations on array

- modifyArray(a, i, e): store the value present in **e**, in the **i-th** element of **array a**

- retrieveArray(a, i) : return the value of the **i-th** element of **array a**

Persistent Data Structure

- Persistency can be achieved in a data structure, when we can preserve the previous version of itself even after modifying the data structure
- Persistency can be **partial:** older versions can be retrieved but cannot be modified**; and fully persistent:** older versions can be modified as well as read by the user
- Array can also be made persistent by few techniques

Method 1

COPY ON WRITE METHOD

Copy on Write method

- On updating an array, a new array is created (allocating memory followed by copying the data) and the corresponding update is made in the new array
- Operations:
  - modifyArray(a, v, i, e): store the value present in **e**, in the **i-th** element of **array a** of **version v**
  - retrieveArray(a, v, i) : return the value of the **i-th** element of **array a** of **version v**

Copy on Write method

- Fully Persistent and easy to implement
- Worst case time complexity:
  - Storing: O(n), n is number of elements in array.
  - Retrieval: O(1), when the different array versions are stored in an array (using array of pointers)
- Space Requirement: O(n*v), n is number of elements in array, v is number of versions (LIMITATION!)

Limitation and Solution

- Limitation: Same elements stored multiple times
- Solution: only the element which is changed, is stored in the array

Method 1

FAT NODE METHOD

Fat Node Method

- Target: Only the particular value of an element in an array is stored in the array, when it is changed.
- Approach: Each element in the persistent array will point to a **linked list,** which is storing the modified values at a particular position of an array, along with the versions in linear fashion
- Retrieval: on requesting the value of a particular version at a particular position, the linked list pointed to by the element of the array is traversed.

Comparison in Fat Node Method

Partial Persistent

- Modifying: version history is not required
- Since the older versions cant be modified, no version history is required
- On requesting a version, a value is chosen whose version is just less than or equal to the requested version

Fully Persistent

- Modifying : version history is required to keep record of ancestors
- Version history (stored in tree/ trie/array) is required to check the ancestor of a requested version
- On requesting a version, the version just less than the requested version, may not be the ancestor of requested version, so the version history must be checked.

Implementation

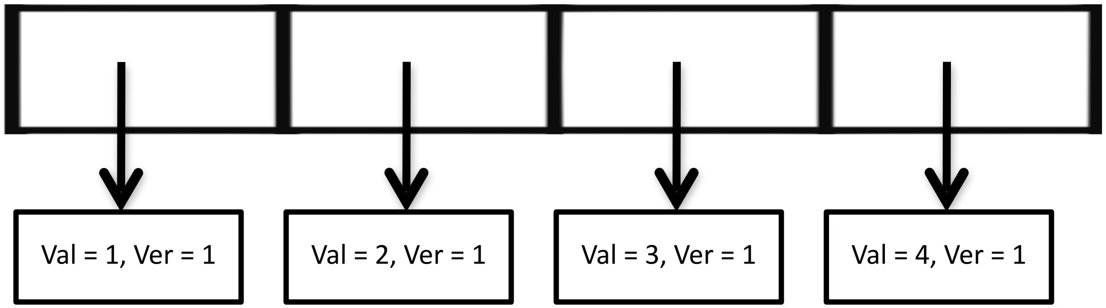
FULLY PERSISTENT ARRAY

Implementation

- Data Structure: Array of pointers, version history (in form of tree/array)
- Operations:
  - ModifyArray(A, V, P, N) = store value N at position P of version V of array A
  - Retrieve(A, V, P) = return value at position P of version V of array A
  - Clear(A) = remove all the previous versions of array A

Simulation

Query: Init(**version** = 0, **arr**{1,2,3,4}, **size** = 4}



Version Track
(for simulation purpose)

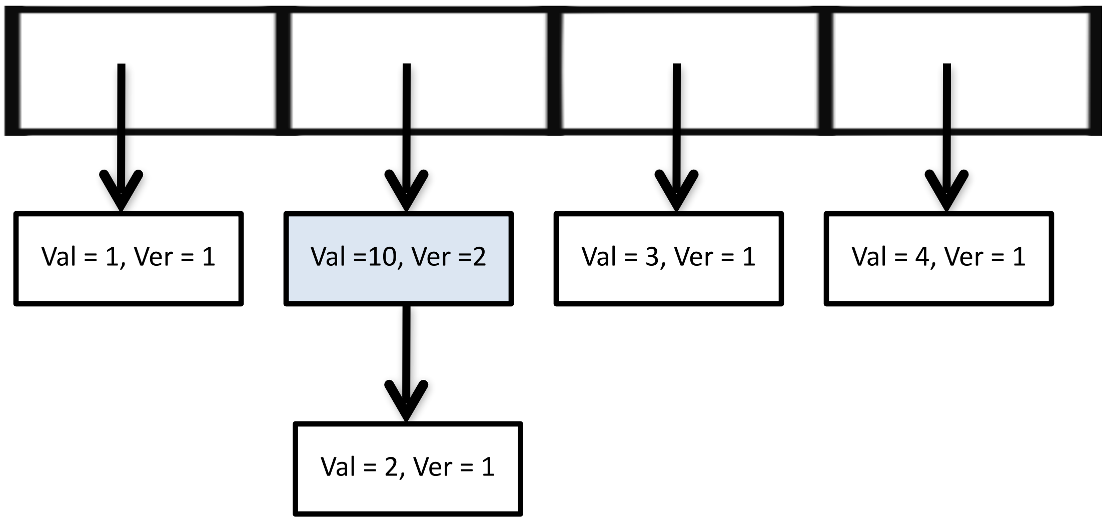
1 2 3 4

Version History

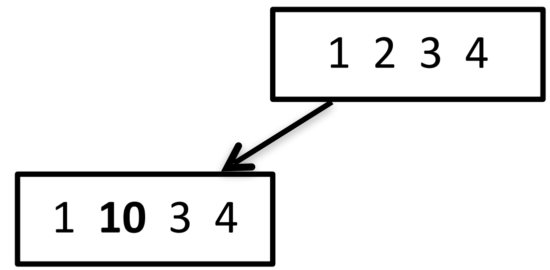
0	1	2	3	4	5	6	7	8	9	10
	0									

Simulation

Query: ModifyArray(**version** = 1, **position** = 1, **value** = 10)



Version Track
(for simulation purpose)

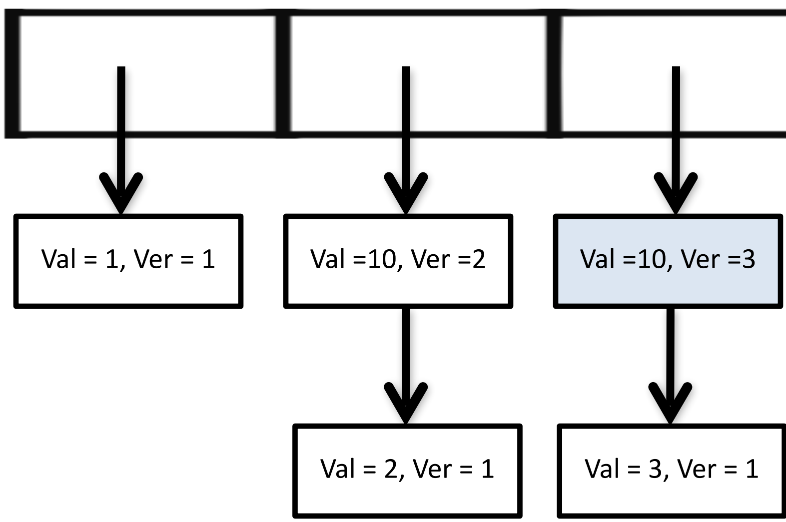


Version 2 created,
from version 1

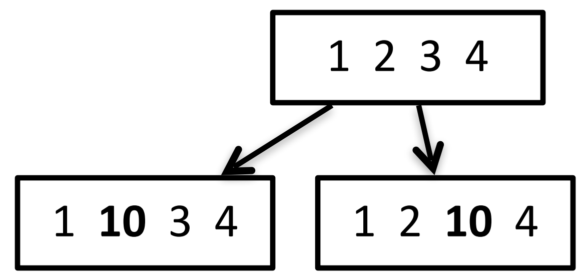
	0	1	2	3	4	5	6	7	8	9	10
Version History		0	1								

Simulation

Query: ModifyArray(**version** = 1, **position** = 2, **value** = 10)



Version Track
(for simulation purpose)

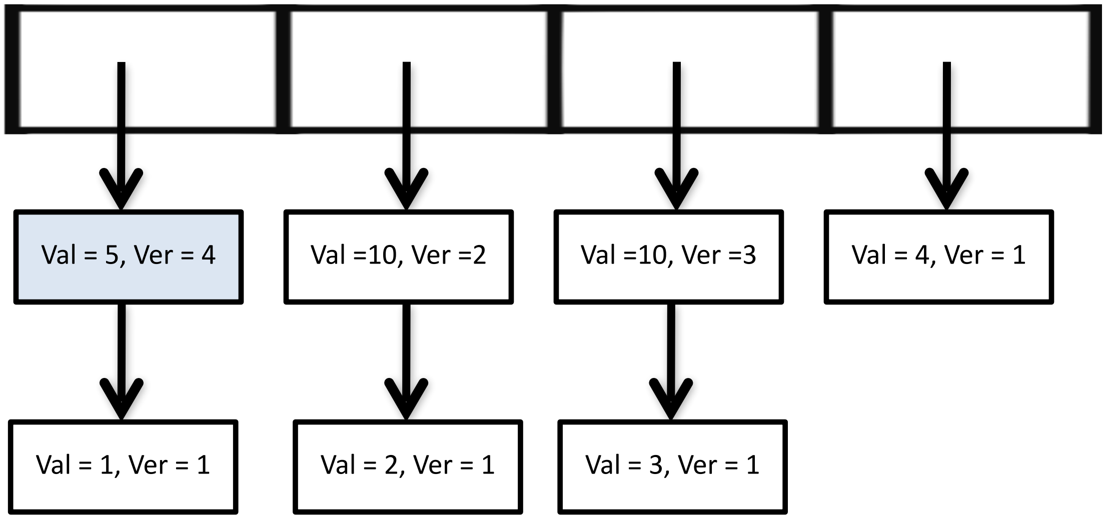


Version 3 created,
from version 1

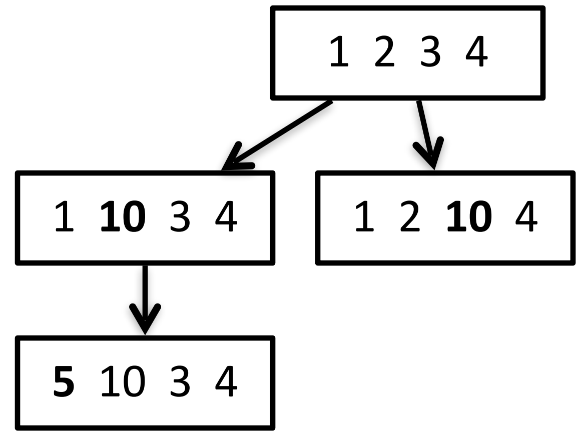
	0	1	2	3	4	5	6	7	8	9	10
Version History		0	1	1							

Simulation

Query: ModifyArray(**version** = 2, **position** = 0, **value** = 5)



Version Track
(for simulation purpose)

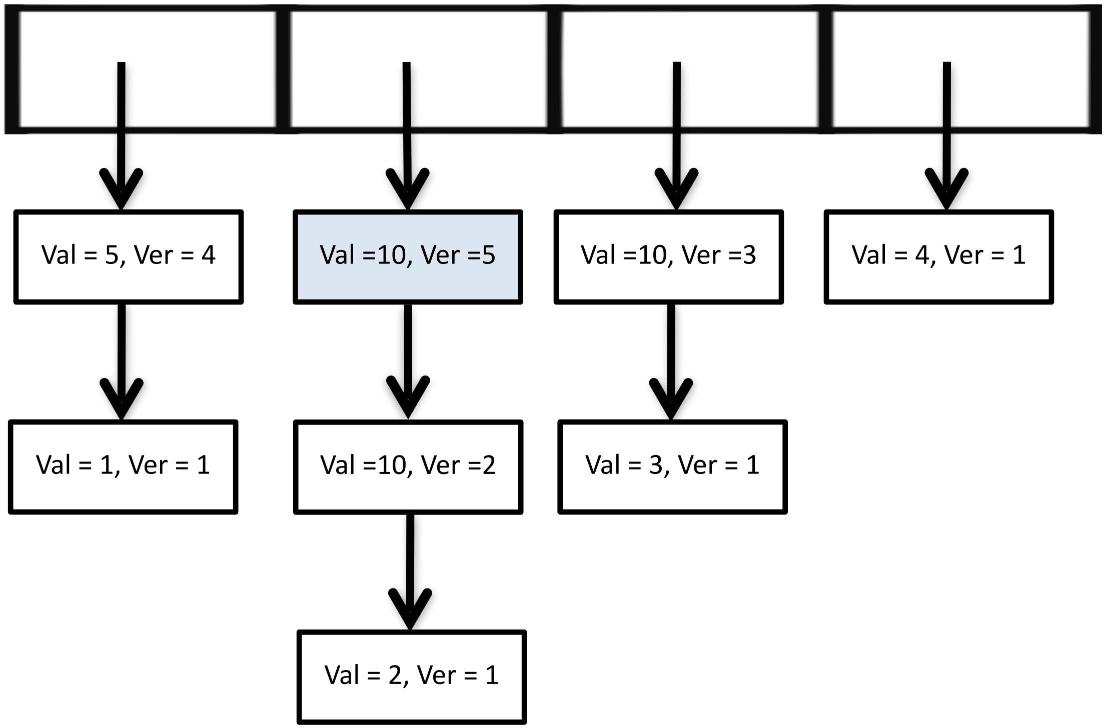


Version 4 created, from version 2

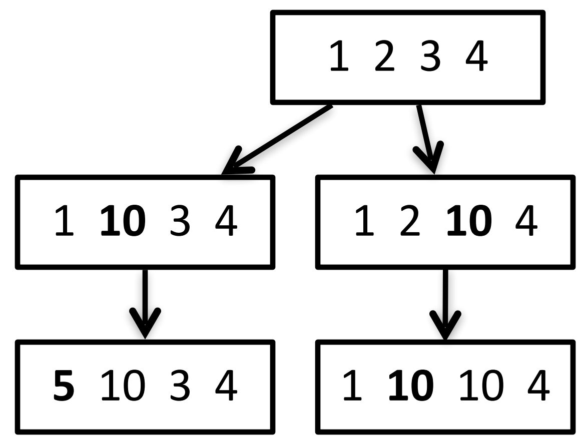
	0	1	2	3	4	5	6	7	8	9	10
Version History		0	1	1	2						

Simulation

Query: ModifyArray(**version** = 3, **position** = 1, **value** = 10)



Version Track
(for simulation purpose)

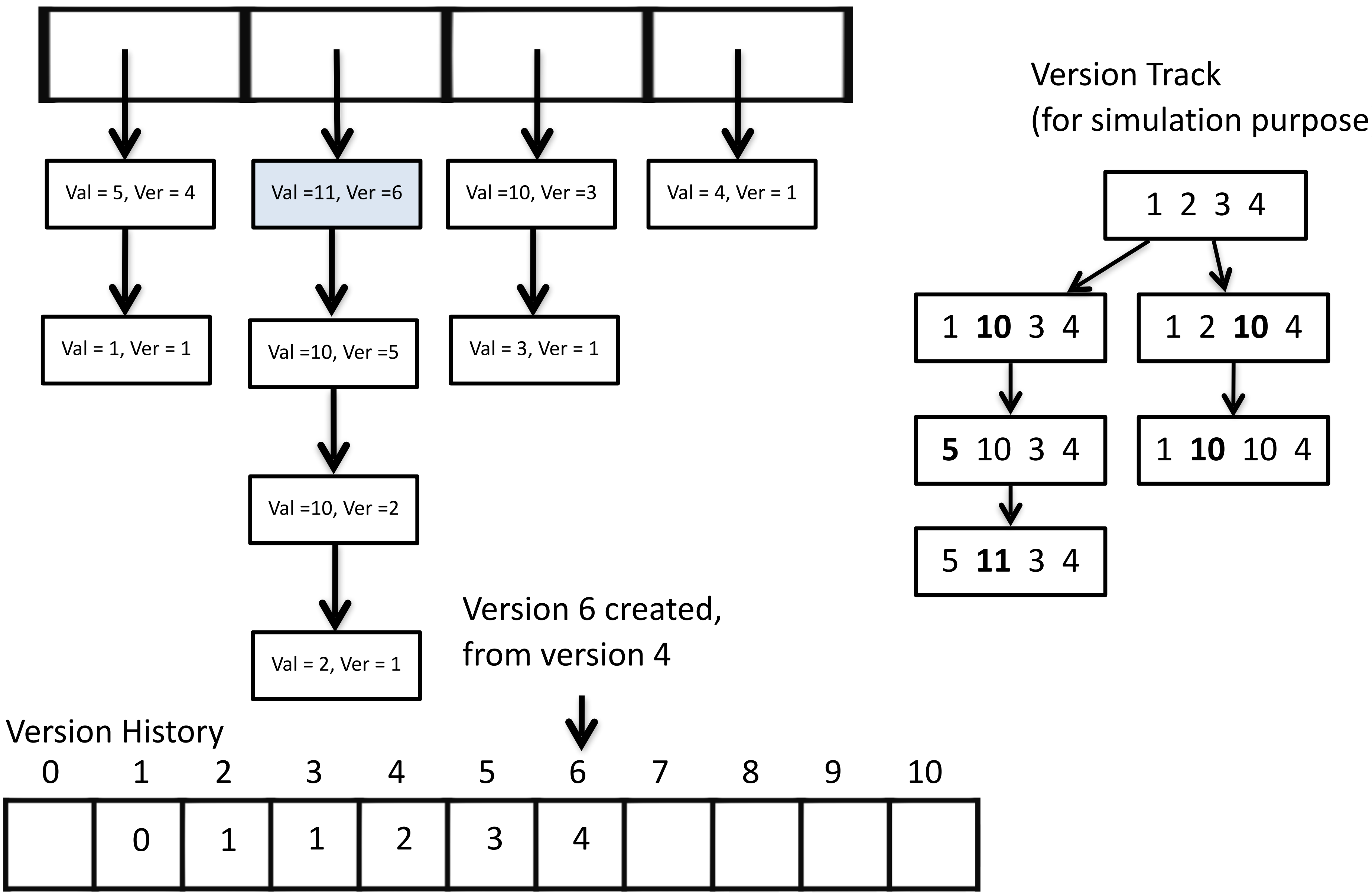


Version 5 created,
from version 3

	0	1	2	3	4	5	6	7	8	9	10
Version History		0	1	1	2	3					

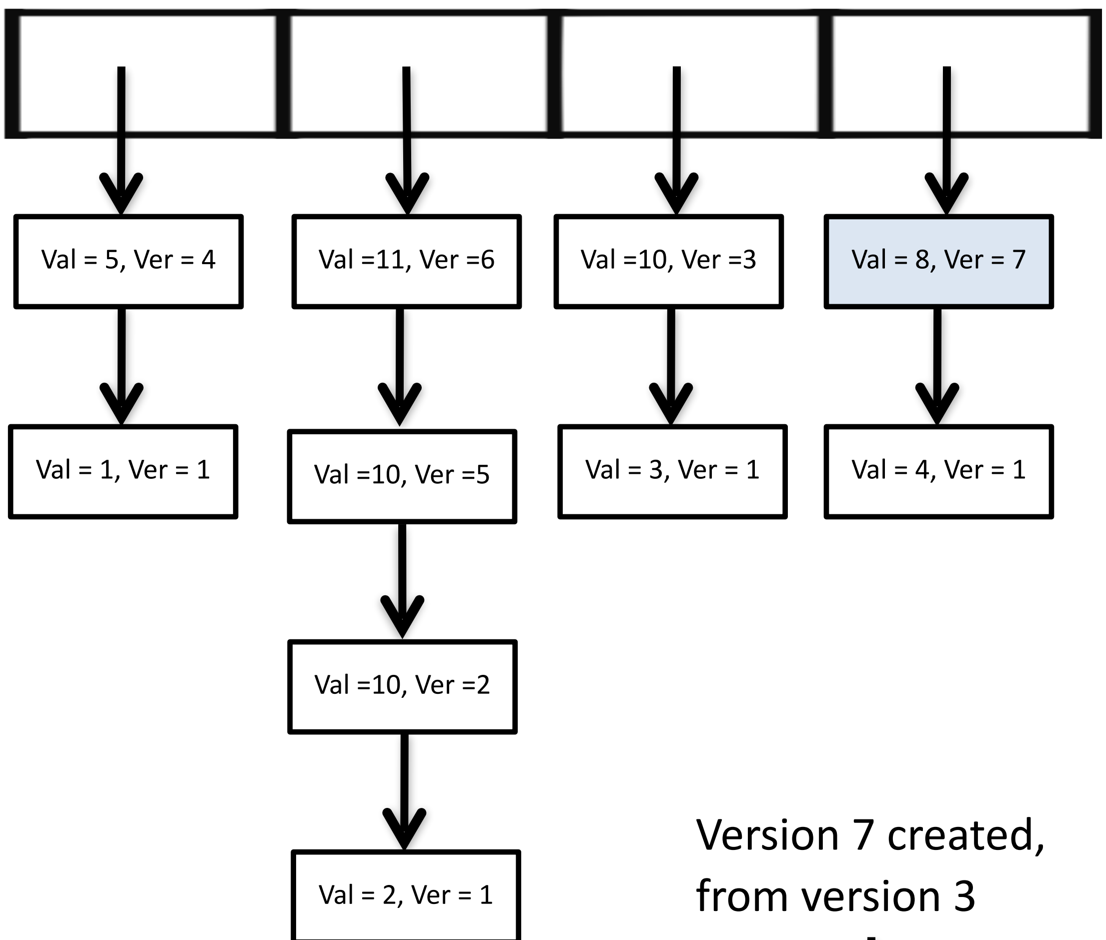
Simulation

Query: ModifyArray(**version** = 4, **position** = 1, **value** = 11)

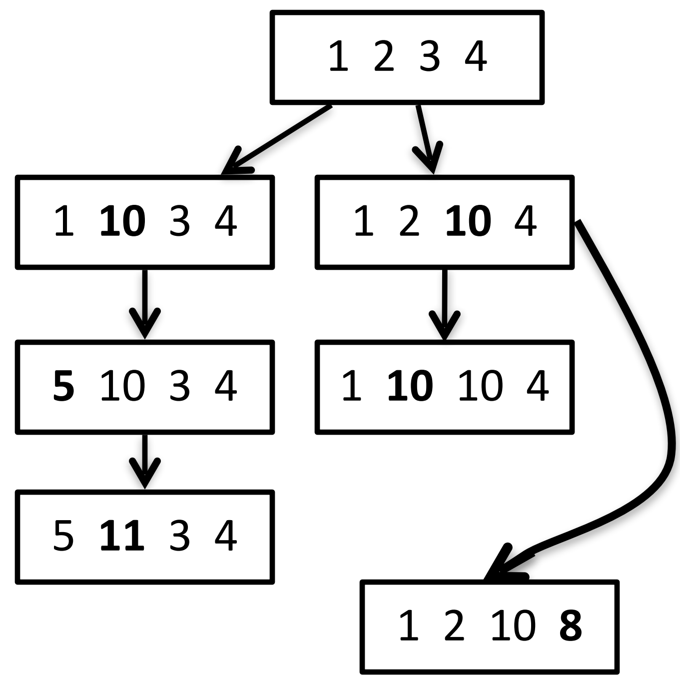


Simulation

Query: ModifyArray(**version** = 3, **position** = 3, **value** = 8)



Version Track
(for simulation purpose)



Version 7 created,
from version 3

Version History

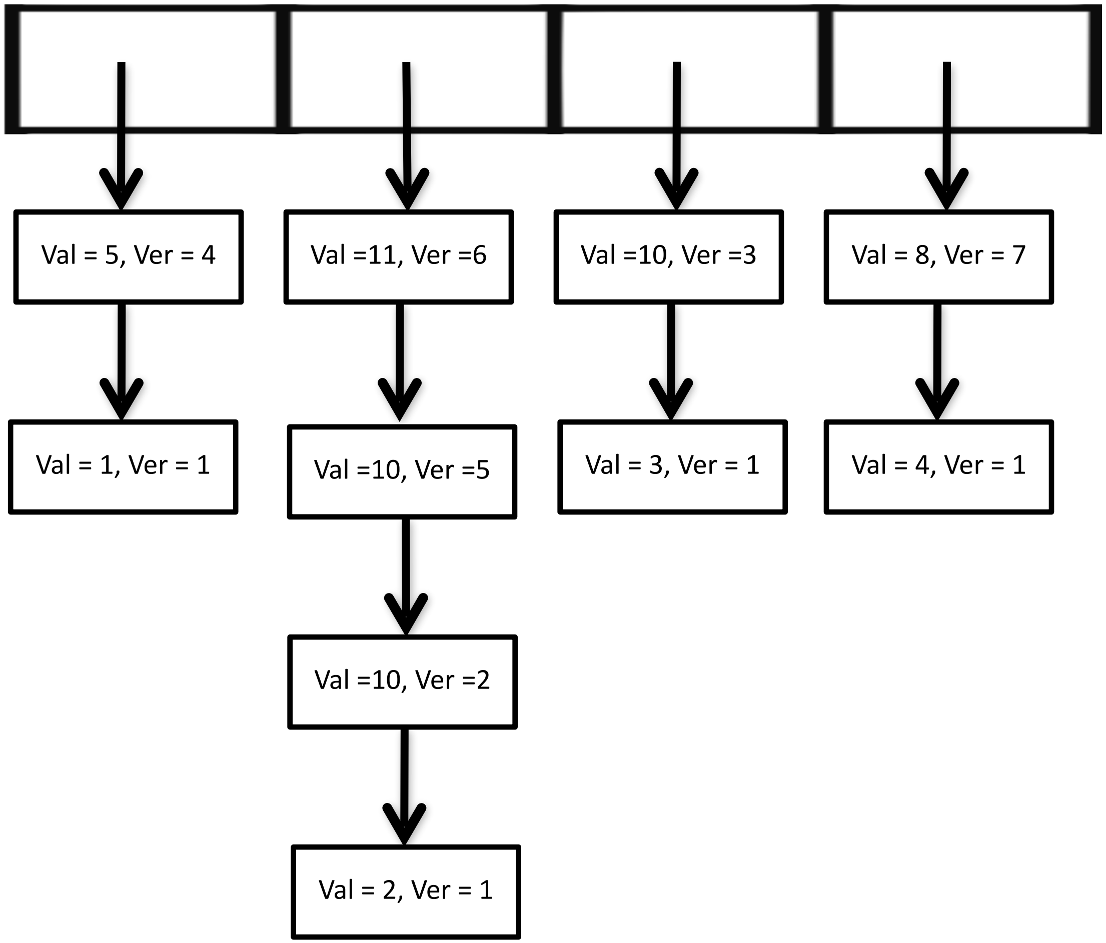
0	1	2	3	4	5	6	7	8	9	10
	0	1	1	2	3	4	3			

Retrieval algorithm

1. Start at the node pointed to by pointer at required position
2. WHILE (required version ≠ node version)
   1. IF (req version < node version) traverse to the next node
   2. ELSEIF (required version > node version) look into the version history to find the immediate ancestor
   3. ELSE return value at the node
   4. ENDIF
3. ENDWHILE

Simulation

Query: Retrieve(**version** = 7, **position** = 1)

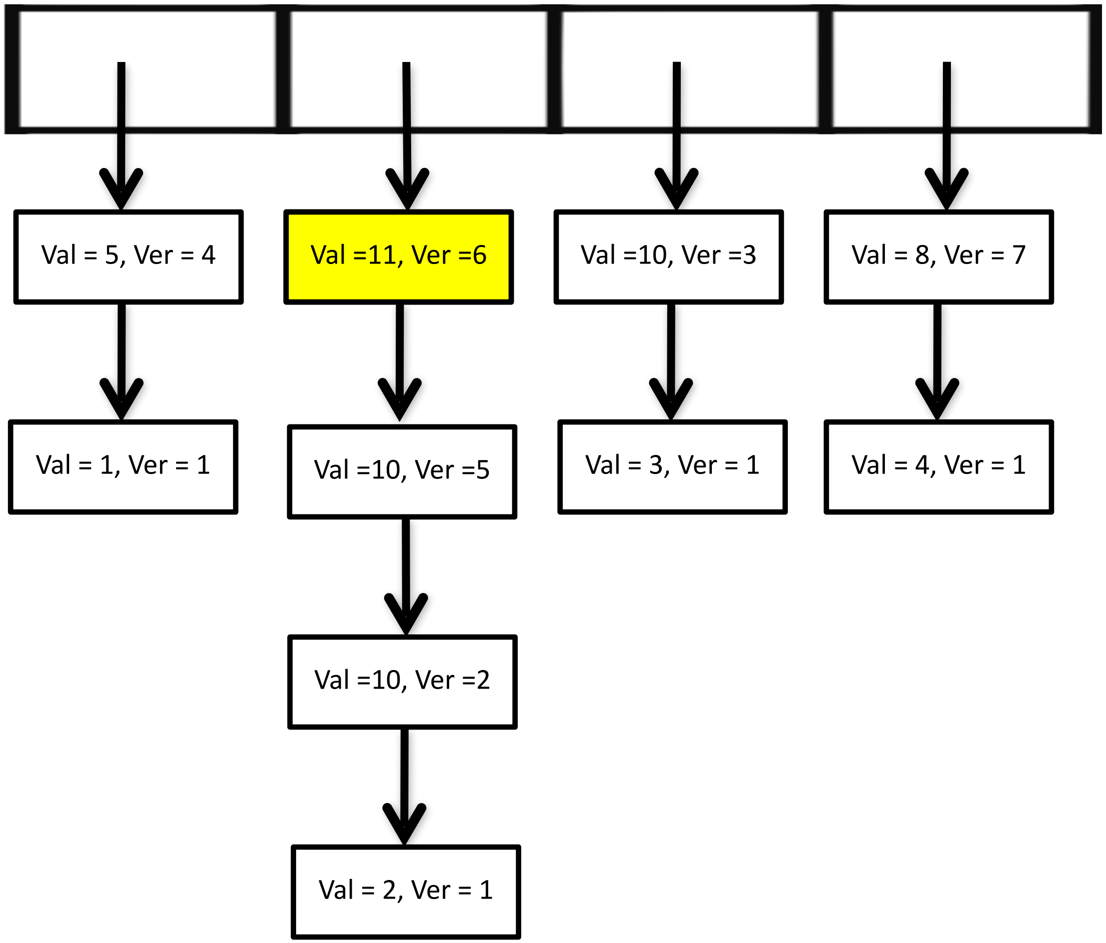


Version History

0	1	2	3	4	5	6	7	8	9	10
	0	1	1	2	3	4	3			

Simulation

Query: Retrieve(**version** = 7, **position** = 1)



Start at Node from Position 2

Version History

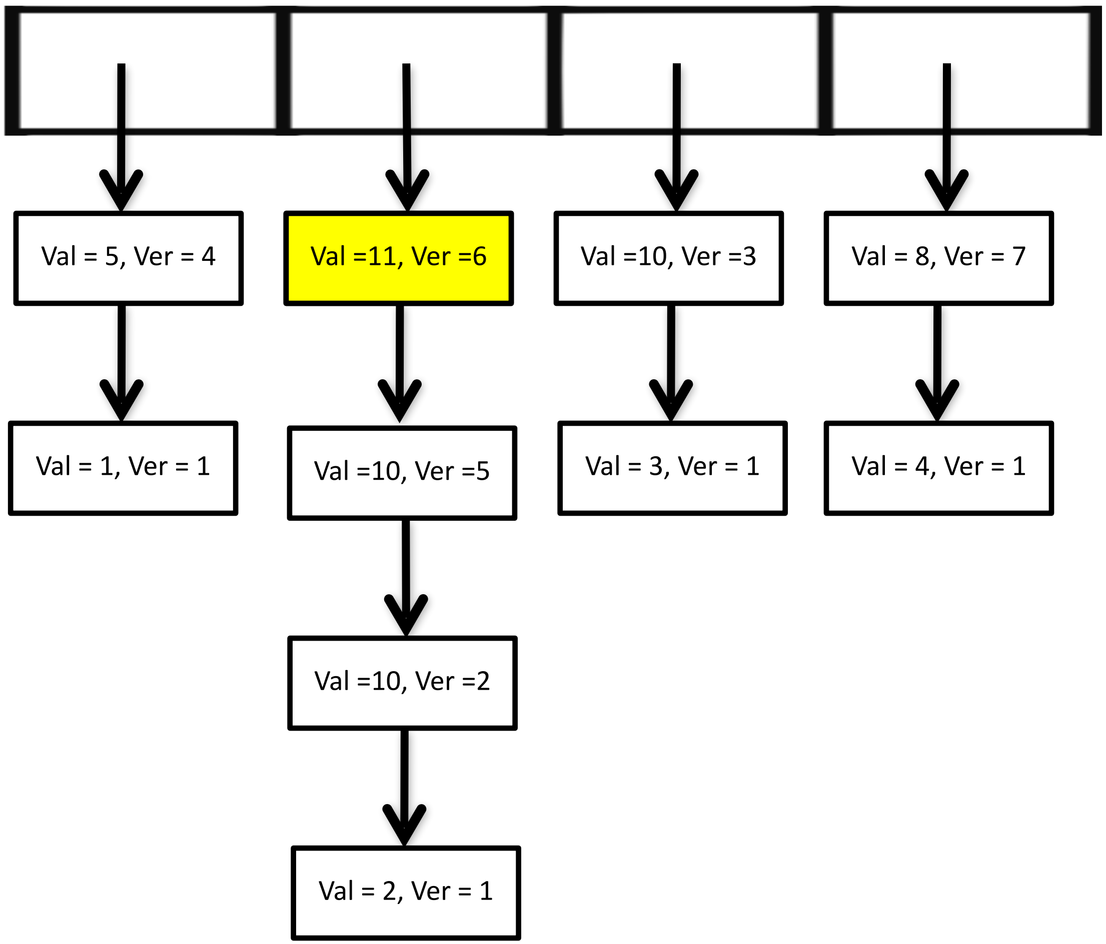
0	1	2	3	4	5	6	7	8	9	10
	0	1	1	2	3	4	3			

Simulation

Query: Retrieve(**version** = 7, **position** = 1)

Start at Node from Position 2

Node Version ≠ required version

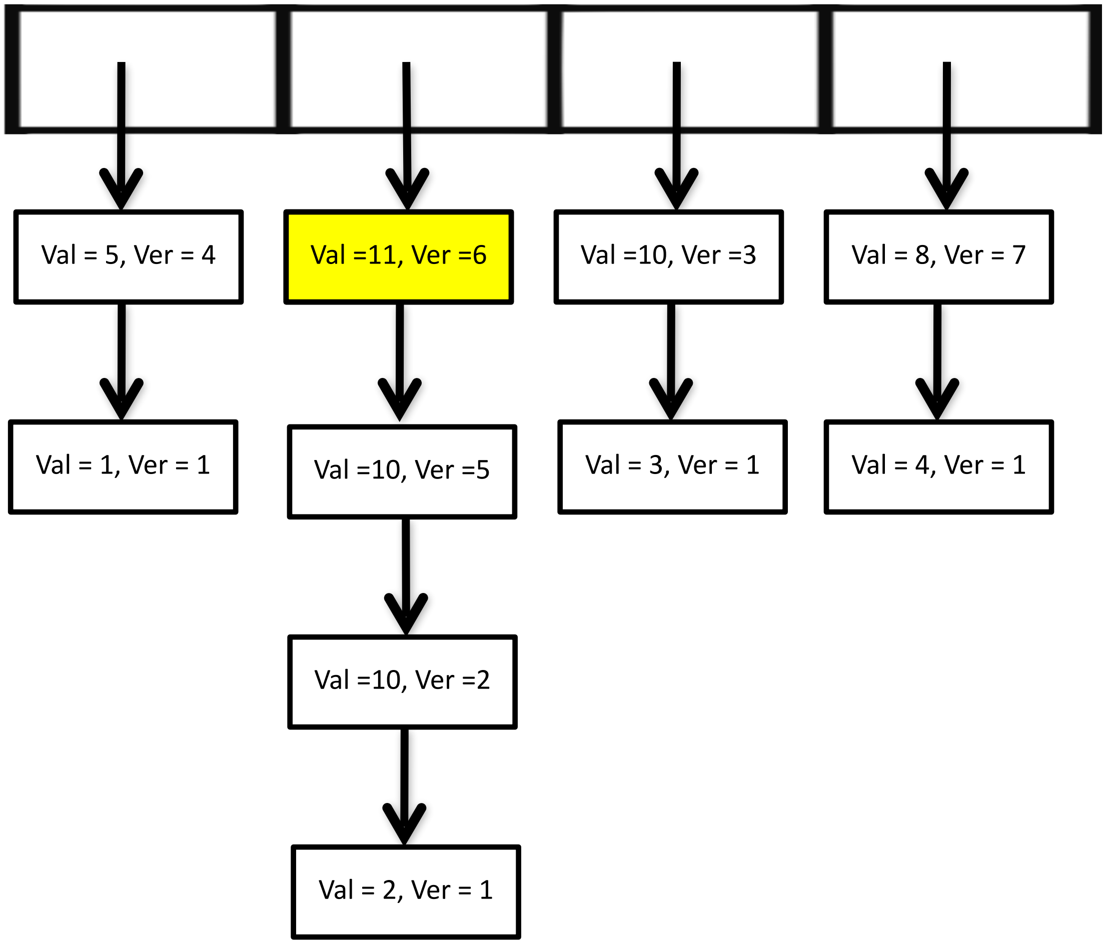


Version History

0	1	2	3	4	5	6	7	8	9	10
	0	1	1	2	3	4	3			

Simulation

Query: Retrieve(**version** = 7, **position** = 1)



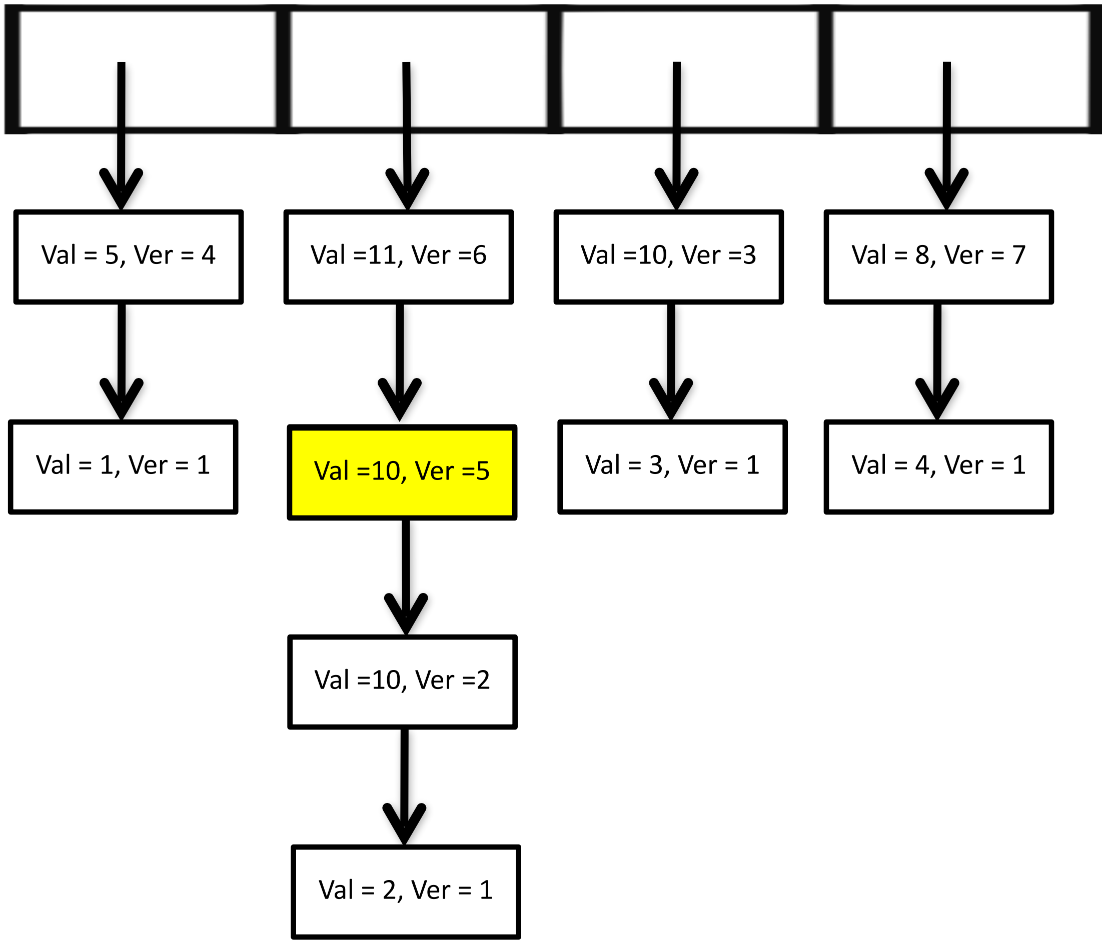
Start at Node from Position 2
Node Version ≠ required version
Node Version < required version,
so, immediate ancestor, 3
is required version now

Version History

0	1	2	3	4	5	6	7	8	9	10
	0	1	1	2	3	4	3			

Simulation

Query: Retrieve(**version** = 7, **position** = 1)



Start at Node from Position 2

Node Version ≠ required version

Node Version < required version,
so, immediate ancestor, 3
is required version now

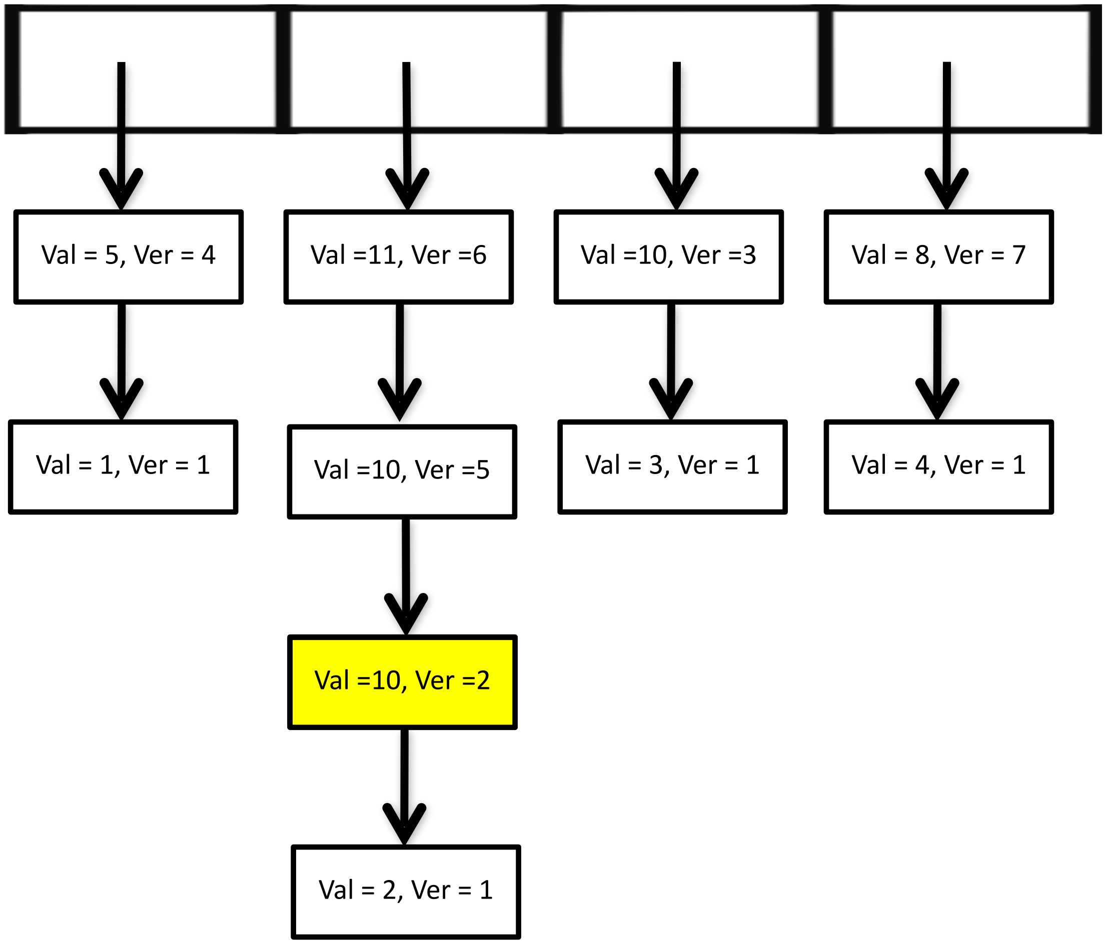
Node Version > required version,
so, go to next node

Version History

0	1	2	3	4	5	6	7	8	9	10
	0	1	1	2	3	4	3			

Simulation

Query: Retrieve(**version** = 7, **position** = 1)



Start at Node from Position 2

Node Version ≠ required version

Node Version < required version,
so, immediate ancestor, 3
is required version now

Node Version > required version,
so, go to next node

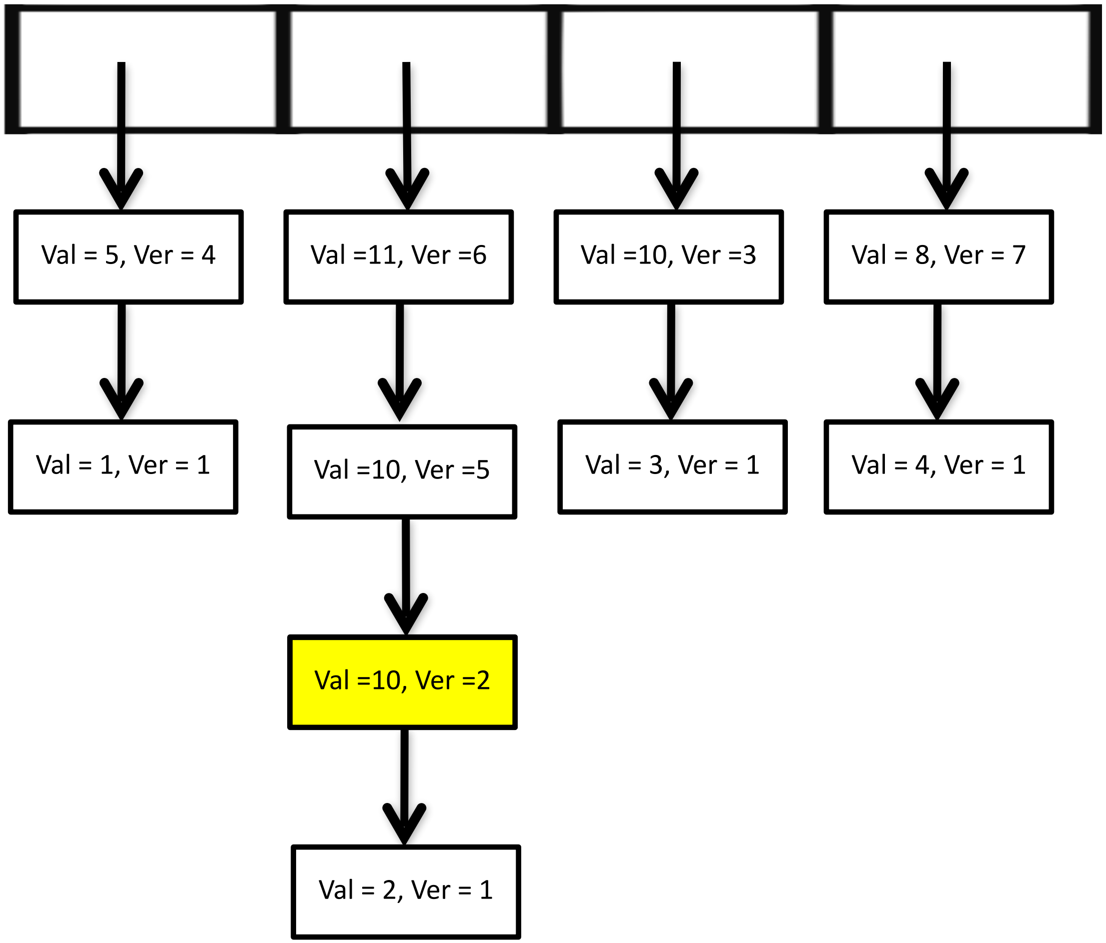
Node Version > required version,
so, go to next node

Version History

0	1	2	3	4	5	6	7	8	9	10
	0	1	1	2	3	4	3			

Simulation

Query: Retrieve(**version** = 7, **position** = 1)



Start at Node from Position 2

Node Version ≠ required version

Node Version < required version, so, immediate ancestor, 3 is required version now

Node Version > required version, so, go to next node

Node Version > required version, so, go to next node

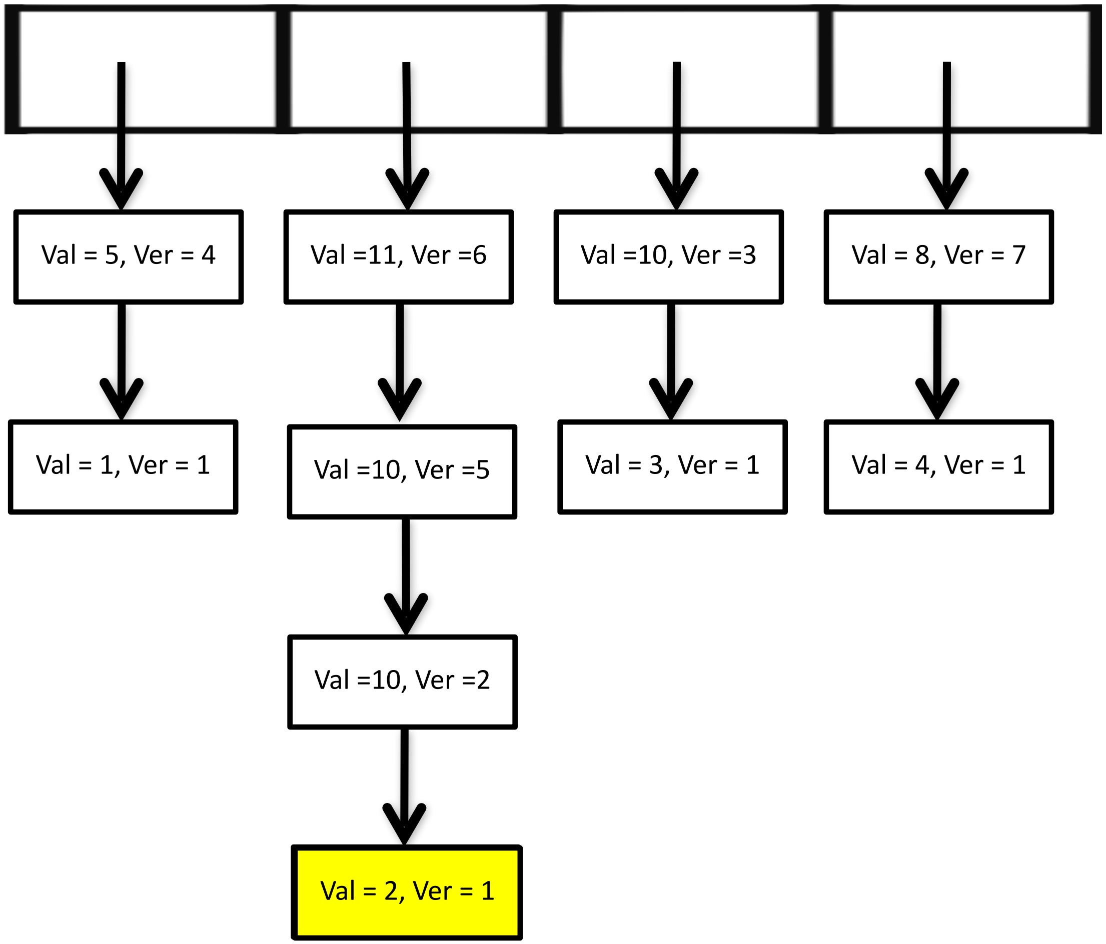
Node Version < required version, so, immediate ancestor, 1 is required version now

Version History

0	1	2	3	4	5	6	7	8	9	10
	0	1	1	2	3	4	3			

Simulation

Query: Retrieve(**version** = 7, **position** = 1)



Start at Node from Position 2

Node Version ≠ required version

Node Version < required version, so, immediate ancestor, 3 is required version now

Node Version > required version, so, go to next node

Node Version > required version, so, go to next node

Node Version < required version, so, immediate ancestor, 1 is required version now

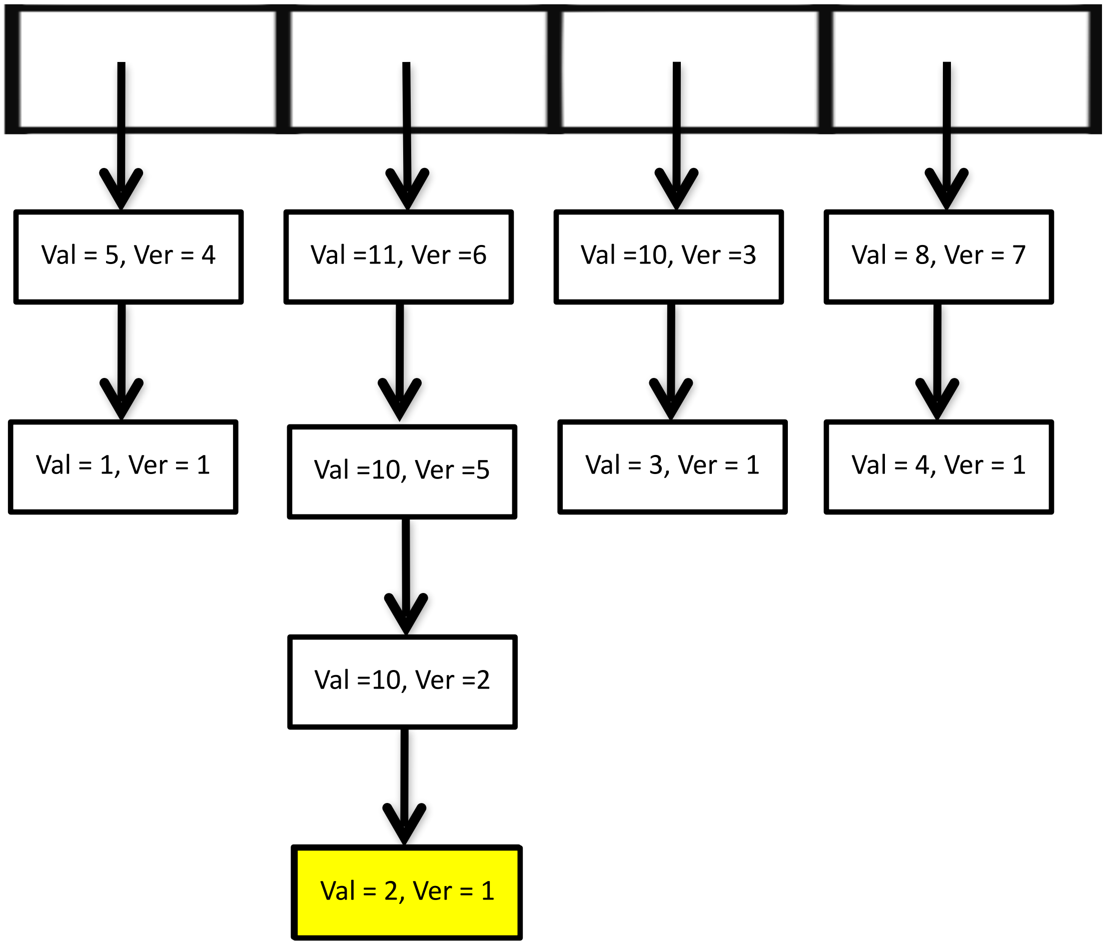
Node Version > required version, so, go to next node

Version History

0	1	2	3	4	5	6	7	8	9	10
	0	1	1	2	3	4	3			

Simulation

Query: Retrieve(**version** = 7, **position** = 1)



Start at Node from Position 2

Node Version ≠ required version

Node Version < required version,
so, immediate ancestor, 3
is required version now

Node Version > required version,
so, go to next node

Node Version > required version,
so, go to next node

Node Version < required version,
so, immediate ancestor, 1
is required version now

Node Version > required version,
so, go to next node

Version History

0	1	2	3	4	5	6	7	8	9	10
	0	1	1	2	3	4	3			

Node Version = Required version, SO Val = 2 returned

Analysis

- Time Complexity:
  - Update: O(1)
  - Worst Case Retrieve: O(V), V = number of versions created

- Space Complexity: O(n), where n = number of modifications, considering the pointer memory overhead and version history storage

Average Time Complexity (Modify)

- For n number of iterations, the time required is proportional to n
- So it can be concluded that average time complexity for single modification of a value at a certain position of persistent array ≈ O(1)

Average Time Complexity (Retrieve)

- m = number of modification at the required node, V = total number of versions created
- ∑m = V , total count of all the modifications in all the elements of array is total number of versions
- Average time complexity can be found by retrieving different versions of the array , and then taking average

$$\begin{pmatrix} \textit{Average time Complexity} \\ \textit{to iterate through whole array} \end{pmatrix} = \frac{\sum \begin{pmatrix} \textit{complexity to iterate through} \\ \textit{whole array for all versions} \end{pmatrix}}{\textit{Number of versions}} = \frac{\sum O(m+v)}{v}$$

$$= \frac{\max\_size * O(\sum m + \sum v)}{v} = \frac{\max\_size * O(V) + O(\sum v)}{v} = \max\_size * \left(O(1) + \frac{O(\sum v)}{v} \right)$$

$$= \max\_size \left(\frac{O(\sum v)}{v} \right)$$

Average Time Complexity (Retrieve)

- Hence, average time to retrieve a particular element of a particular version is $= \left(O(1) + \frac{O(\sum v)}{v} \right)$
- The term $\frac{O(\sum v)}{v}$ can have different time complexity **depending in the version history**
- **Best case**: All versions are created from a single ancestor, then

$$\left(\frac{O(\sum v)}{v} \right) = O\left(\frac{\sum log_v v}{v} \right) = O\left(\frac{\sum 1}{v} \right) = O\left(\frac{v}{v} \right) = O(1)$$

- Worst Case: All the versions are created from the corresponding latest version,

$$\left(\frac{O(\sum v)}{v}\right) = O\left(\frac{\frac{v(v+1)}{2}}{v}\right) = O\left(\frac{v+1}{2}\right) = O(v)$$

  - Reason: For the first array, the list will be traversed upto the end version, but for the latest version, only the 1st node will be checked, hence, ∑v = 1+2+3+.. +v = v(v+1)/2

- 2nd Worst Case: Let, the version history tree forms a binary balanced tree, hence the time complexity comes out to be

$$\frac{O(\sum v)}{v} = O\left(\frac{\sum log_2 v}{v}\right) = O\left(\frac{log_2 v!}{v}\right)$$

≃ **constant**

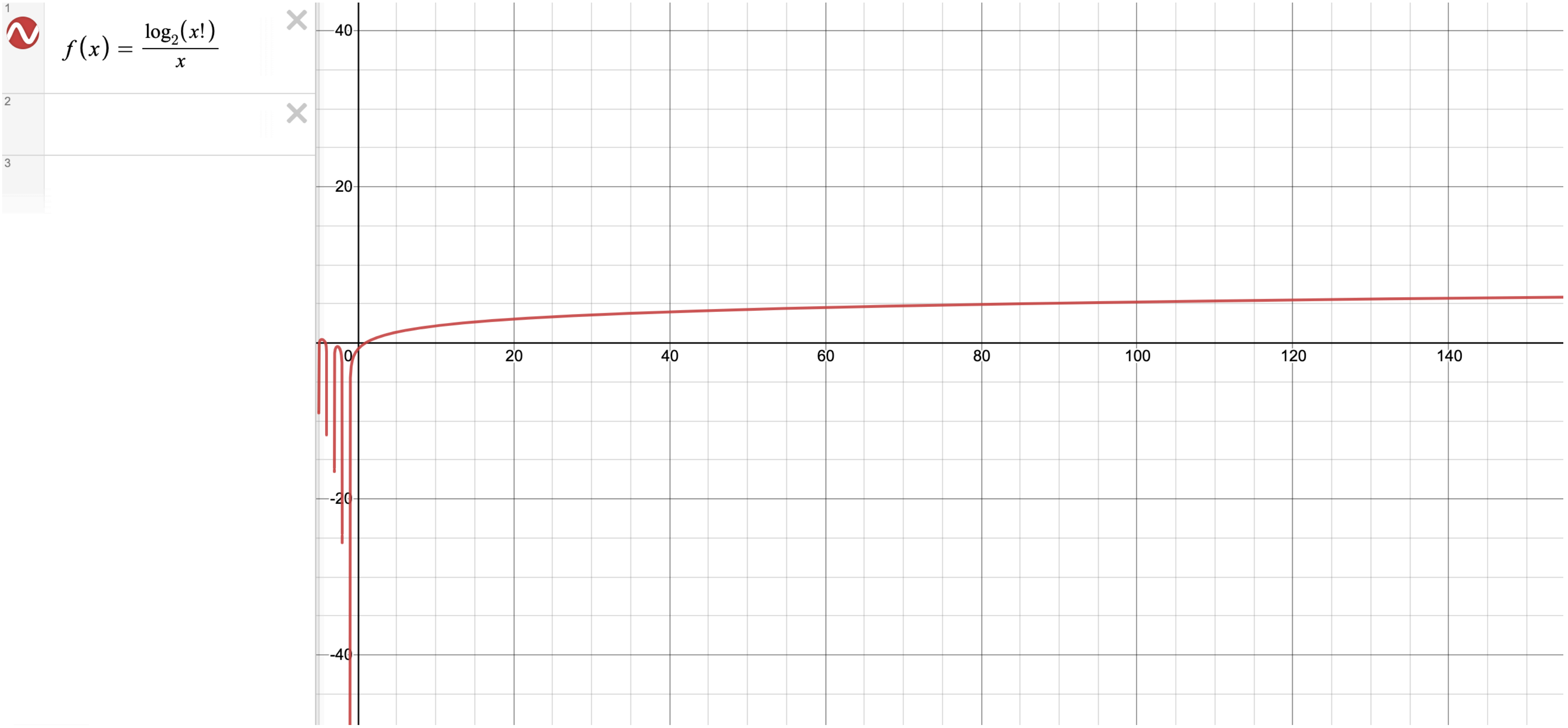
This function has a slope of almost zero for
a significantly large V
and
It is a retarding function.
So, We can take its value as Constant.

$$f(p) = \frac{\log_p(V!)}{V}$$

It is a Monotonic decreasing
function w.r.t. p for given V, p,v>0

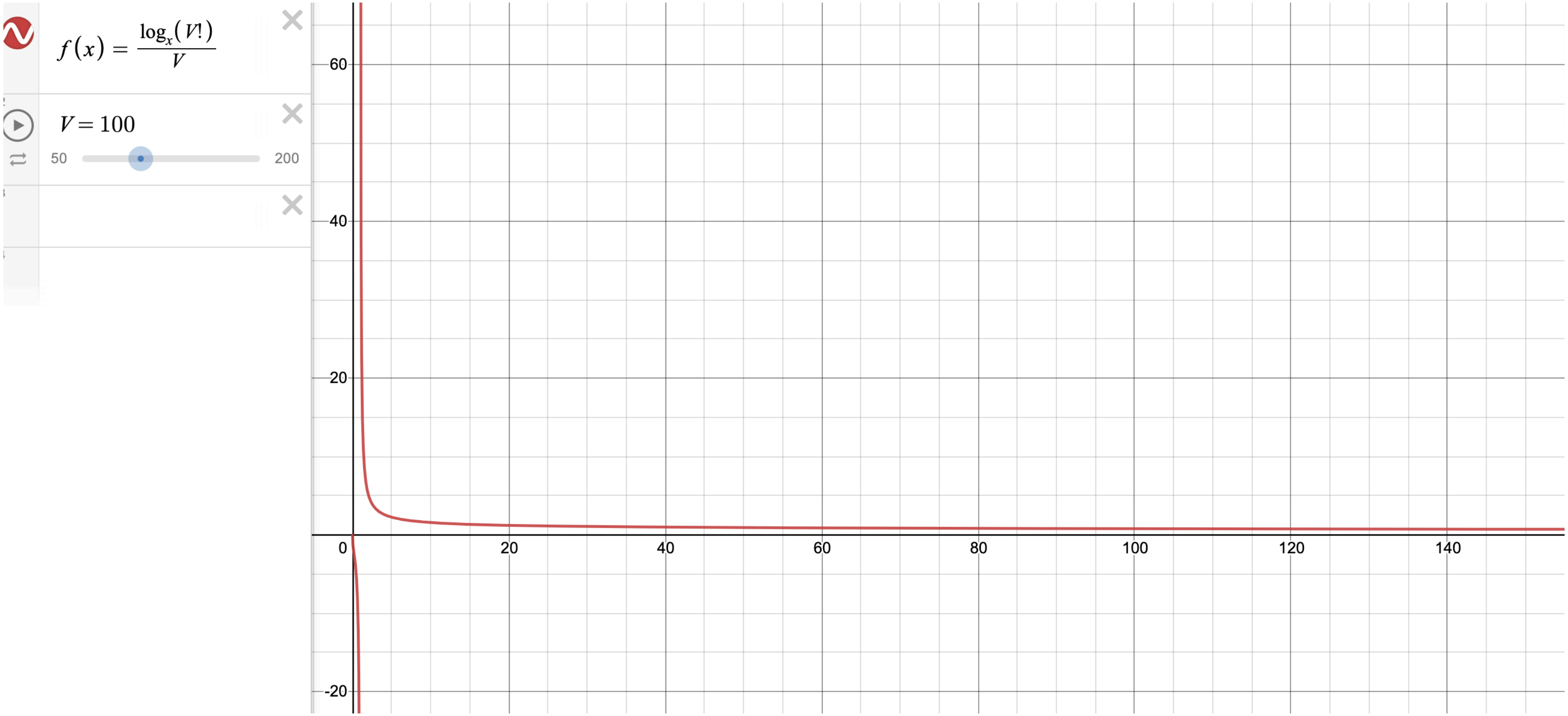
- Since 2-ary version tree returns the value in almost constant time, any retrieval from an array having p-ary (p>2, what we do in Full Persistence) version history tree, has a time complexity of O(1).

Function Behaviour (1)



We can see, that f(x) Almost converges

Function Behaviour (2)



$$f(p) = \frac{\log_p(V!)}{V}$$

It is a Monotonic decreasing
function w.r.t. p for given V, p,v>0