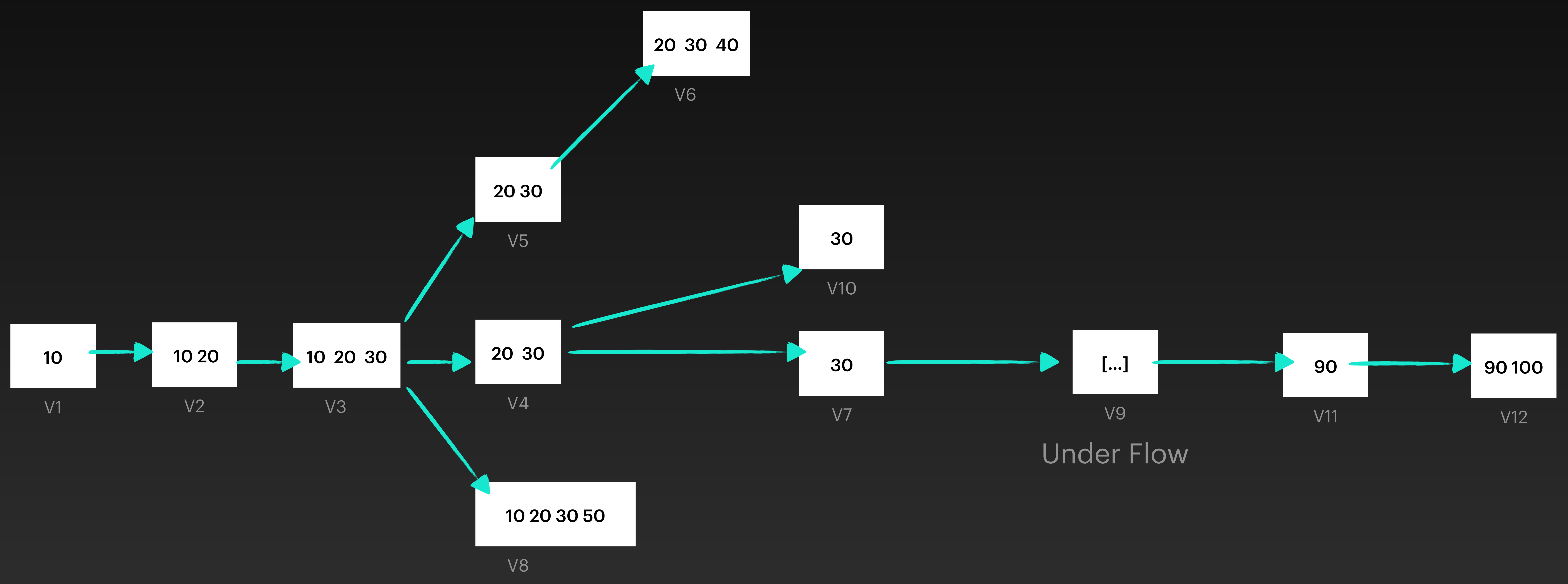
Full Persistent Queue

Using Binary Lifting and Sparse Table

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

Ephemeral Equivalent



Revisiting Binary Lifting and UpTable

We will store this n-arry Version Tree in Special Type of Sparse Matrix

J

V

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
5	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
6	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
7	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
8	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
9	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
10	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
11	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
12	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Initially All Cell are -1

This is UP_TABLE

UP_TABLE[v][j] = 2^j th ancestor of v

So,
There are **MAX_VER_COUNT** rows
There are **log2(MAX_VER_COUNT)** columns

We will hold **MAX_VER_COUNT = 2^16 = 65536 versions in RAM at a time**

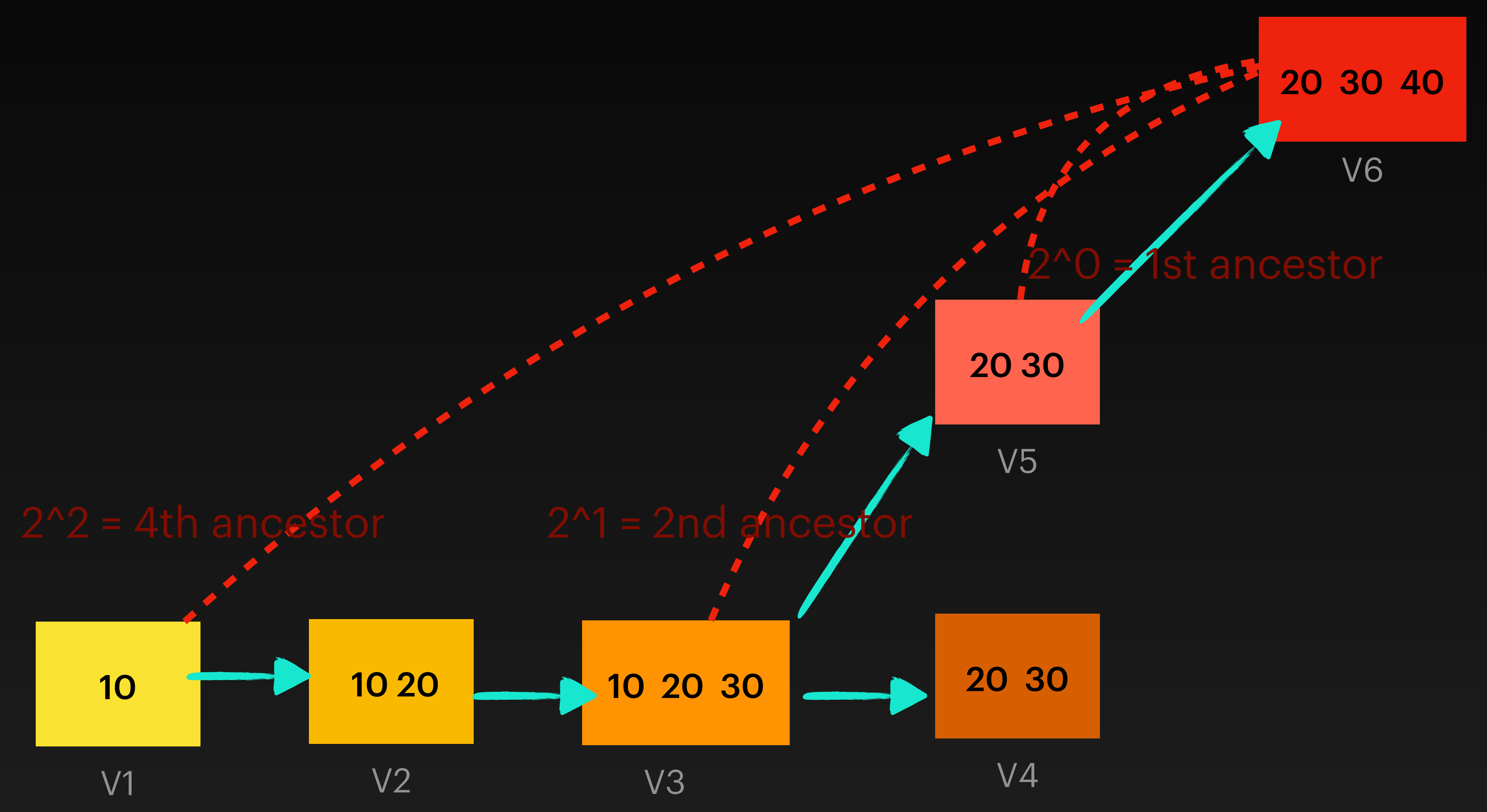
We will store this n-arry Version Tree in Special Type of Sparse Matrix

J

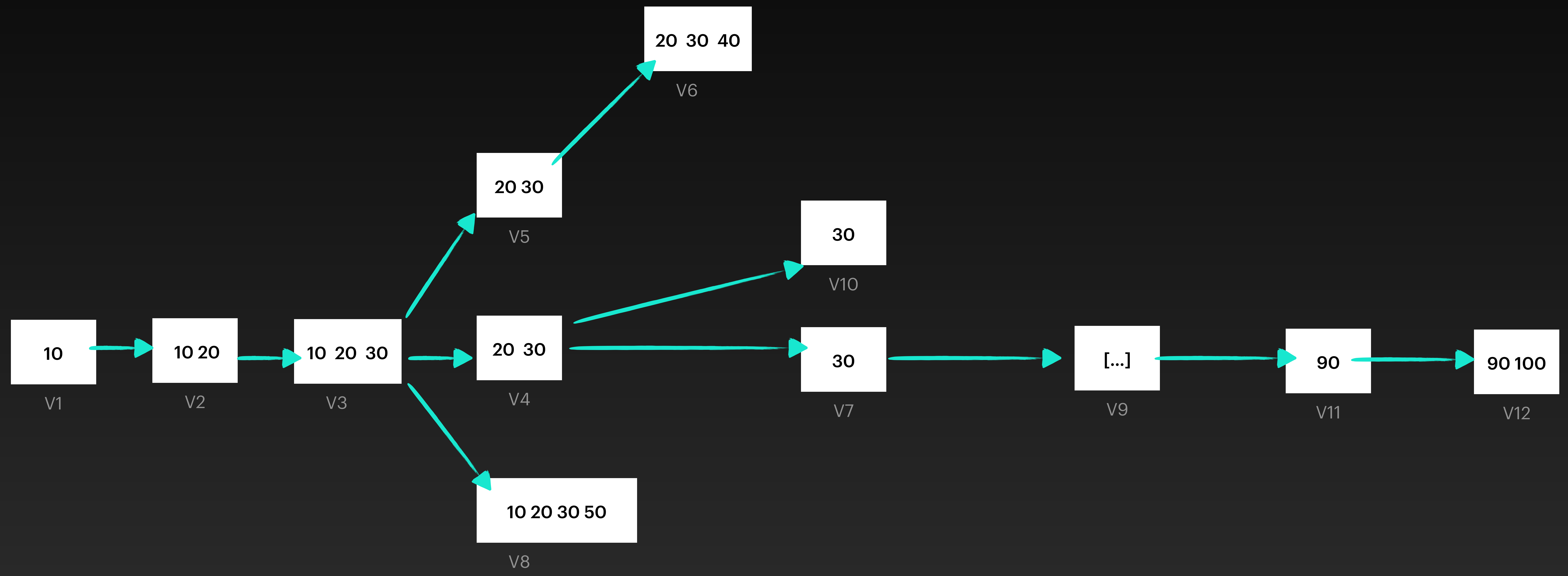
V

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
3	2	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
4	3	2	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
5	3	2	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
6	5	3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
7	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
8	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
9	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
10	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
11	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
12	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Initially All Cell are -1



Store The Versions in UP_TABLE



J

V

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
3	2	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
4	3	2	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
5	3	2	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
6	5	3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
7	4	3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
8	5	3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
9	7	4	2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
10	4	3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
11	9	7	3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
12	11	9	4	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Initially All Cell are -1

How to fill the row for v th version ???

2^0 th ancs. Of v (say vp0)

2^1 th ancs. Of v = 2^0 th ansc. Of vp0

...

...

2^j th ancs. Of v = 2^j th ansc. Of vp_j-1

UP_TABLE [v][0] = Parent of V

UP_TABLE [v][1] = UP[UP_TABLE [v][0]][0]

UP_TABLE [v][1] = UP[UP_TABLE [v][0]][0]

...

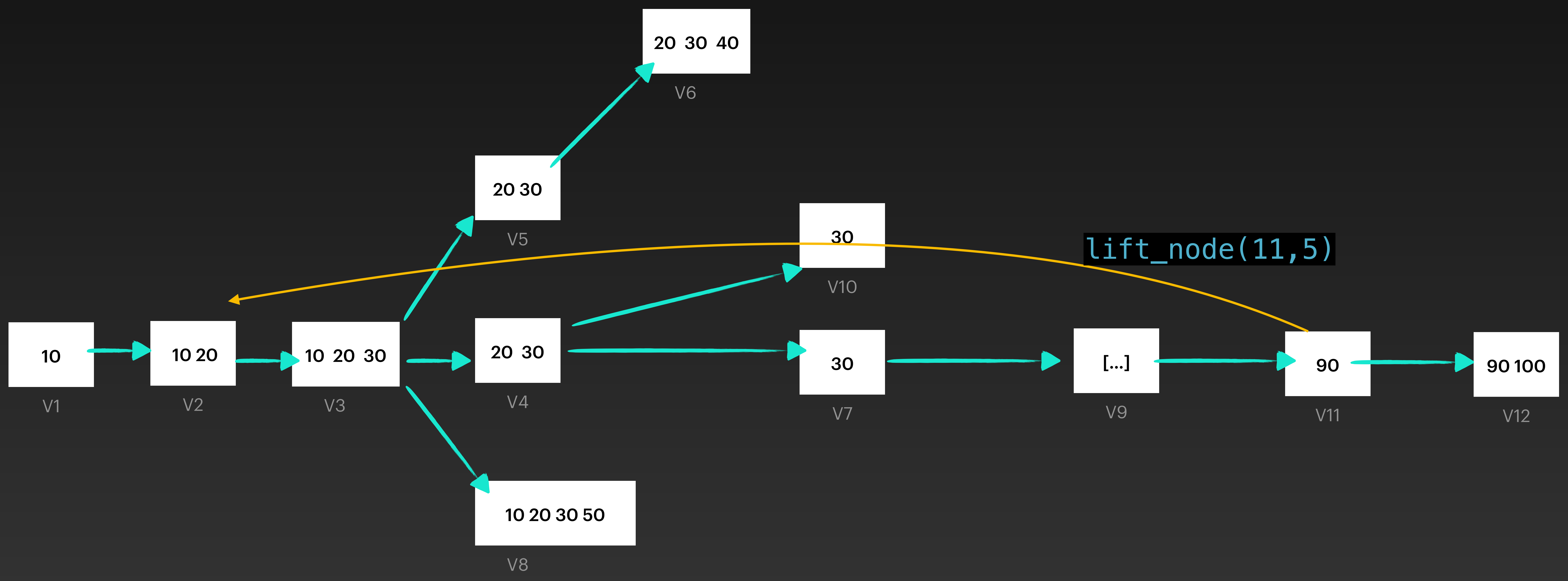
...

...

```cpp
void FP_QUEUE:: add_to_table(int src,int par){
    table[src][0] = par;
    for(int  i=1;i<20;i++){
        if(table[src][i-1]==-1)
            break;
        table[src][i] = table[ table[src][i-1]][i-1];
    } // as par > child ( No need to do BFS)
}
```

How to get
Which was the version T times before V?
Hence
We have to jump T unit above V

```
int lift_node(int node, int jmp_req);
```



FROM VER 11

JUMP_REQ = 5

1 0 1

2^2 2^1 2^0

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
3	2	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
4	3	2	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
5	3	2	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
6	5	3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
7	4	3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
8	5	3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
9	7	4	2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
10	4	3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
11	9	7	3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
12	11	9	4	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
3	2	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
4	3	2	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
5	3	2	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
6	5	3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
7	4	3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
8	5	3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
9	7	4	2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
10	4	3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
11	9	7	3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
12	11	9	4	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

FROM VER 11

JUMP_REQ = 5

1 0 1

2^2 2^1 2^0

No movement as bit = 0

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
3	2	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
4	3	2	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
5	3	2	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
6	5	3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
7	4	3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
8	5	3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
9	7	4	2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
10	4	3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
11	9	7	3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
12	11	9	4	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

FROM VER 11

JUMP_REQ = 5

1 0 1

2^2 2^1 2^0

So,

`lift_node(11,5) = 2`

Optimising Space

```
int table[MAX_NO_VER][LOG_MAX_NO_VER]
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
3	2	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
4	3	2	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
5	3	2	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
6	5	3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
7	4	3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
8	5	3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
9	7	4	2	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
10	4	3	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
11	9	7	3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
12	11	9	4	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Space : O (V * log V)

```
vector<vector<int>>TABLE;
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0	-1														
1	0														
2	1	0													
3	2	1													
4	3	2	0												
5	3	2	0												
6	5	3	1												
7	4	3	1												
8	5	3	1	-1											
9	7	4	2	-1											
10	4	3	1	-1											
11	9	7	3	-1											
12	11	9	4	0											

-> Size: log(6) + 1 = 3

Size of Reduced Sparse Table Calculation:

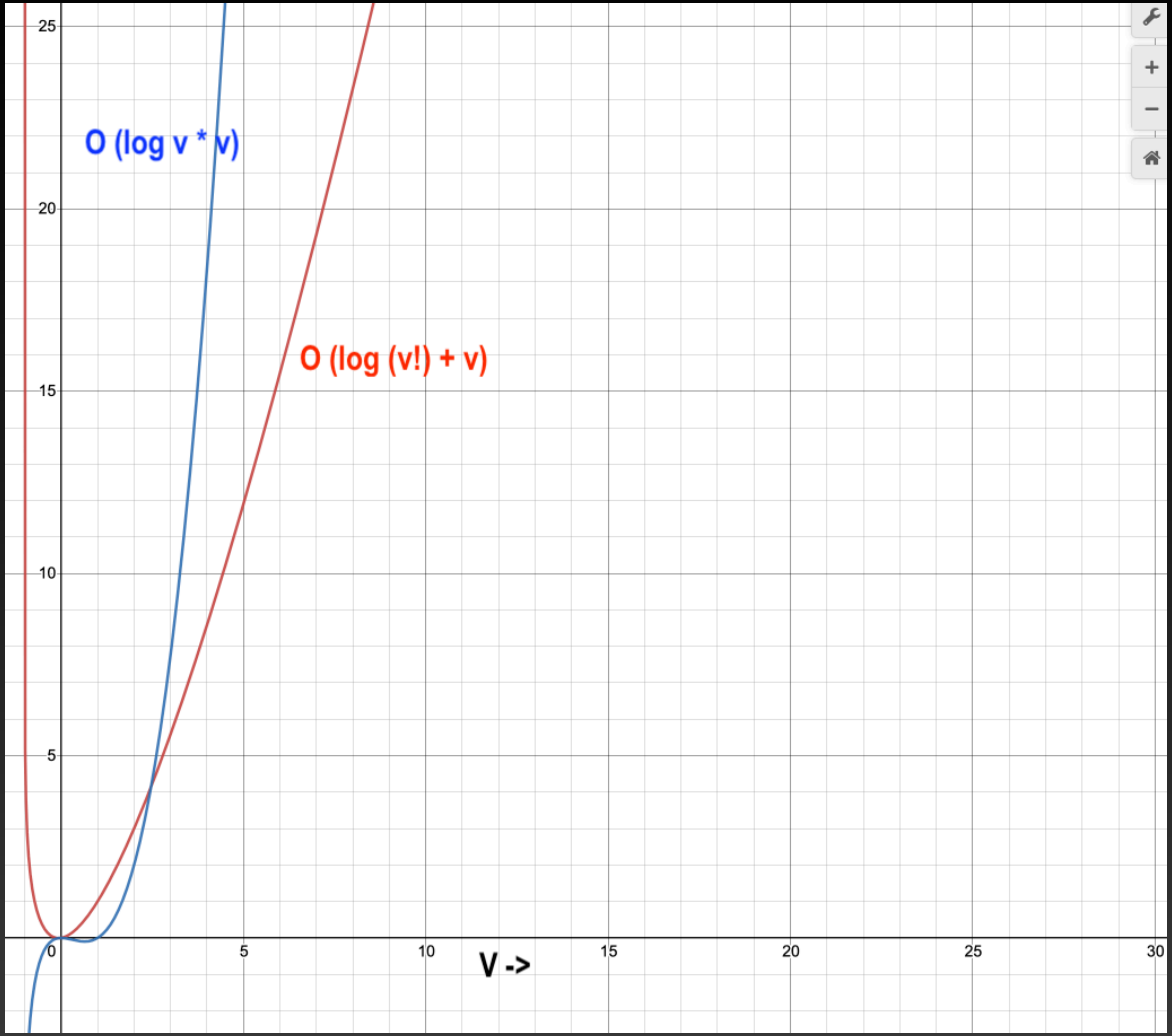
$$1 + (\log(1)+1) + (\log(2)+1) + (\log(3) + 1) + \ldots + (\log(MAX\_V) + 1)$$
$$= \Sigma\,(1 + \log V)$$
$$= v + \Sigma \log v$$
$$= v + \log (v!)$$

Space : O (V + log (V!))

O (V * log V)

Vs

O (V + log (V!))



```
int lift_node(int node, int jmp_req);
```

Time complexity: O(log v)

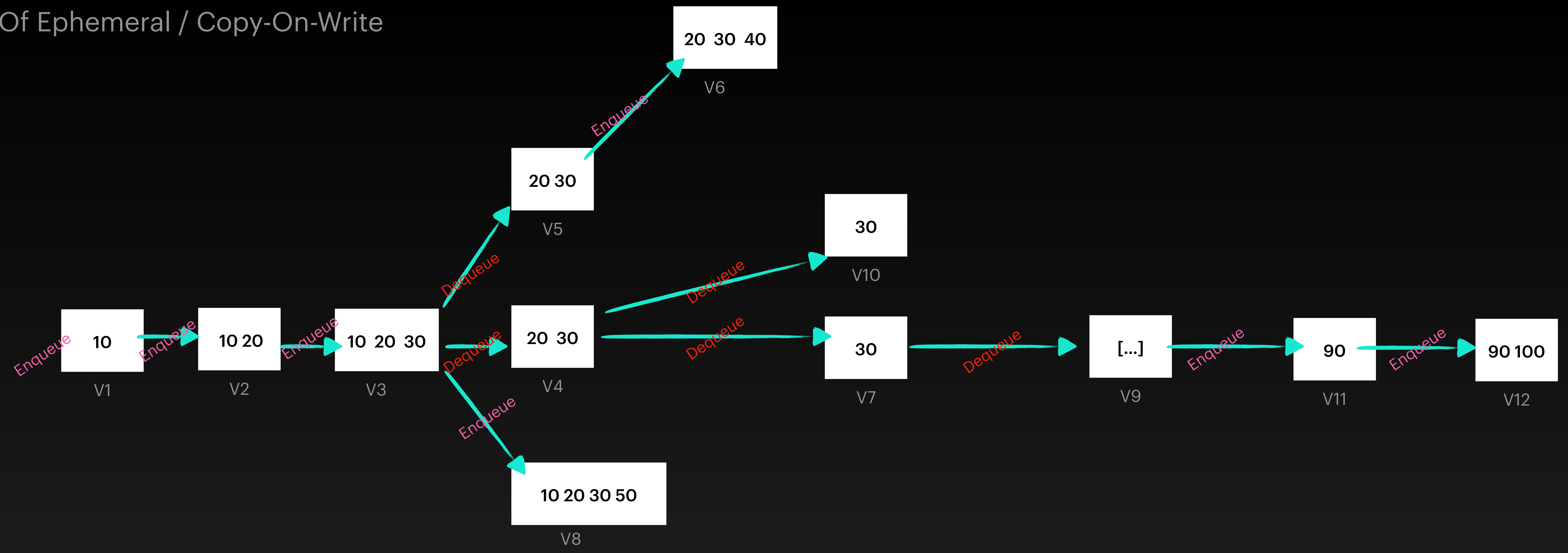
```
int FP_QUEUE:: lift_node(int node, int jmp_req){
    for(int i = 19; i>= 0;i--){
       if (jmp_req == 0 || node == -1)
           break;

        if (jmp_req >= (1<<i)){
            jmp_req -= (i<<i);
            node = table[node][i];
        }
    }
    return node;
}
```

So ... handling of version branching done!!!

Now How to store the Queue Data!!

```
class FP_QUEUE
{
private:
    // ver_child -> (2^i th)ver_ancsestor_par mapping
    int table[MAX_NO_VER][LOG_MAX_NO_VER];
    // map(ver) = {data, front_ver,cur_length} of rear
    int map[MAX_NO_VER][3] ;
    // to hold what type of update happened in version v, 'E' for Enqueue & 'D' for Dequeue
    char type_of_update[MAX_NO_VER];
    // to hold the real times
    int cur_time;

...

public:
...
};
```

Abstract Layout Of Ephemeral / Copy-On-Write

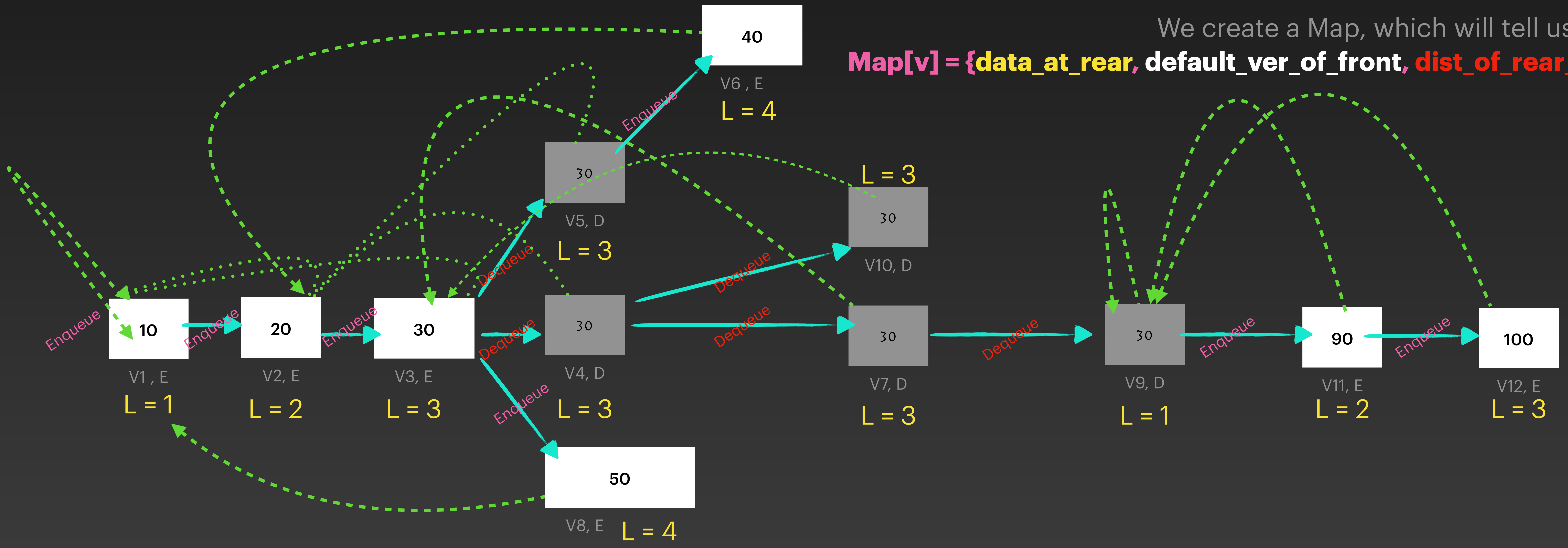


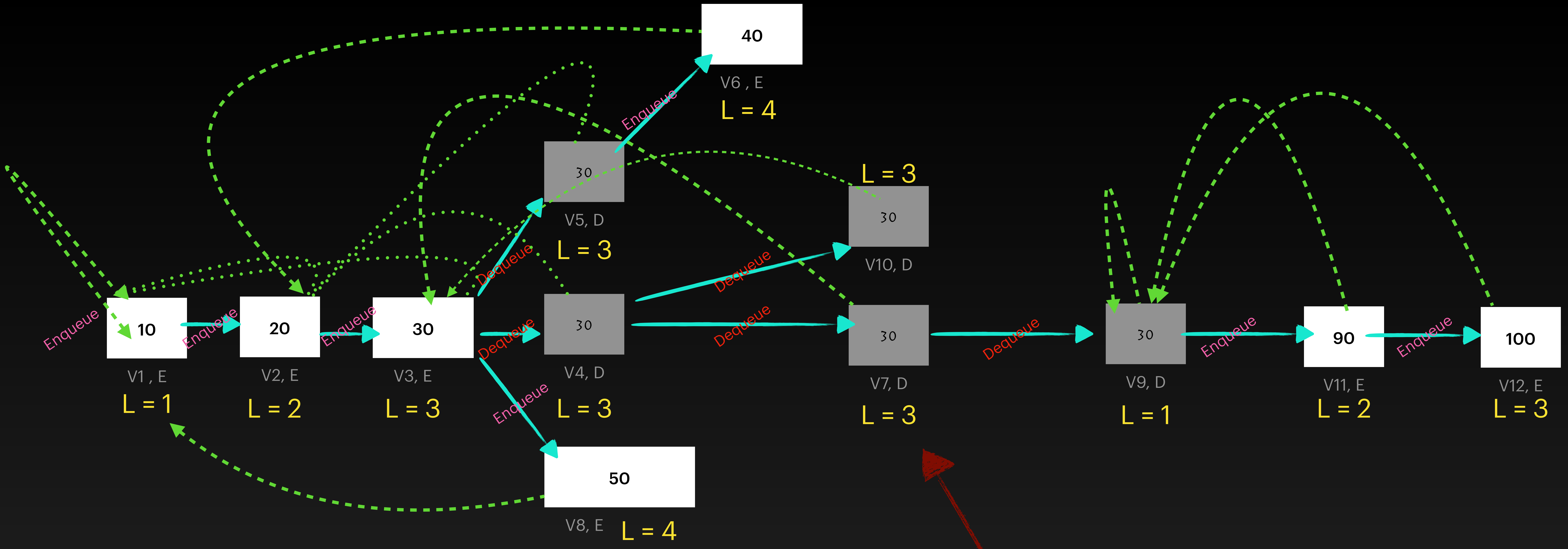
Abstract Layout Of Persistence

Here we don't create any Physical Node through LinkedList

We create a Map, which will tell us

Map[v] = {data_at_rear, default_ver_of_front, dist_of_rear_from_1st_'E'_type_node}





int map[MAX_VER][3]

	REAR_DATA	default_ver_of_front	dist_of_rear_from_1st_'E'_type_node
0	-1	0	0
1	10	1	1
2	20	1	2
3	30	1	3
4	30	2	3
5	30	2	3
6	40	2	4
7	30	3	3
8	50	2	4
9	30	9	1
10	30	3	3
11	90	9	2
12	100	9	3

char type_of_ver[MAX_VER] ... it stores "After which type of operation v created?"

Type
-
E
E
E
D
D
E
D
E
D
E
E
E

Example: v7

How to Enqueue?

```
// to enqueue the Data at rear along par_ver
void enqueue(int data, int par_ver);
```

Example:

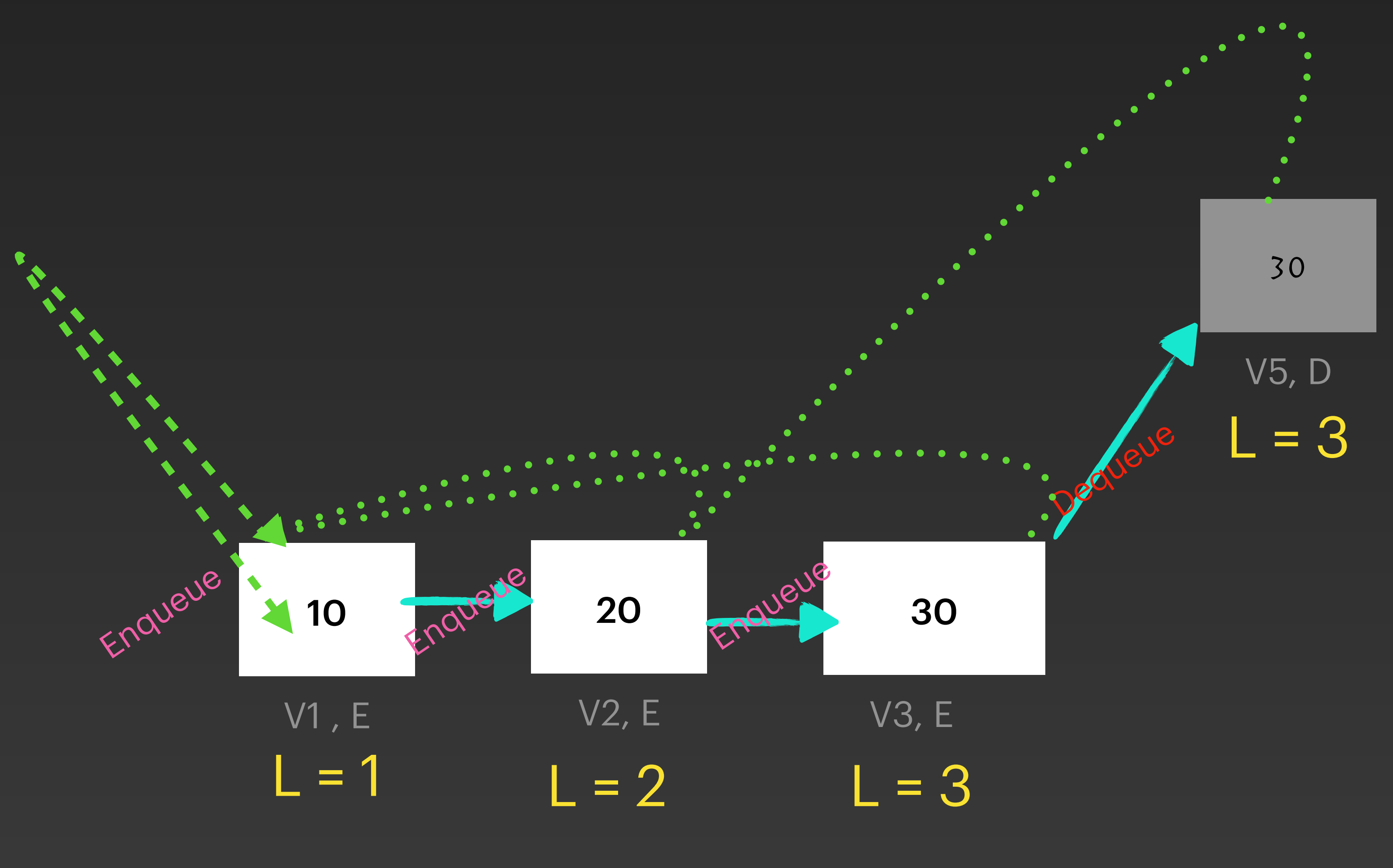
```
q.enqueue(40, 5);
```

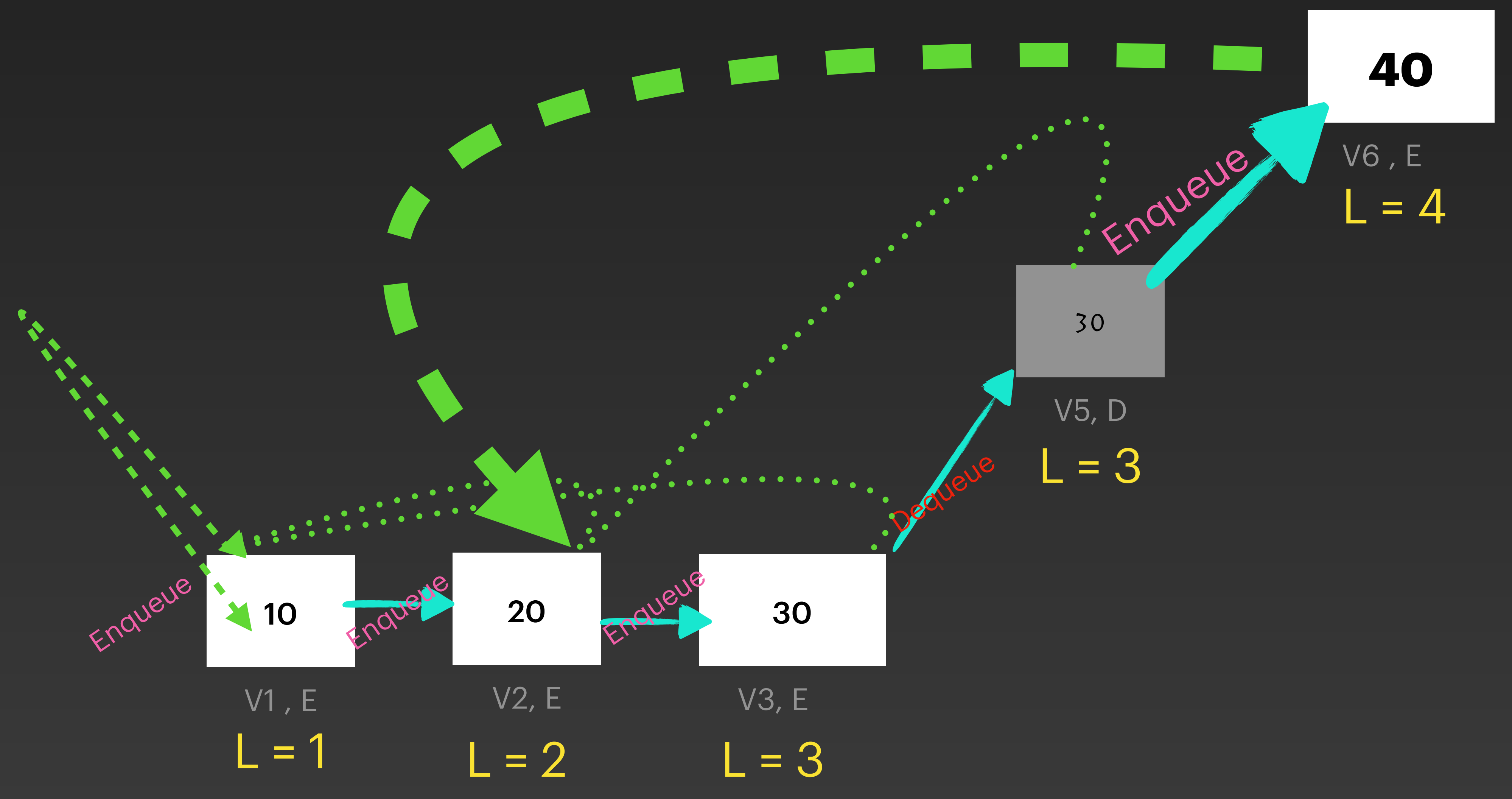
Strategy

1. Check For Valid Version
2. Cur_Time++;
3. Create A Mapping of Node Details at Current Time
4. Here “default_ver_of_front“ is the “default_ver_of_front“ of the par_ver
5. Type will be ‘E’

	REAR_DATA	default_ver_of_front	dist_of_rear_from_1st_‘E’_type_node
6	40	2	4

Type
E





How to Enqueue?

C++ Code:

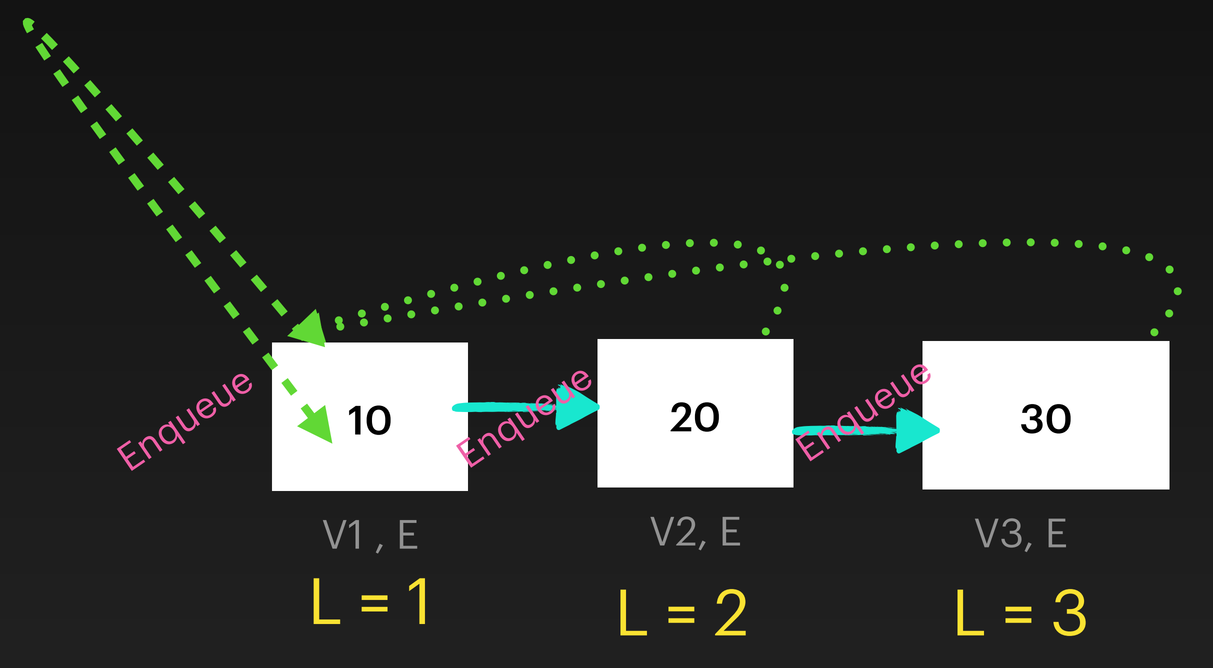
```cpp
void enqueue(int data, int par_ver){
    CHECK_VERSION_(par_ver)
    if(cur_time == MAX_NO_VER){
        cout<<"No Support to hold further versions in RAM ... You store them in Disk\n";
        return;
    }

    cur_time++;
    map[cur_time][0] = data;
    int cur_legth = map[cur_time][2] = map[par_ver][2]+1; // length ++
    map[cur_time][1] = (cur_legth!=1)?map[par_ver][1]:cur_time; // front = parent's front
    type_of_update[cur_time] = 'E';
    add_to_table(cur_time, par_ver);
}
```

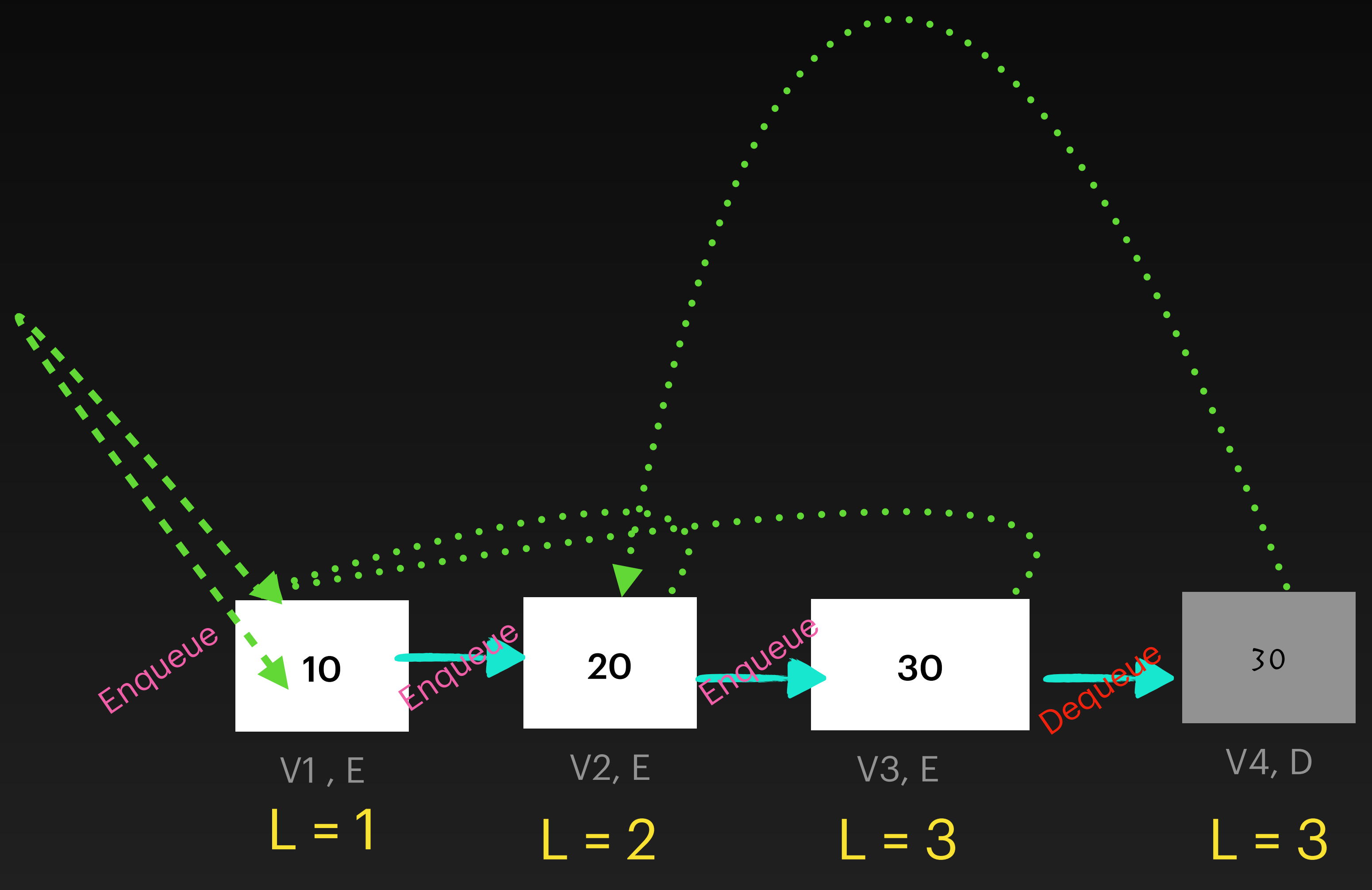
How to Dequeue?

```
// to dequeue the Data at front along par_ver
int dequeue(int par_ver);
```

Case1 : If There is a ‘E’ type node at [L-2] unit above

```
q.dequeue(3);
```





	REAR_DATA	default_ver_of_front	dist_of_rear_from_1st_‘E’_type_node
4	30	2	3

Type
D

Strategy

1. Check For Valid Version
2. Check for UnderFlow at that version, if UnderFlowed -> Return
3. Cur_Time++;
4. Create A Mapping of Node Details at Current Time, and, copy the DATA and LENGTH part. LENGTH will be same as there is no further ***shrinking***
5. Here “default_ver_of_front“ is the one level below “default_ver_of_front“ of the par_ver as there is no further ***shrinking***
6. **To Obtain this, we retrieve the Version Node [L-2] above the REAR through Binary Lifting**
7. Type will be ‘D’

How to Dequeue?

```
// to dequeue the Data at front along par_ver
int dequeue(int par_ver);
```

Case2 : If There is a 'D' type node at [L-2] unit above, We will shrink the length further

```
q.dequeue(7);
```

Strategy

1. Similar as CASE 1 except the "default_ver_of_front" and "dist_of_rear_from_1st_'E'_type_node" field
2. Here, if we get 'D' type Version Node after lifting of [L-2] level above, we will keep on lifting [L-3], [L-4] .. , [1], [0] level above REAR, Until we find a 'E' type node or we reach the REAR NODE itself. In Parallel way, we will keep track how much further SHRINKING has been done.
3. Here, the v7 node redirects to v4 after ([L-2] = 3- 2 = 1) level Lifting. v4 is 'D' type node, so, we do further Shrinking and we reach v7. Still v7 is a 'D' type node. Then, we reach V9 and we stop. So, we set map[9][1] = 9.

4. We have 2 units of FURTHER_SHRINK. So, we will deduct that amount from the length, ser, map[9][2] = 1 (3 -2)

```
q.dequeue(7);
```

	REAR_DATA	default_ver_of_front	dist_of_rear_from_1st_'E'_type_node
9	30	9	1

Type
D

How to Dequeue?

C++ Code

```
int dequeue(int par_ver){
    CHECK_VERSION(par_ver)

    if(cur_time == MAX_NO_VER){
        cout<<"No Support to hold further versions in RAM ... You store them in Disk\n";
        return SENTINEL;
    }
    int prev_length = map[par_ver][2];
    int prev_front = map[par_ver][1];

    if(type_of_update[map[par_ver][1]]=='D' || map[par_ver][1]==0){
        cout<<"Queue Underflowed!!\n";
        return SENTINEL;
    }

    int shrink = 0;
    int new_length = prev_length; // +1 -1
    int new_front = getNewFront(par_ver,prev_length-2,shrink); // upshift will deduct by 2

    // the follwing condition happens iff there is no element left
    // there is no need to do this ... just for sake of understanding
    if(new_front==par_ver && type_of_update[new_front]=='D'){
        new_front = cur_time+1;
        shrink++;
    }

    new_length-=shrink;

    cur_time++;

    map[cur_time][0] = map[par_ver][0]; // front data won't change
    map[cur_time][1] = new_front;
    map[cur_time][2] = new_length;
    type_of_update[cur_time] = 'D';
    add_to_table(cur_time, par_ver);
    return map[prev_front][0];
}
```

```
// to return 'E' type node which is atmost _upShift_
//unit above than rear at par_ver and also furthest to rear at par_ver
```

```
int getNewFront(int par_ver,int up_Shift, int &shrink){
    CHECK_VERSION(par_ver)
    int up_node = lift_node(par_ver,up_Shift);

    if(up_node==par_ver)
        return up_node;

    if(type_of_update[up_node]=='E')
        return up_node;
    else{
        shrink++;
        return getNewFront(par_ver, up_Shift-1,shrink);}
}
```

Time Complexity

- Enqueue(data,par_ver): O(log_2 v), In Average It Is ~ O(1), Proof is Given In Next Page
- Dequeue(par_ver):
  - O(log_2 v) [In Average Case]
  - O(V . log_2 v) [In Worst Case]
- rear(ver): O(1)
- front(ver):
  - O(1)   [In Average Case]
  - O(log_2 v)   [In Worst Case]

Auxiliary Space Complexity

- O (V + log (V!)) to Hold The UP_TABLE
- O (V) to Hold the MAP
- O (V) to Hold the TYPE_OF_VER

Why Enqueueing Has Almost Constant Time ?

When there are v versions in our current state,
we face O($\log_2 v$) to enqueue an element.
So,we have to take the average from v = 1 to V [$\sum v = V$, total Versions]

Average time taken = $O(\sum \log_2 v) / V = O(\log_2 V!)/V \approxeq O(\text{constant})$

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

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.

Function Behaviour

We can see, that f(x) Almost converges