Example of storing sums

We have a tree where every node
stores some properties about an interval.

The root-node's interval is 1-N (all the elements).

The leaves' interval is x-x (only one element).

All the other nodes have two children.
Let the node's interval = [x,y] and mid = (x+y)/2
Then the left-children interval is [x,mid]
and the right-children interval is [mid+1,y]

Updating a node
(in our case adding)

1) Follow the path to this node

2) Update it's value (add the given value)

3) Climb the path updating the nodes' values, using their children's values
(value(child1) + value(child2))

Complexity O (logN) since we do a constant amount of work at each node, and we visit O (logN) nodes.

In the following examples :

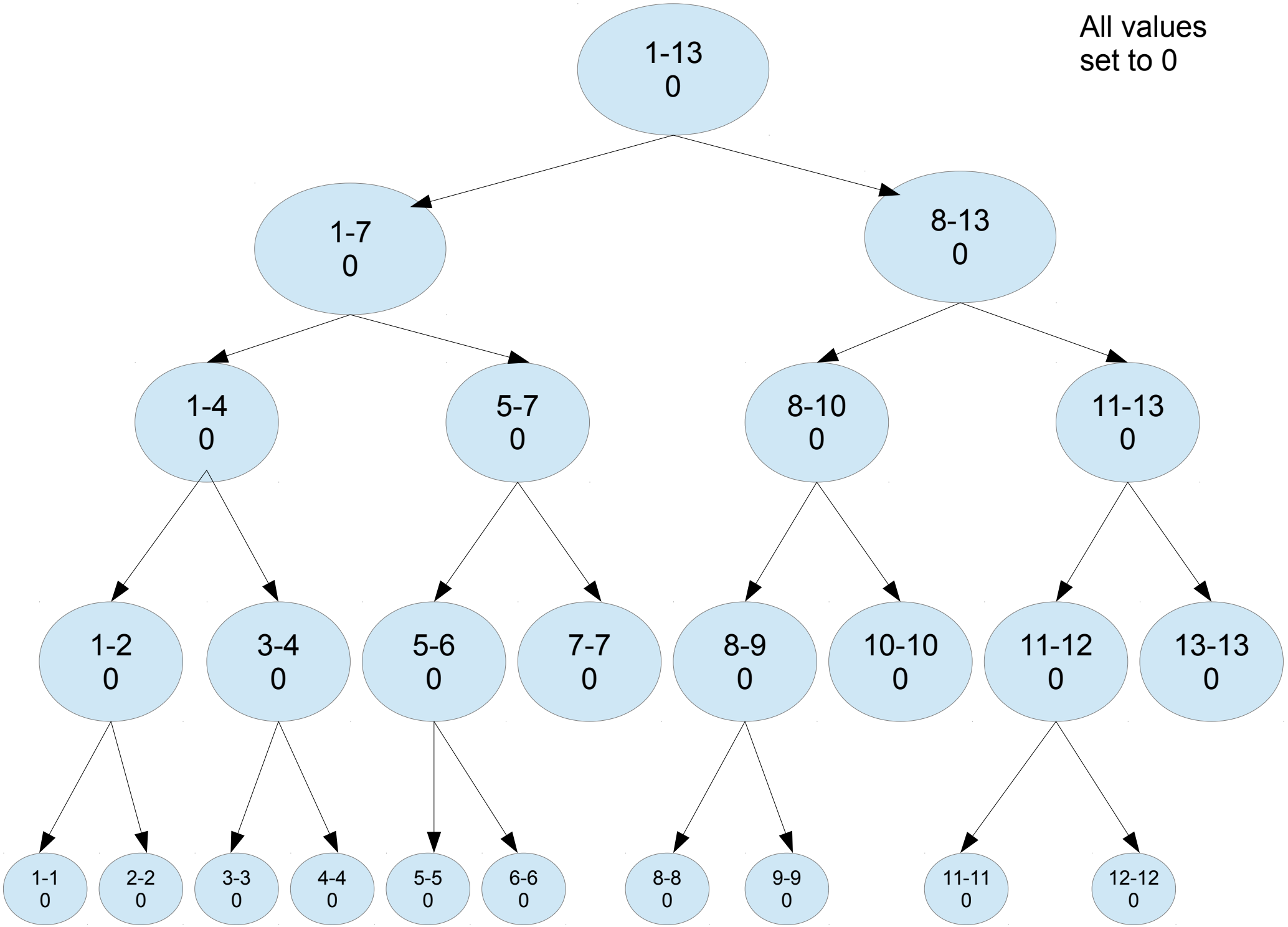
Blue means this node hasn't been reached.

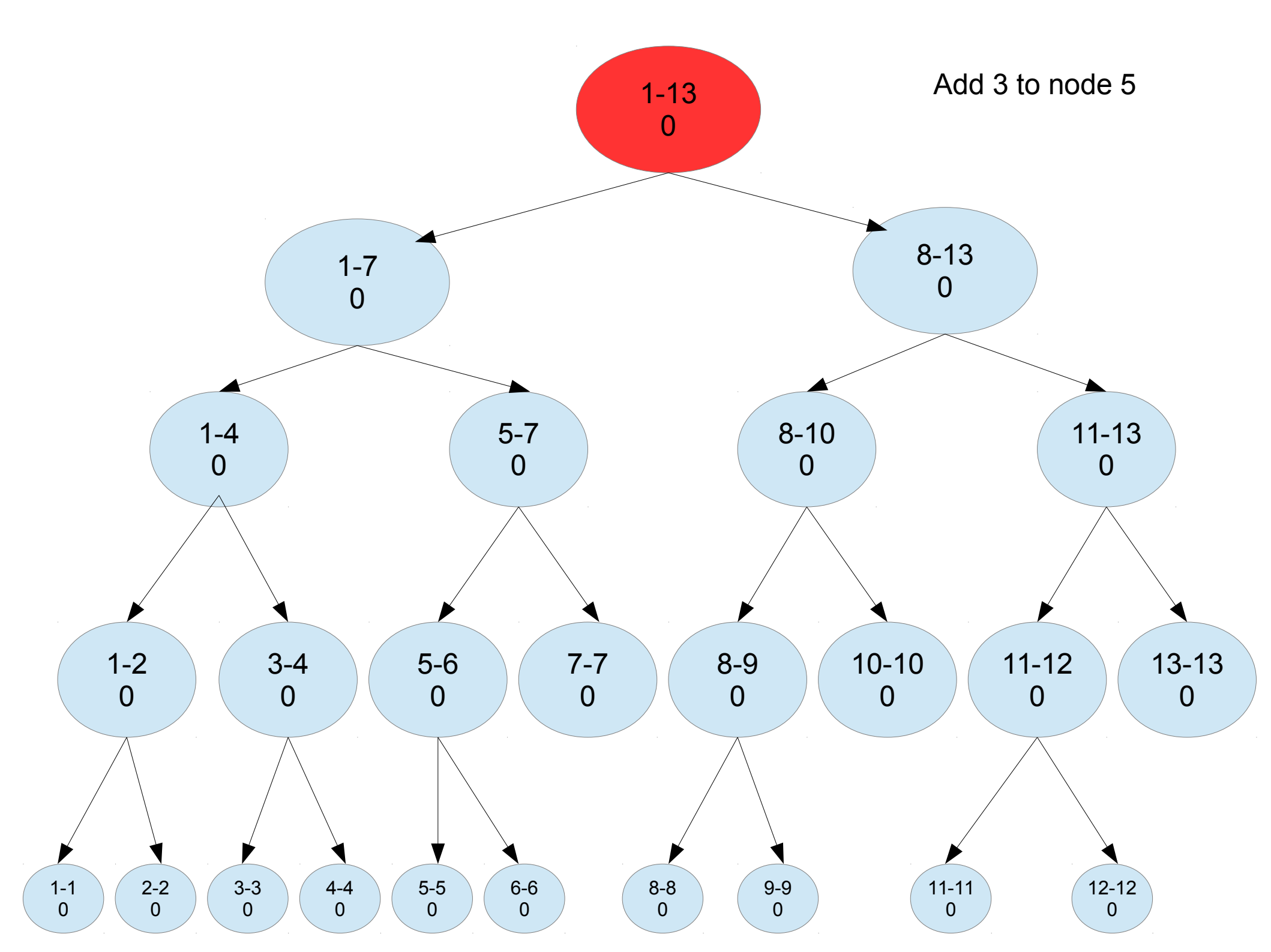
Red means we still process this node.

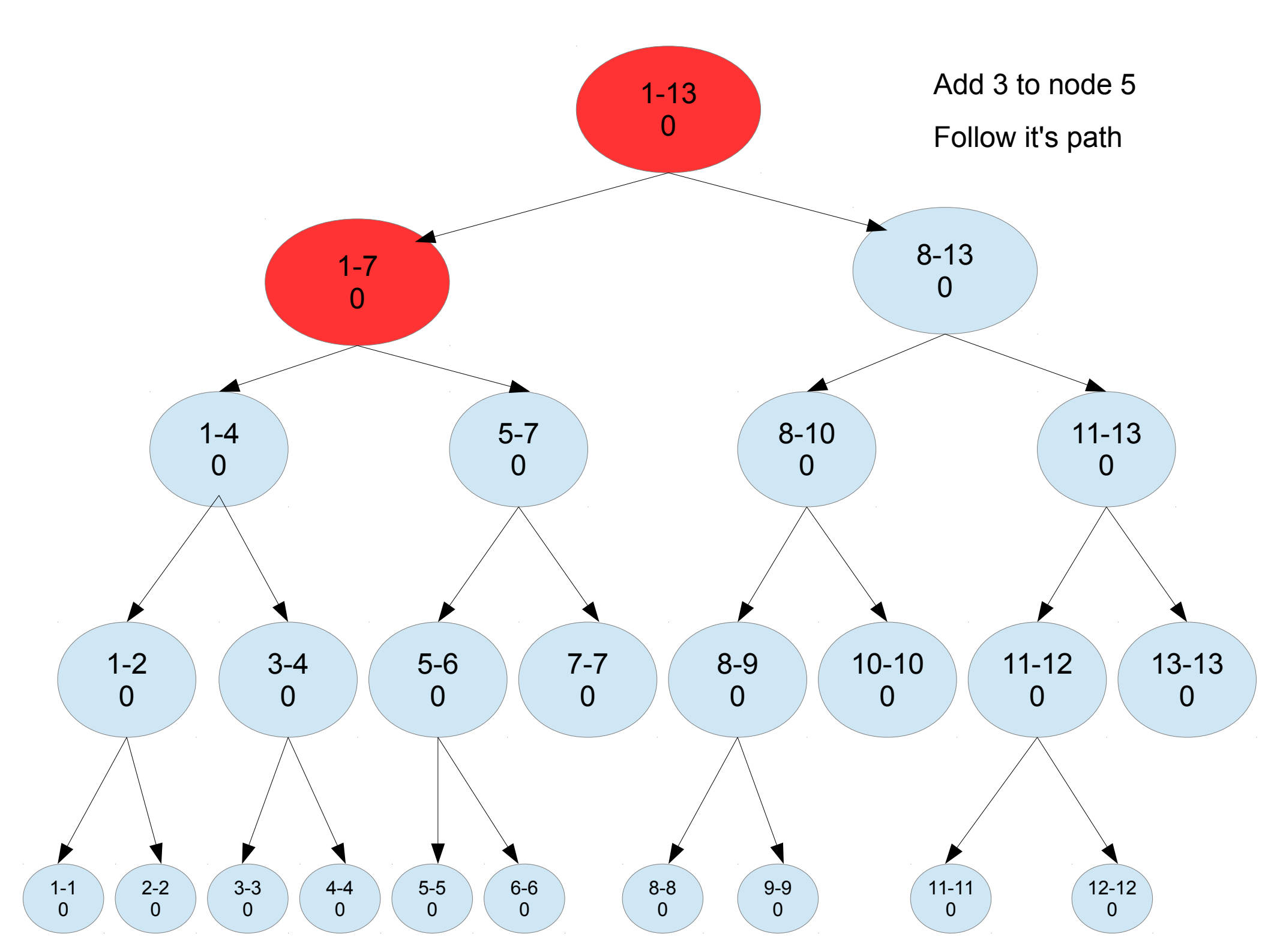
Purple means we are done processing this node.

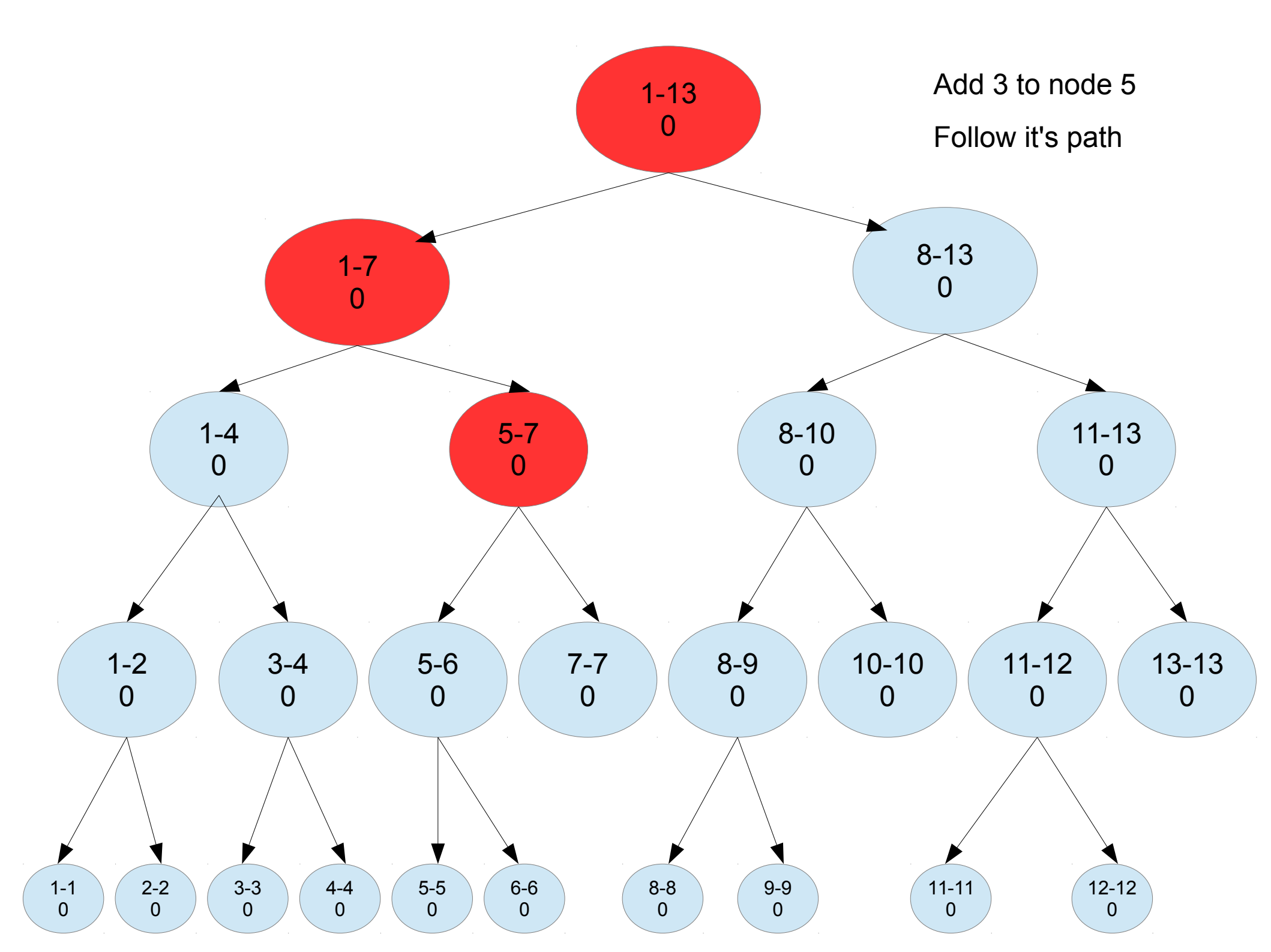
Code in C++ is given at the end of the presentation.

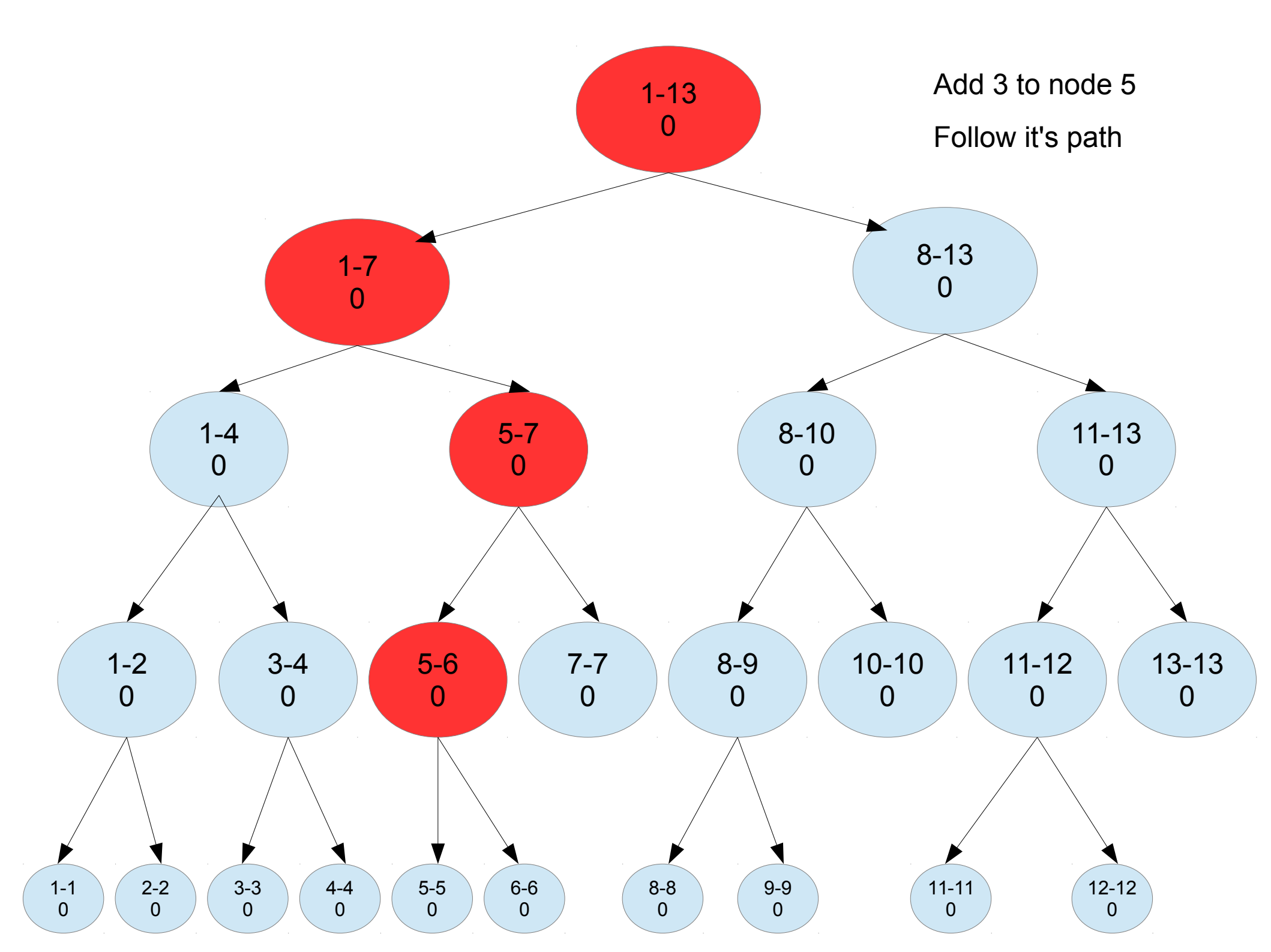
If you don't need lazy propagation, just remove the
highlighted parts.

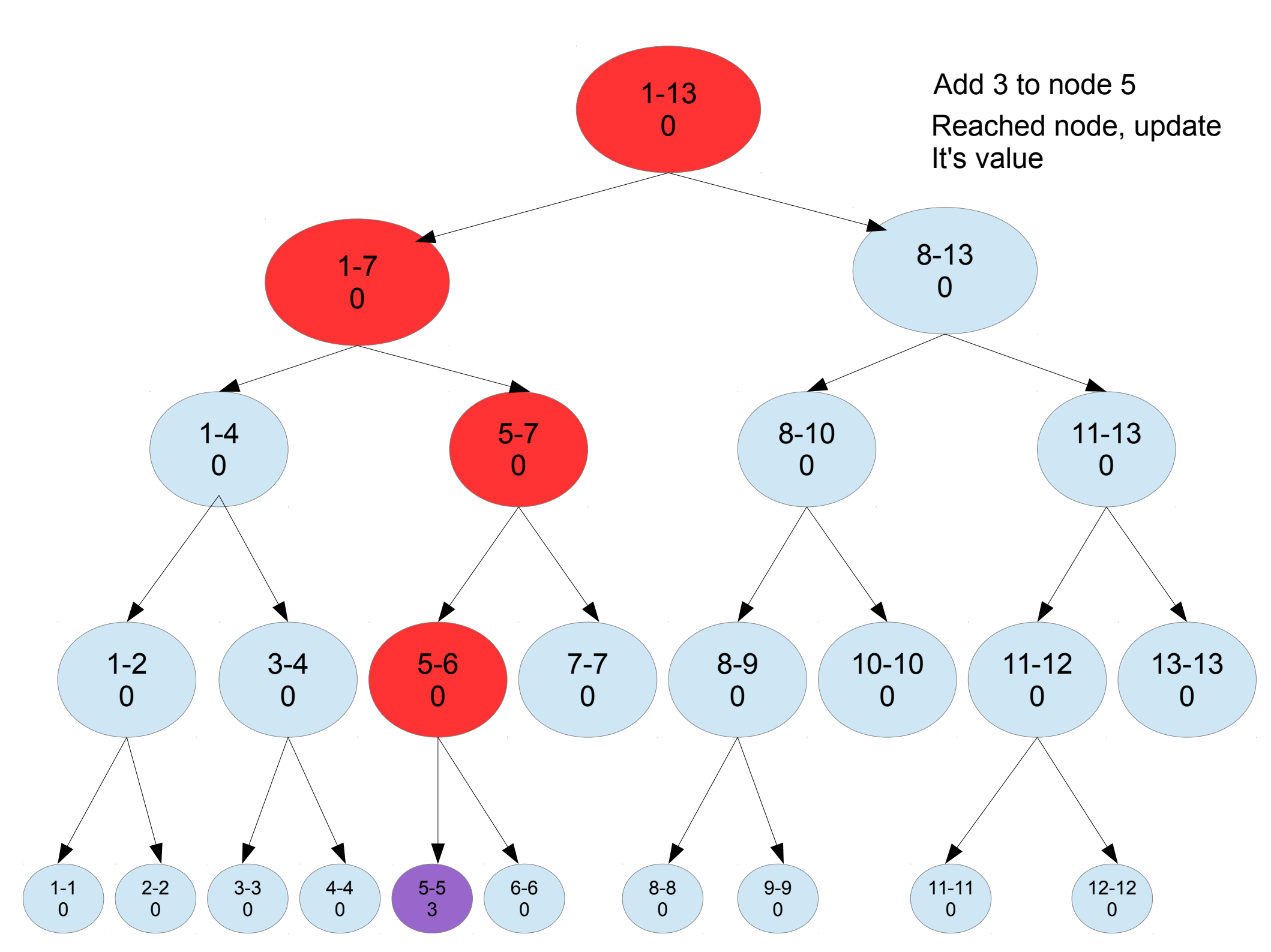

All values
set to 0
1-13
0
1-7
0
8-13
0
1-4
0
5-7
0
8-10
0
11-13
0
1-2
0
3-4
0
5-6
0
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
0
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0

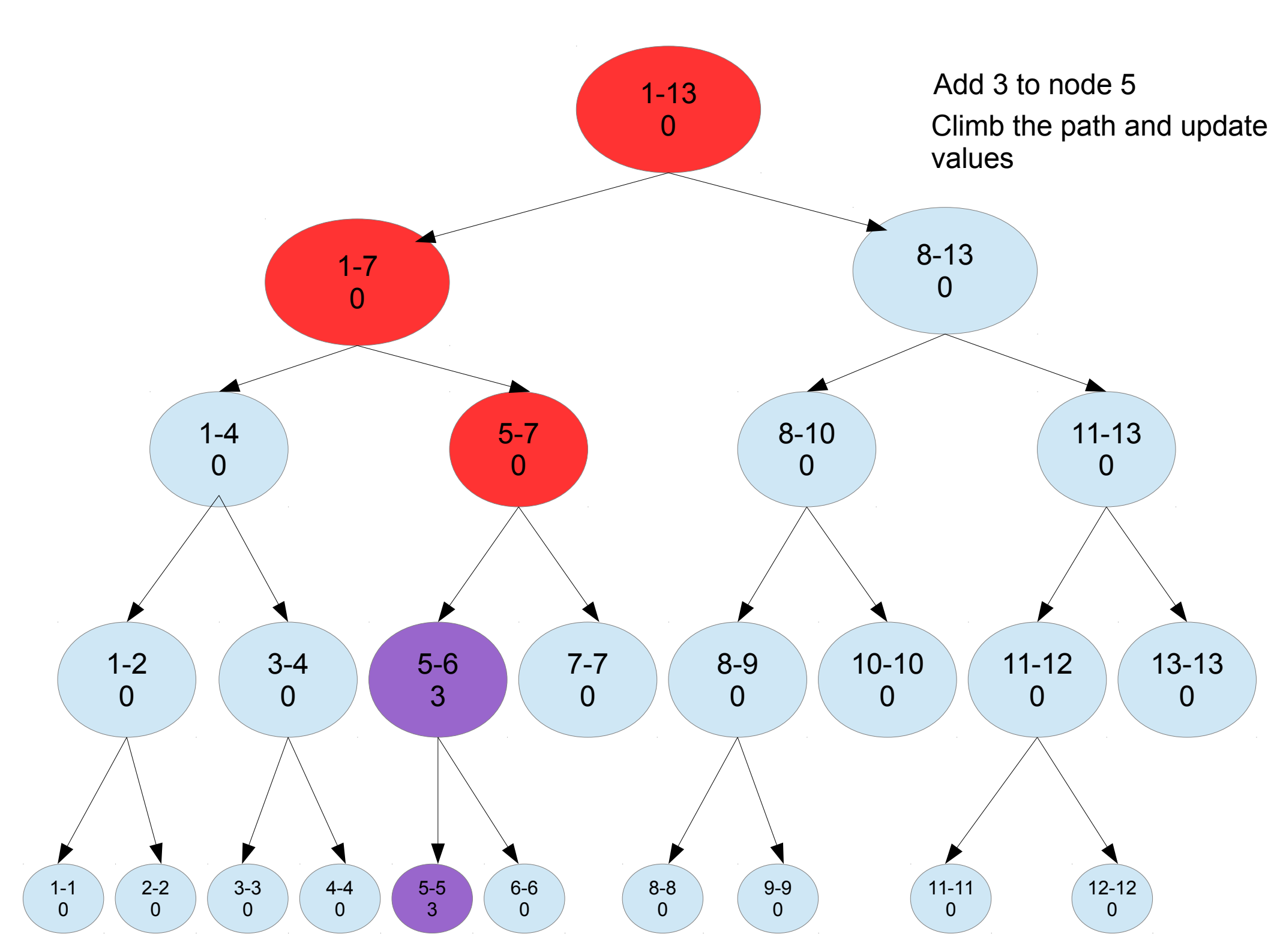

Add 3 to node 5
1-13
0
1-7
0
8-13
0
1-4
0
5-7
0
8-10
0
11-13
0
1-2
0
3-4
0
5-6
0
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
0
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0

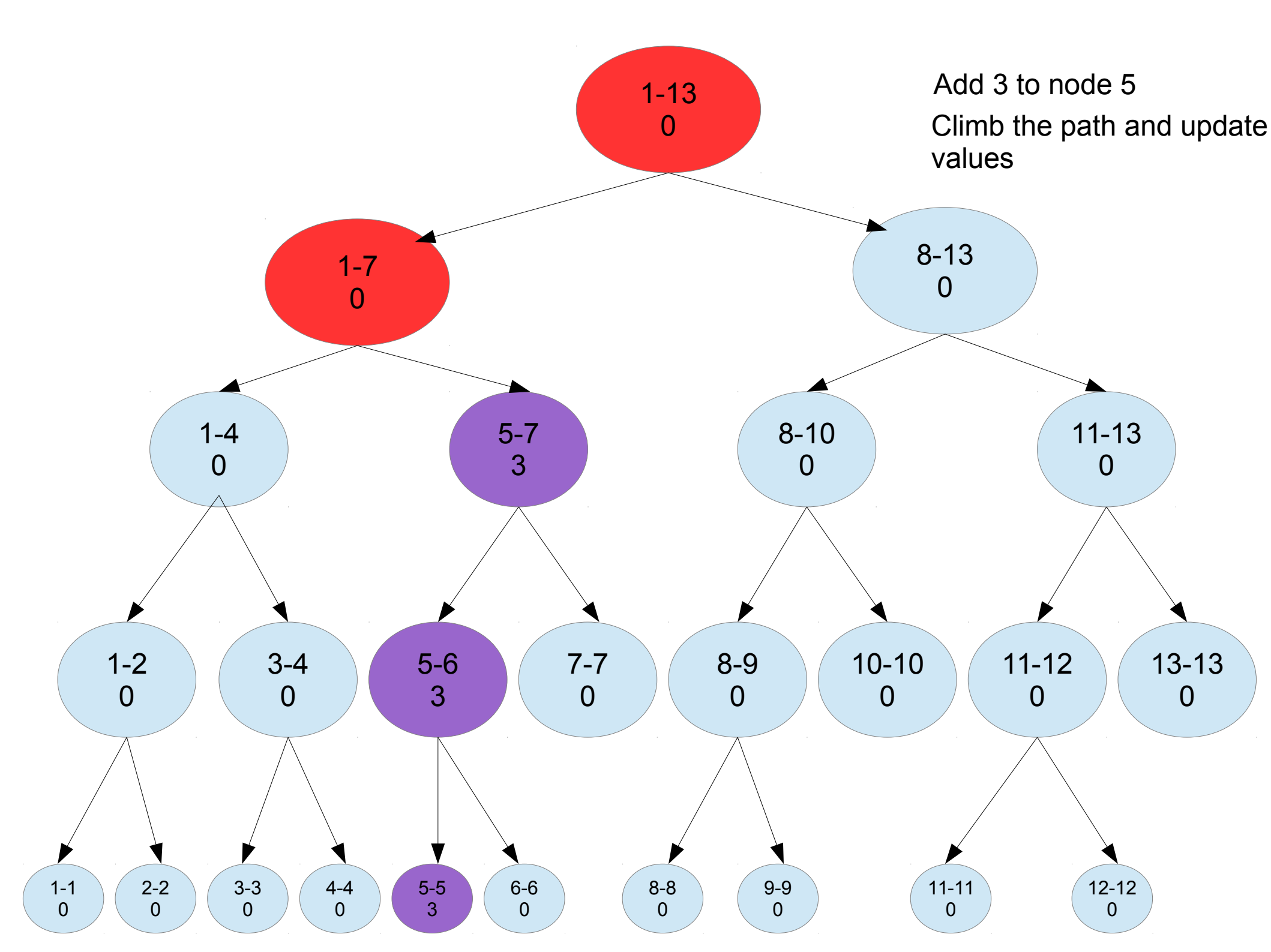

Add 3 to node 5
Follow it's path
1-13
0
1-7
0
8-13
0
1-4
0
5-7
0
8-10
0
11-13
0
1-2
0
3-4
0
5-6
0
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
0
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0

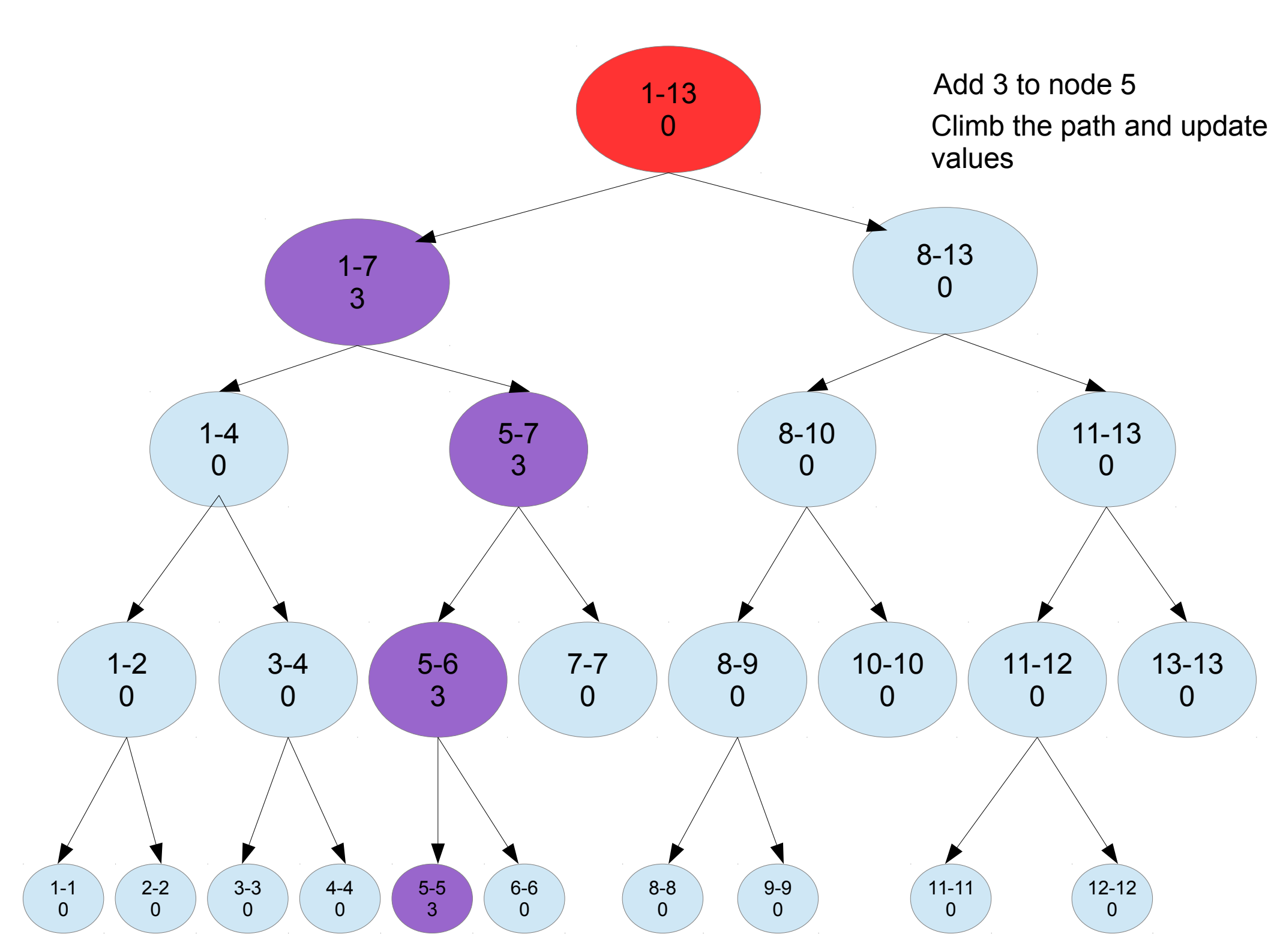

Add 3 to node 5
Follow it's path
1-13
0
1-7
0
8-13
0
1-4
0
5-7
0
8-10
0
11-13
0
1-2
0
3-4
0
5-6
0
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
0
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0

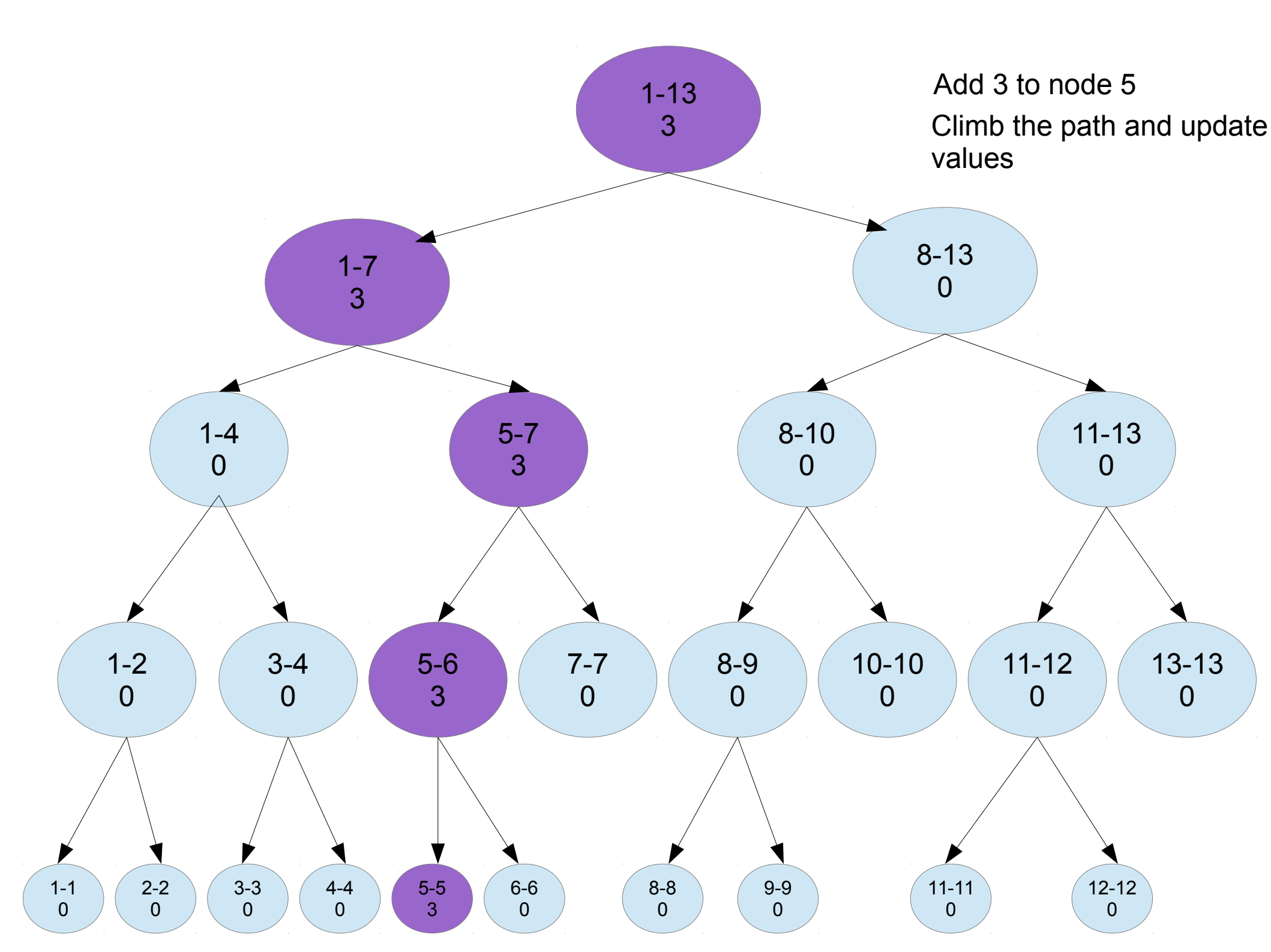

Add 3 to node 5
Follow it's path
1-13
0
1-7
0
8-13
0
1-4
0
5-7
0
8-10
0
11-13
0
1-2
0
3-4
0
5-6
0
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
0
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0


Add 3 to node 5
Reached node, update
It's value
1-13
0
1-7
0
8-13
0
1-4
0
5-7
0
8-10
0
11-13
0
1-2
0
3-4
0
5-6
0
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0


1-13
0
Add 3 to node 5
Climb the path and update values
1-7
0
8-13
0
1-4
0
5-7
0
8-10
0
11-13
0
1-2
0
3-4
0
5-6
3
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0


1-13
0
Add 3 to node 5
Climb the path and update values
1-7
0
8-13
0
1-4
0
5-7
3
8-10
0
11-13
0
1-2
0
3-4
0
5-6
3
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0


1-13
0
Add 3 to node 5
Climb the path and update values
1-7
3
8-13
0
1-4
0
5-7
3
8-10
0
11-13
0
1-2
0
3-4
0
5-6
3
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0


1-13
3
Add 3 to node 5
Climb the path and update values
1-7
3
8-13
0
1-4
0
5-7
3
8-10
0
11-13
0
1-2
0
3-4
0
5-6
3
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0

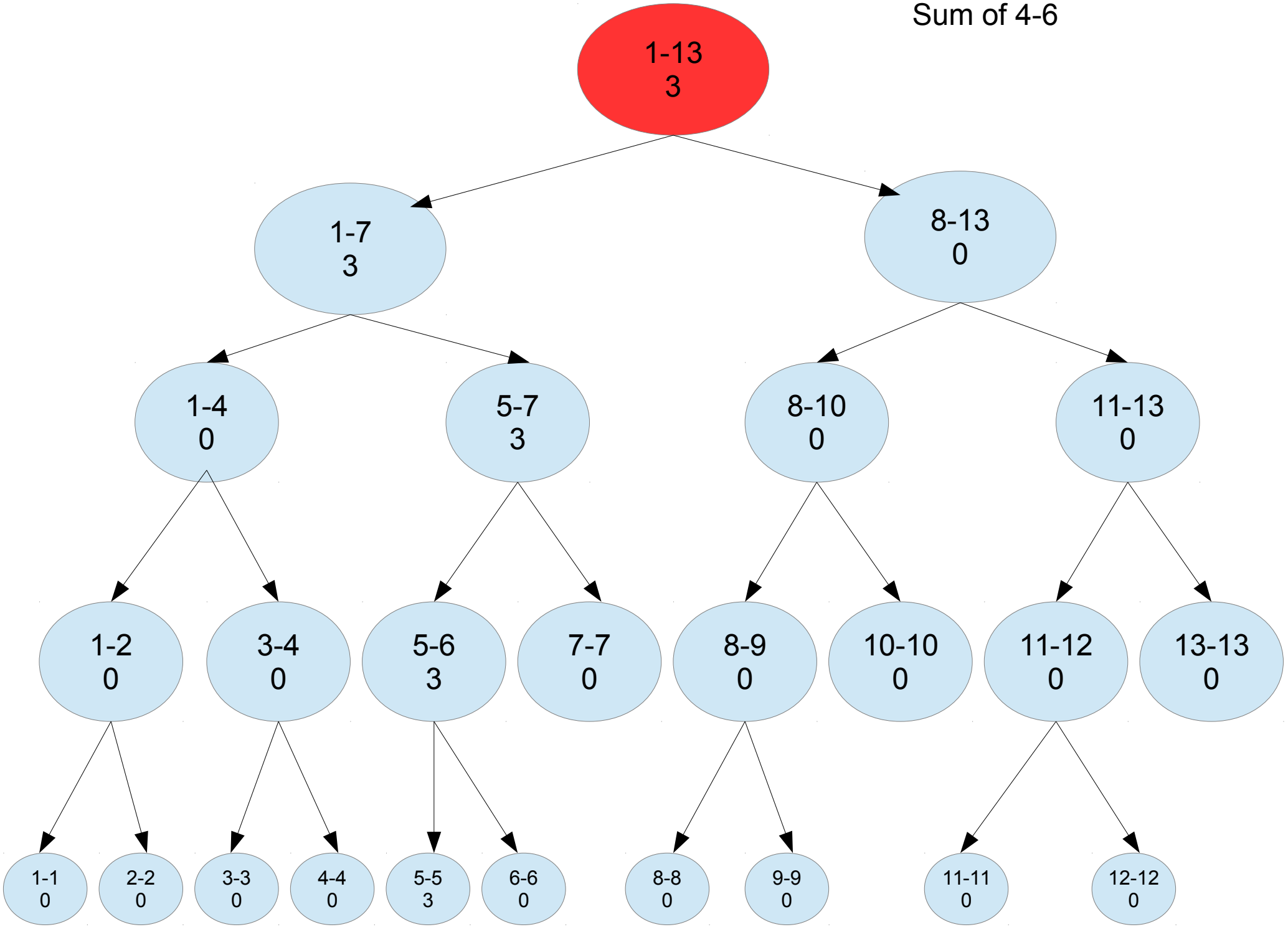
Answering a query
(in our case finding a sum)

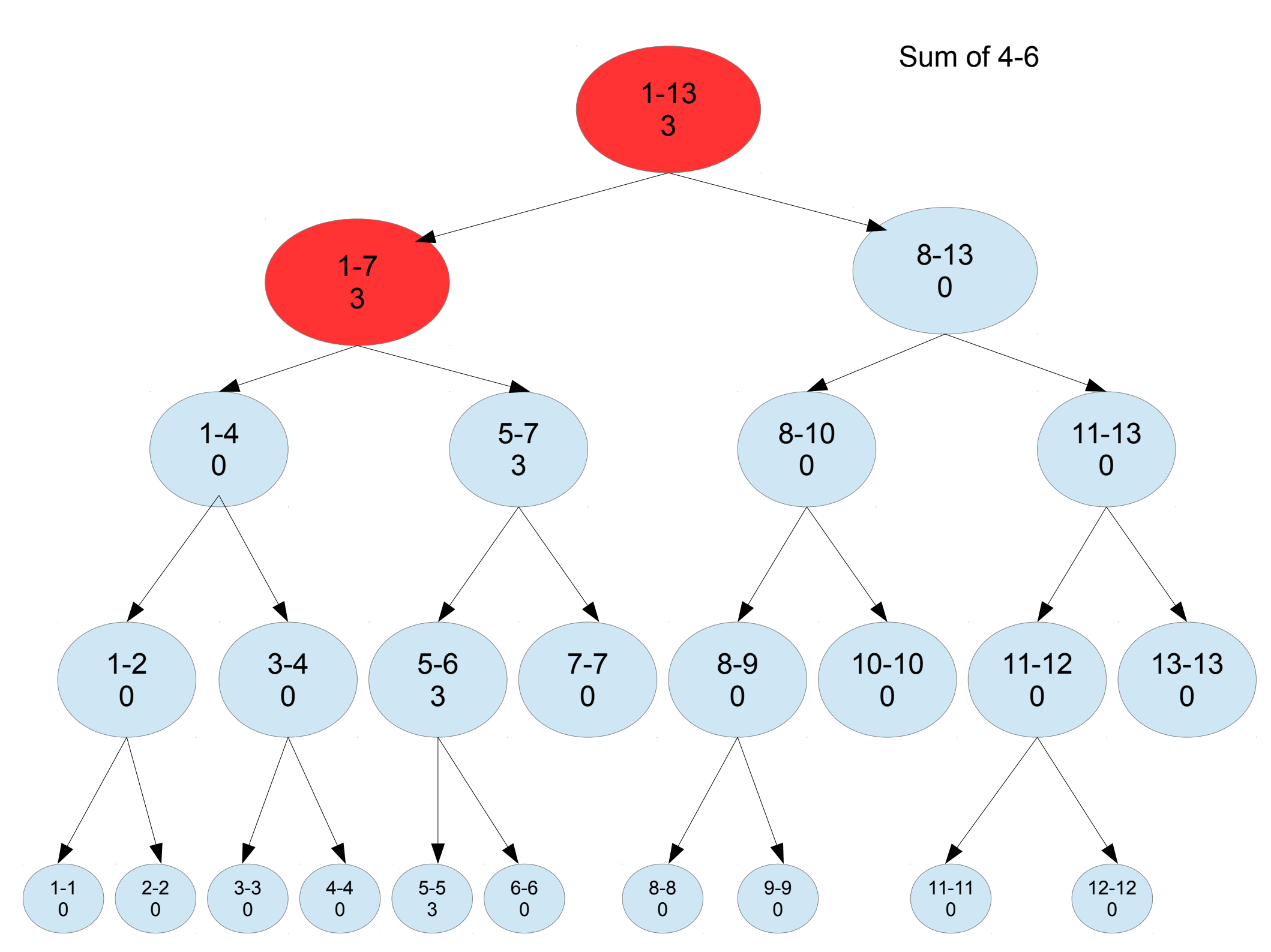
Process a node (begin with the root) :

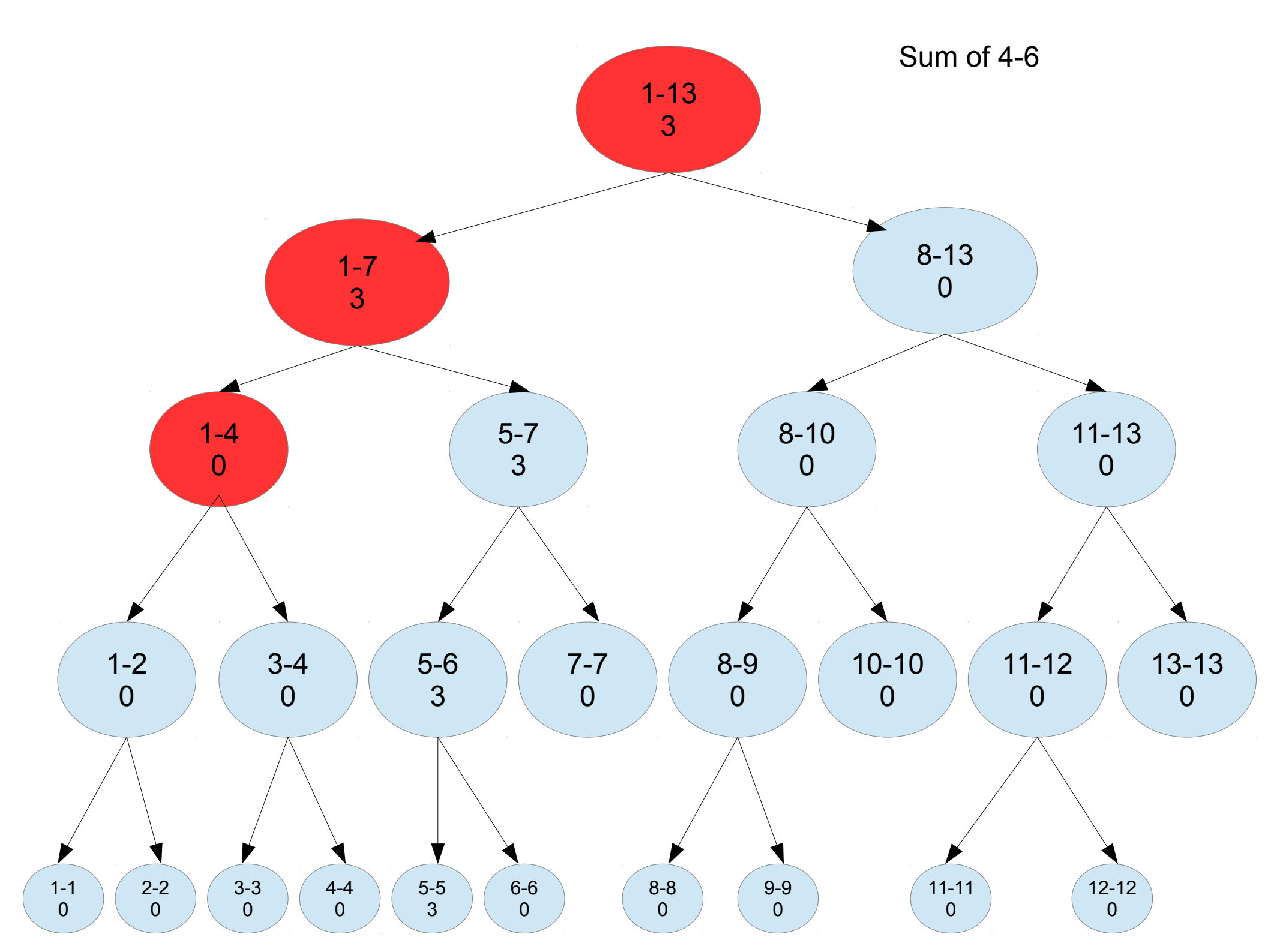
If the node completely belongs to the given interval, return the answer.

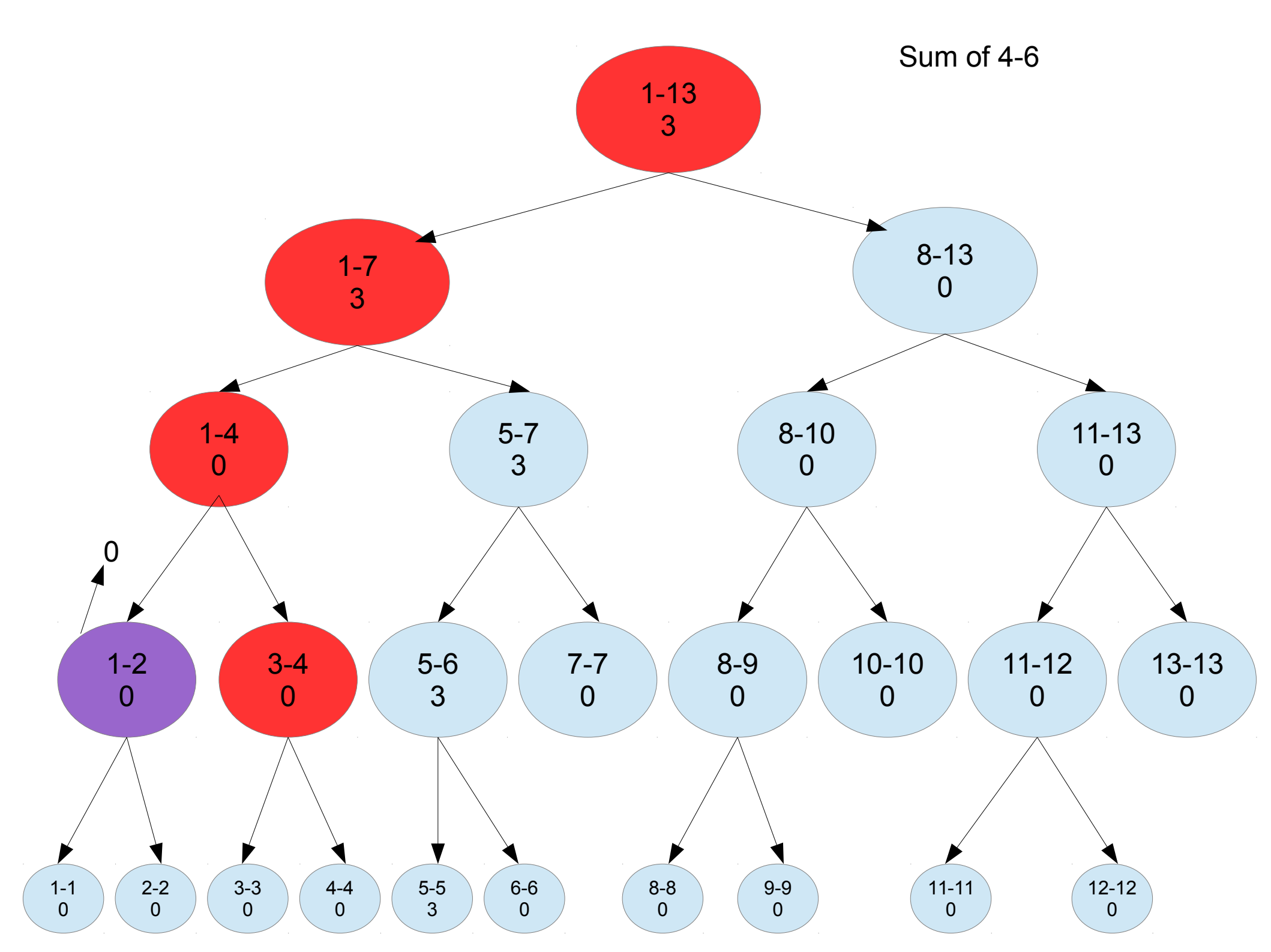
If it doesn't share a point, return a value which will not change the answer (in our case 0, in case of finding minimum, return INF, etc)

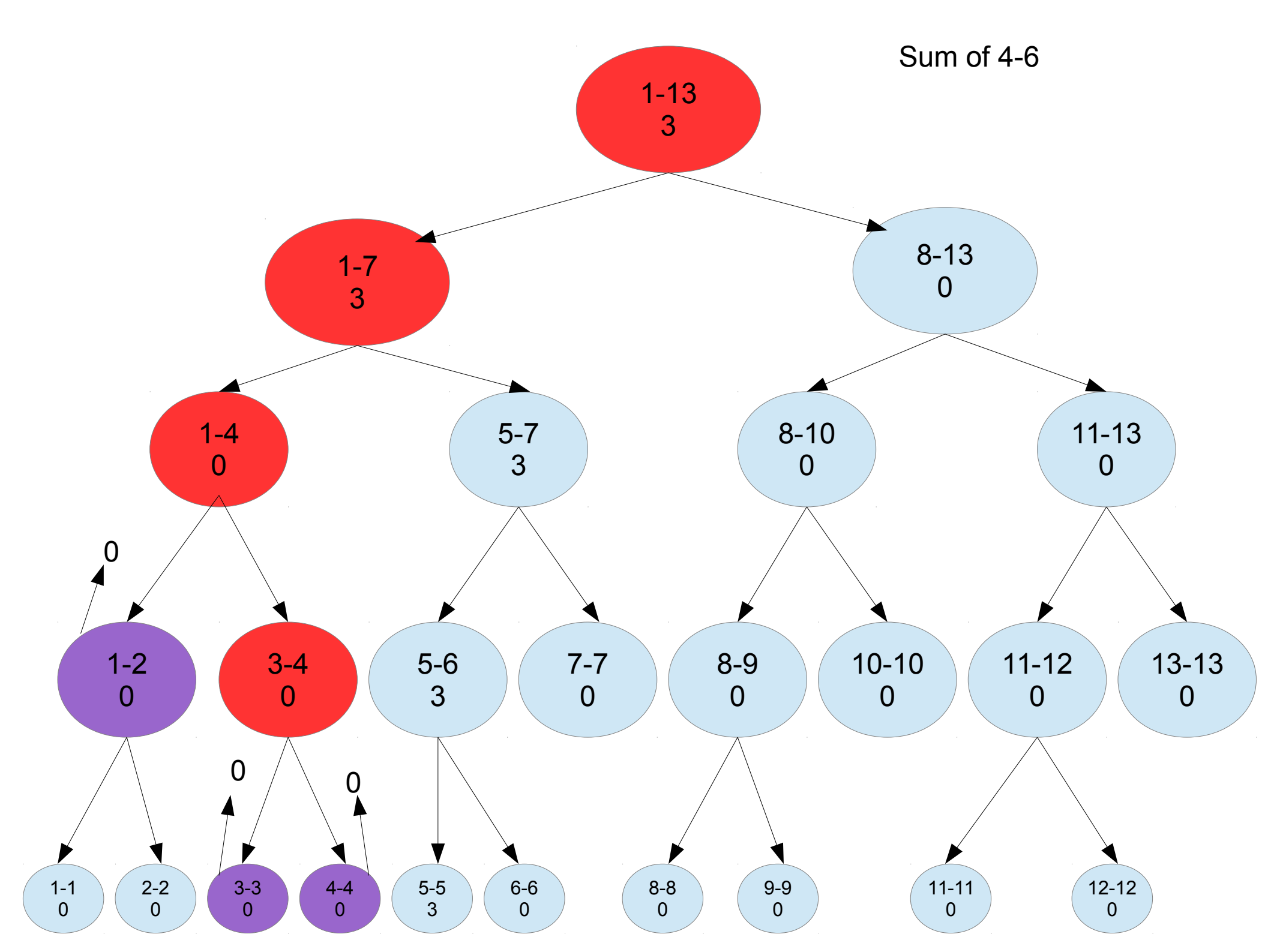
Else, recursively get the answer of it's children and combine to get the answer (in our case add them).

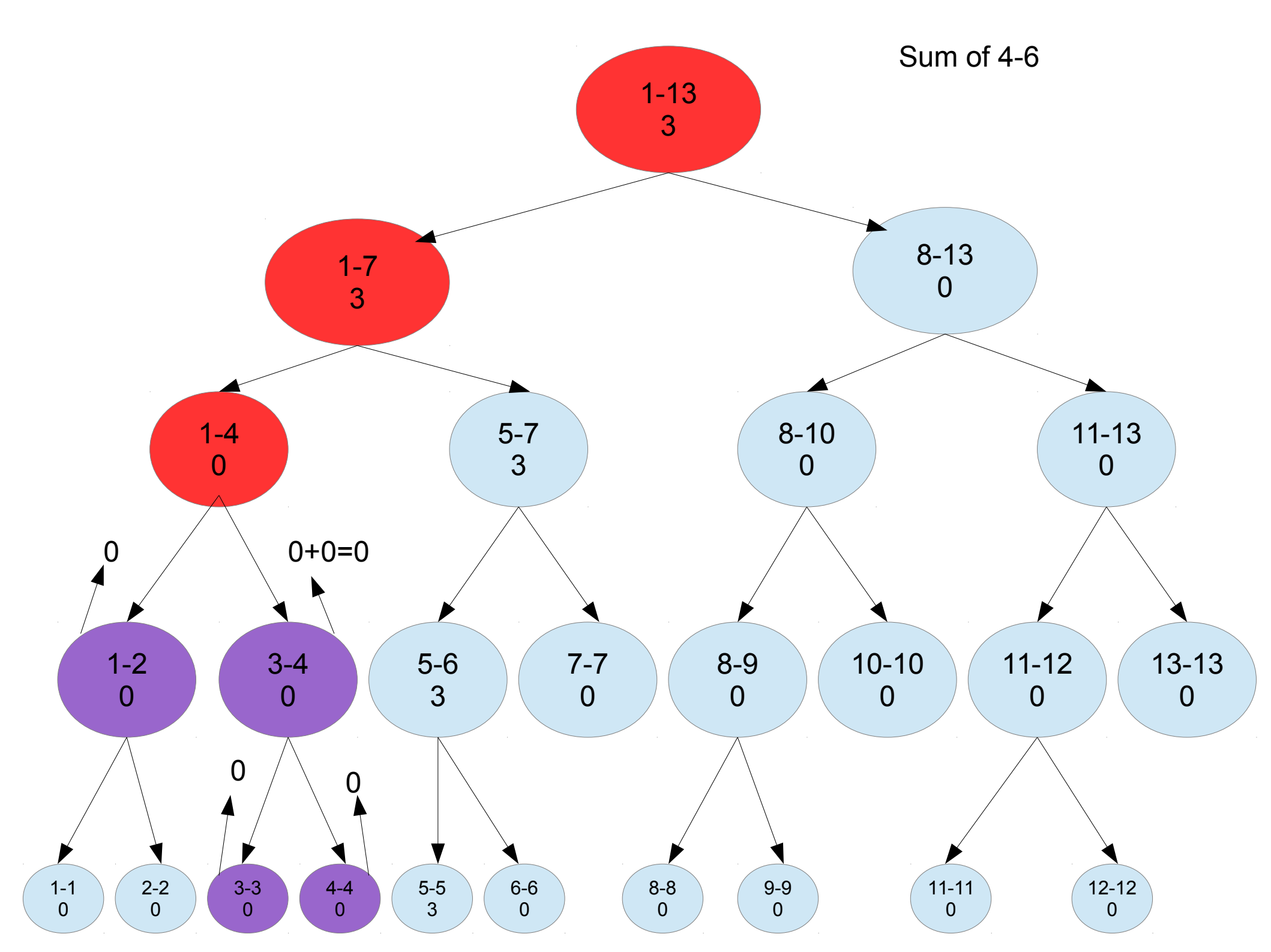

Sum of 4-6
1-13
3
1-7
3
8-13
0
1-4
0
5-7
3
8-10
0
11-13
0
1-2
0
3-4
0
5-6
3
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0

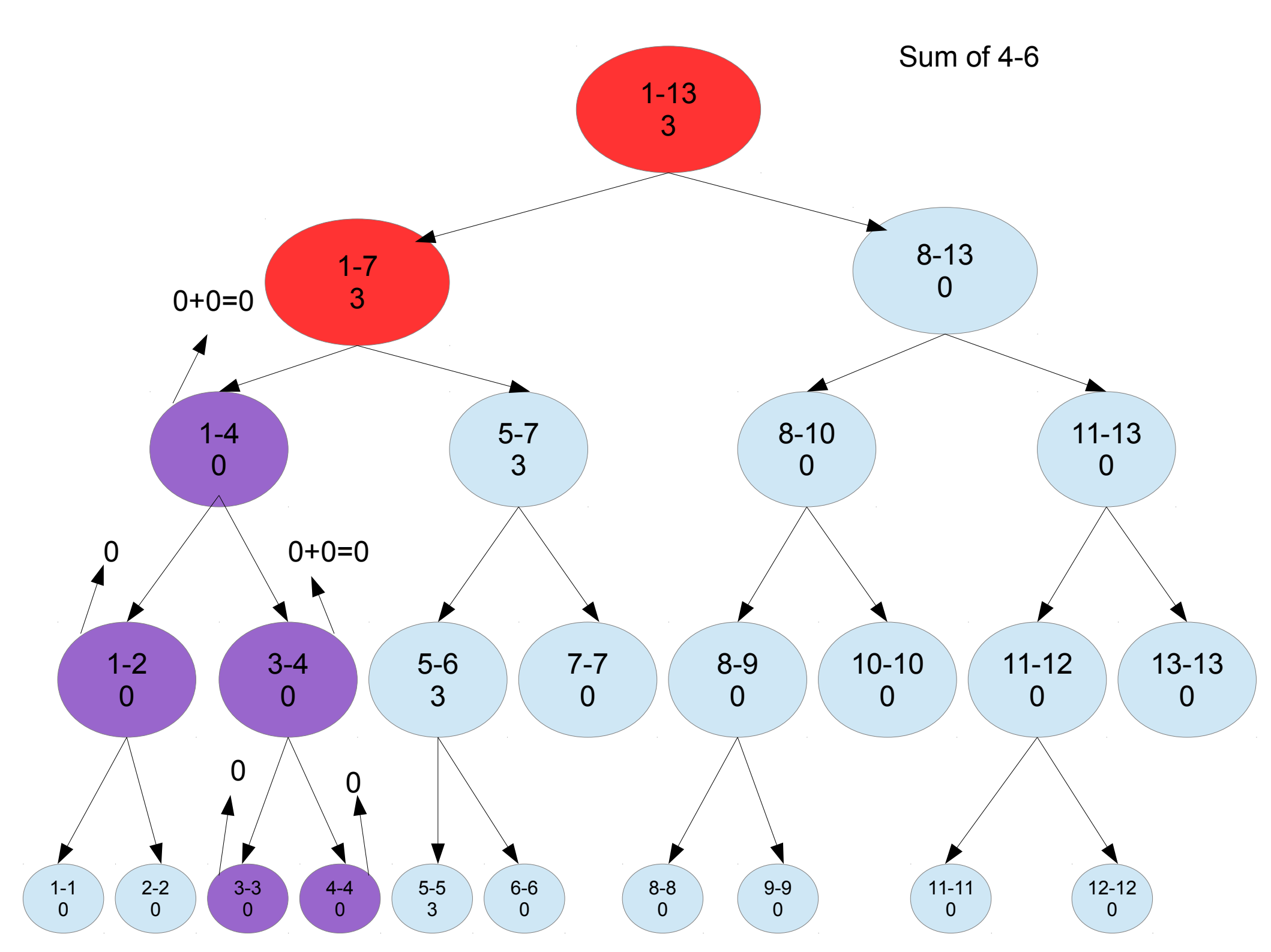

Sum of 4-6
1-13
3
1-7
3
8-13
0
1-4
0
5-7
3
8-10
0
11-13
0
1-2
0
3-4
0
5-6
3
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0

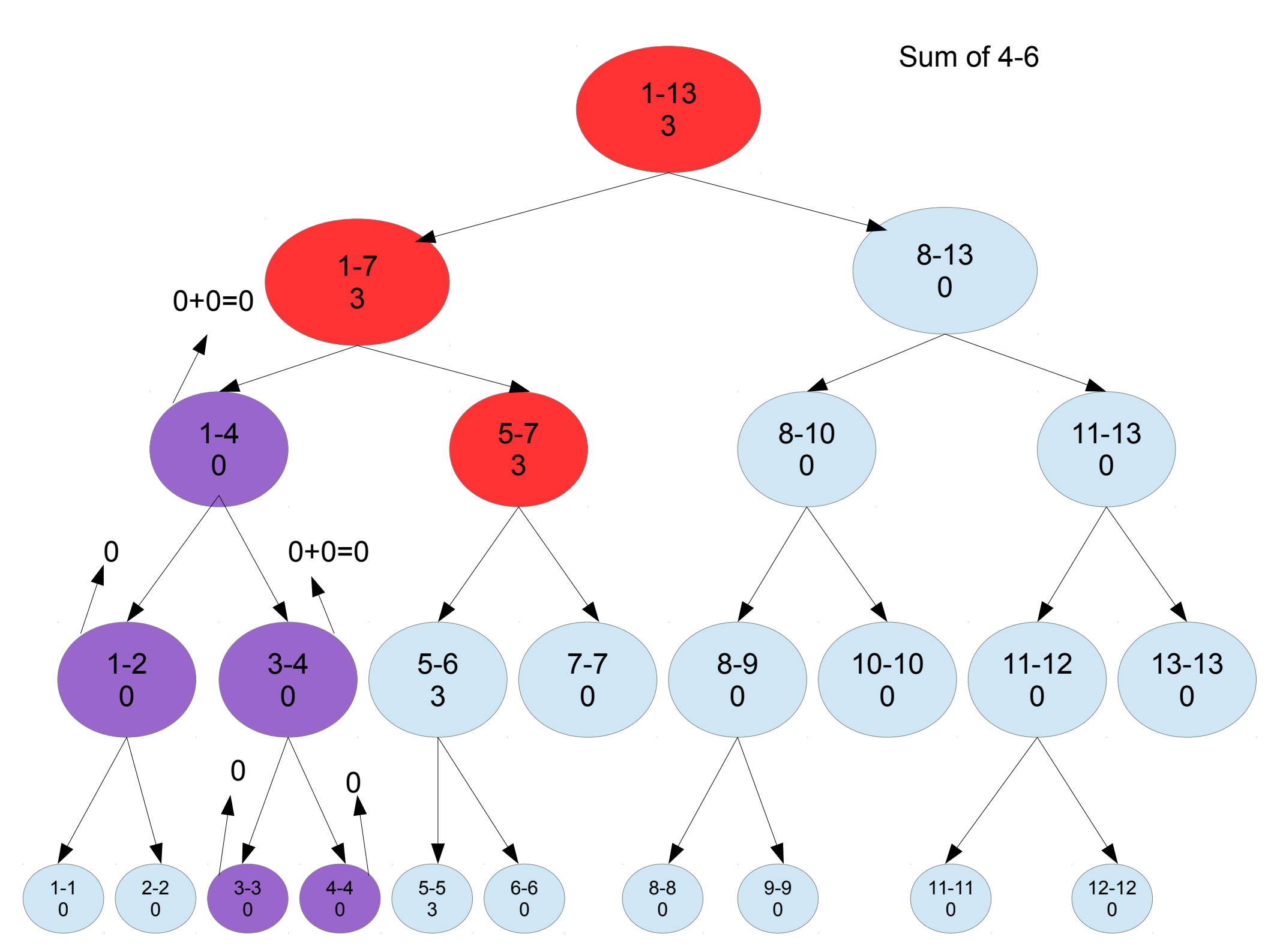

Sum of 4-6
1-13
3
1-7
3
8-13
0
1-4
0
5-7
3
8-10
0
11-13
0
1-2
0
3-4
0
5-6
3
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0

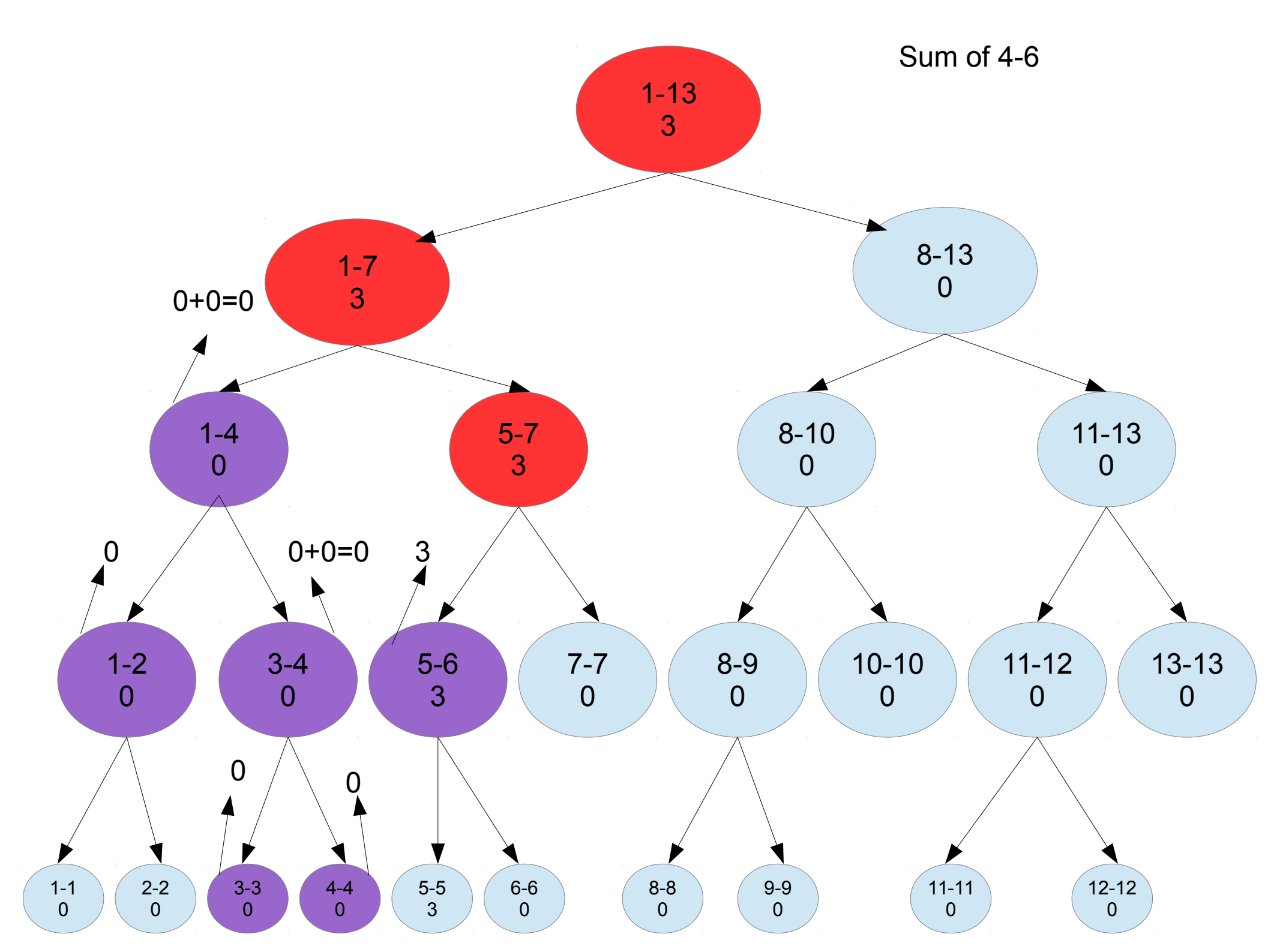

Sum of 4-6
1-13
3
1-7
3
8-13
0
1-4
0
5-7
3
8-10
0
11-13
0
0
1-2
0
3-4
0
5-6
3
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0

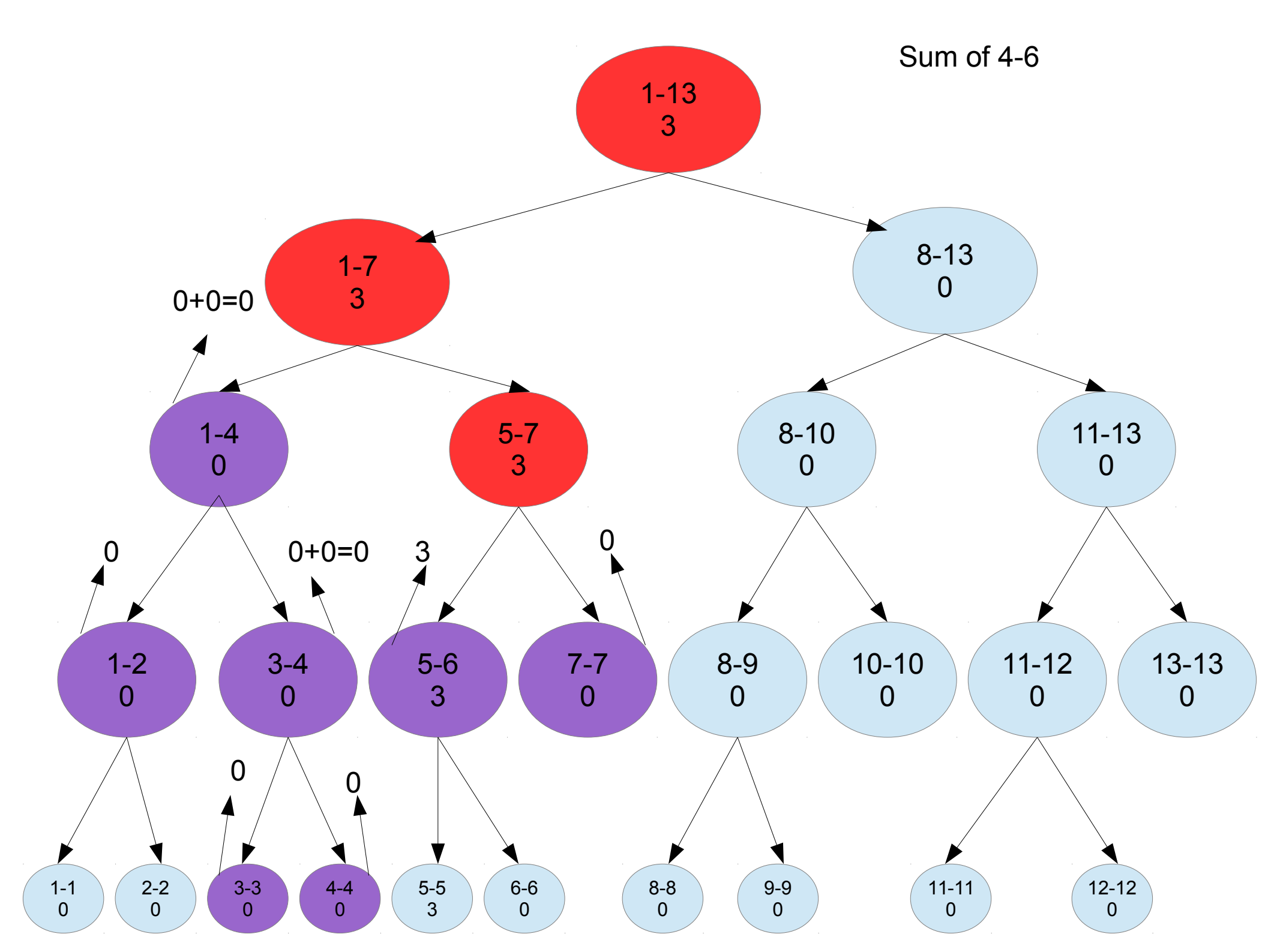

Sum of 4-6
1-13
3
1-7
3
8-13
0
1-4
0
5-7
3
8-10
0
11-13
0
0
1-2
0
3-4
0
5-6
3
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
0
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0

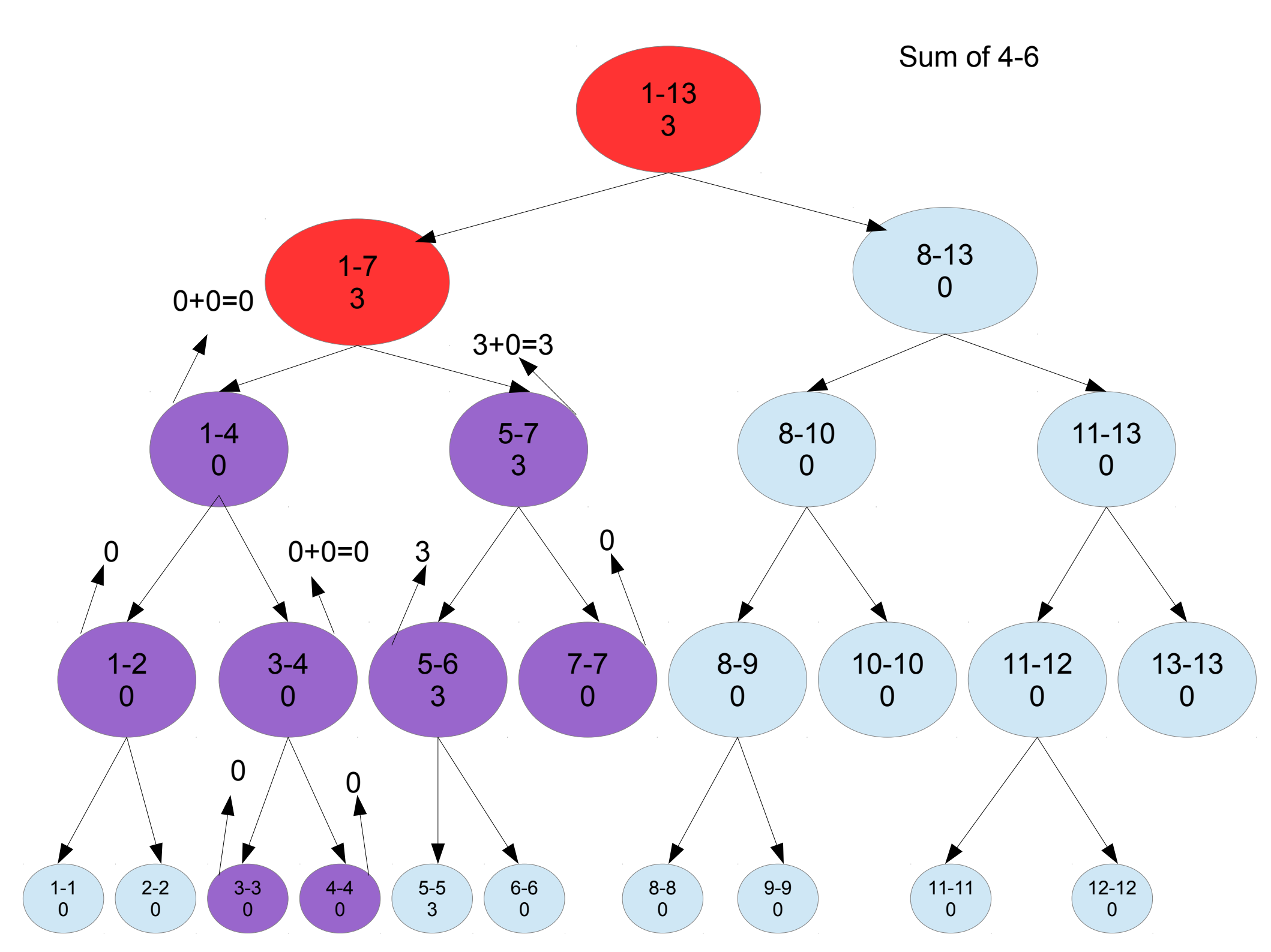

Sum of 4-6
1-13
3
1-7
3
8-13
0
1-4
0
5-7
3
8-10
0
11-13
0
0
0+0=0
1-2
0
3-4
0
5-6
3
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
0
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0

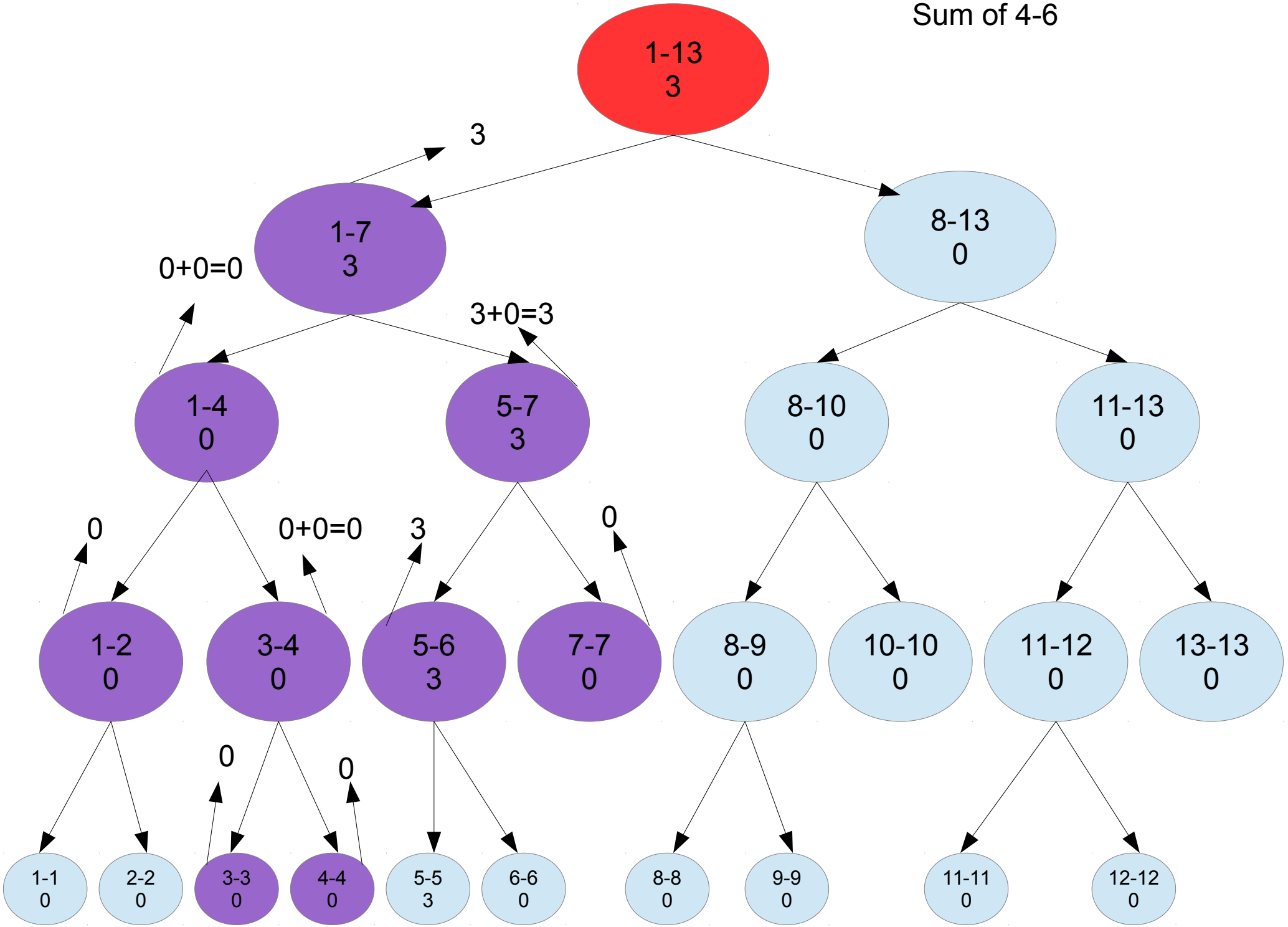

Sum of 4-6
1-13
3
1-7
3
8-13
0
0+0=0
1-4
0
5-7
3
8-10
0
11-13
0
0
0+0=0
1-2
0
3-4
0
5-6
3
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
0
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0

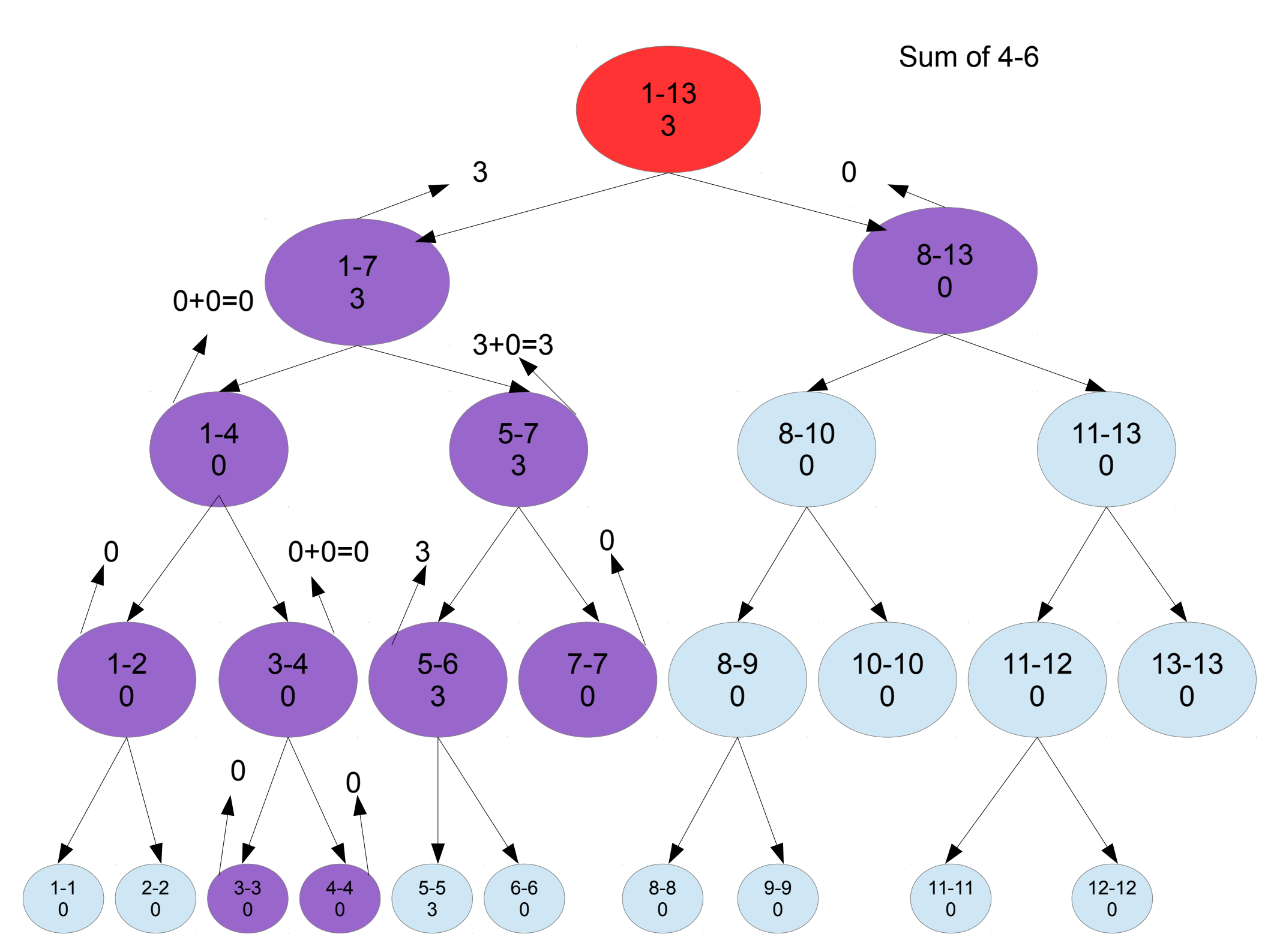

Sum of 4-6
1-13
3
1-7
3
8-13
0
0+0=0
1-4
0
5-7
3
8-10
0
11-13
0
0
0+0=0
1-2
0
3-4
0
5-6
3
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
0
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0

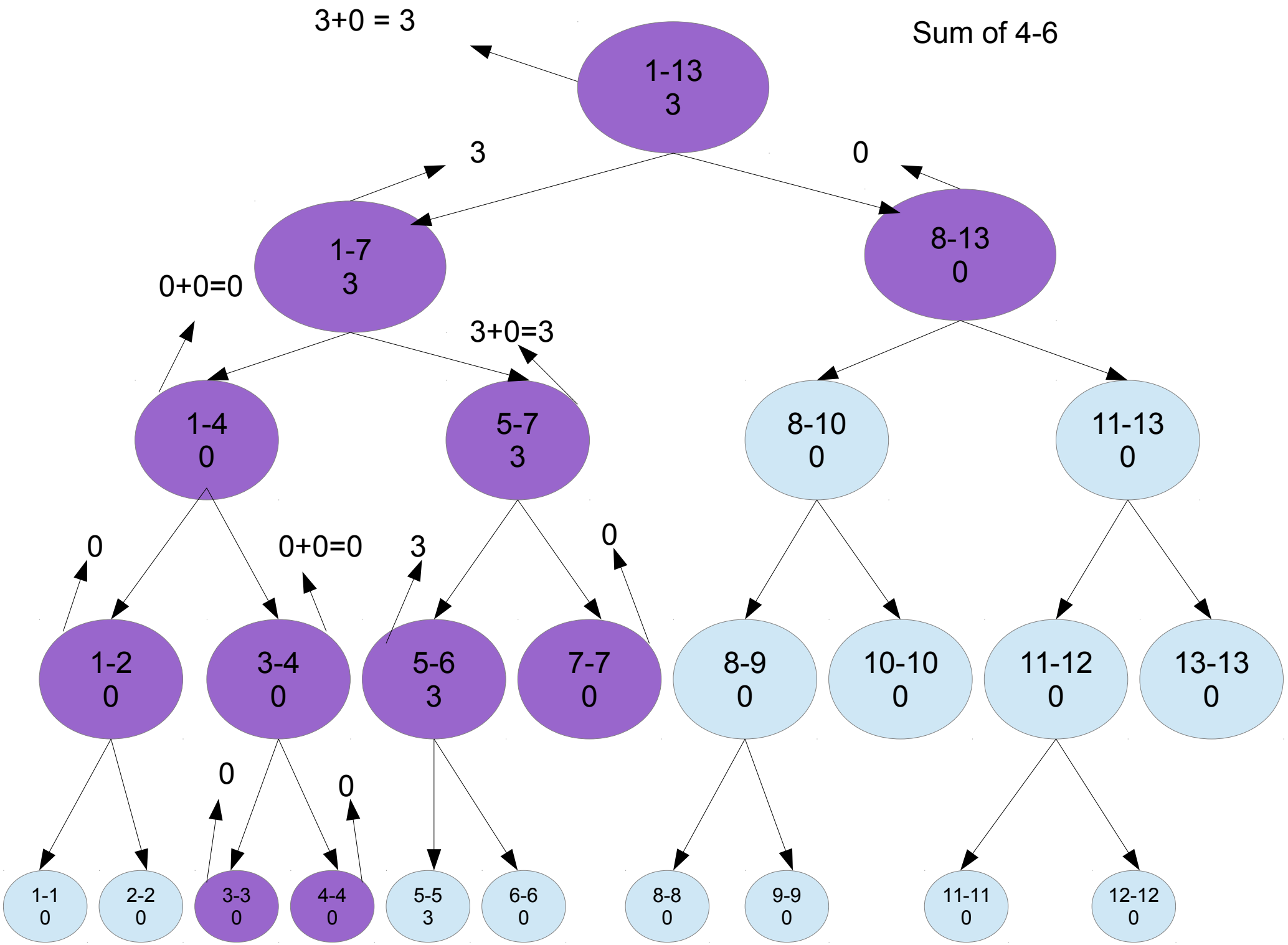

Sum of 4-6
1-13
3
1-7
3
8-13
0
0+0=0
1-4
0
5-7
3
8-10
0
11-13
0
0
0+0=0
3
1-2
0
3-4
0
5-6
3
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
0
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0


Sum of 4-6
1-13
3
1-7
3
8-13
0
0+0=0
1-4
0
5-7
3
8-10
0
11-13
0
0
0+0=0
3
0
1-2
0
3-4
0
5-6
3
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
0
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0


Sum of 4-6
1-13
3
1-7
3
8-13
0
0+0=0
3+0=3
1-4
0
5-7
3
8-10
0
11-13
0
0
0+0=0
3
0
1-2
0
3-4
0
5-6
3
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
0
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0


Sum of 4-6
1-13
3
3
1-7
3
8-13
0
0+0=0
3+0=3
1-4
0
5-7
3
8-10
0
11-13
0
0
0+0=0
3
0
1-2
0
3-4
0
5-6
3
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
0
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0


Sum of 4-6
1-13
3
3
0
1-7
3
8-13
0
0+0=0
3+0=3
1-4
0
5-7
3
8-10
0
11-13
0
0
0+0=0
3
0
1-2
0
3-4
0
5-6
3
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
0
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0


3+0 = 3
Sum of 4-6
1-13
3
3
0
1-7
3
8-13
0
0+0=0
3+0=3
1-4
0
5-7
3
8-10
0
11-13
0
0
0+0=0
3
0
1-2
0
3-4
0
5-6
3
7-7
0
8-9
0
10-10
0
11-12
0
13-13
0
0
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0

Complexity : O (logN)

To prove it, we first observe these :

Lemma 1: Following a path from the root to a leaf visits O(logN) nodes.

Everytime, the interval is divided by two, so after visiting x descendants, the interval is [N / (2^x)] units long. We stop when the interval reaches 1 unit long (reaches a leaf), which means
N / (2^x) == 1 <=>
N = 2^x <=>
x = logN

Lemma 2 : If we are at a node V [x-y] and we ask for an interval [qx,qy], where x=qx, we get the answer in O (logN).

Let v1 = Left child of V, v2 = Right child of V

Suppose qy is smaller than or equal to v1's ending point.
Then v2 will return 0 without visiting it's descendants, so we only visit v1.

Else if qy is greater than v1's ending point,
Then v1 will return it's value in constant time without visiting it's descendants, and we only visit v2.

Both ways, we only visit one descendant, so according to Lemma 1,
O(logN) nodes are visited. Since we process each one in O(1) time,
O(logN) time is required in total.

Lemma 3 : If we are at a node V [x-y] and we ask for an interval and we ask for an interval [qx,qy], where x=qx, we get the answer in O (logN).

It is proved exactly the same way with Lemma 2.

Complexity Proof :

Starting from the root, we are visiting only one descendant k times (possibly k=0). If we never split our search to two descendants, O (logN) time is needed according to Lemma 1.

Else, we have the interval [x,y] and we ask for [qx,qy]. We split to :
v1 [x, (x+y)/2] and we ask for [qx, (x+y)/2]
v2 [(x+y)/2 + 1, y] and we ask for [(x+y)/2 + 1, qy]

According to Lemma 2 and 3, both of these searches will need O (logN), so total complexity is :
O(k) + 2 * O (logN), where k is obviously smaller than logN.

This gives us O(logN) nodes visited, O(1) time in each node, so O(logN) time.

Conclusion : Every interval can be seen as O (logN) independent intervals of a segment tree, so total complexity for every query will be O (logN*qt), where qt is the time needed to get the answer from every node (in our case this is constant).

Other examples (where qt differs) :

Find how many elements in a given interval are less than K.

Obviously, to answer this query we need to sort our elements.
In every node, we keep our elements sorted (this can be done using the Merge-Sort algorithm).

To answer this query for one of our nodes, we simply binary search for the answer, so qt=logN.

By using the techiques described before, we can make the whole algorithm run in O (log^2 (N)) per query.

One more interesting technique, tree linearization, can be seen here :
http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=lowestCommonAncestor
(in the “Reduction from LCA to RMQ” section).
The only difference is that we will write every node only once instead of every time we visit it.

Given a node of a rooted tree, with this technique, all of it's descendants are in a continuous interval, so you can use a segment tree to answer queries of the form :
If every node has a label, find the sum of the labels of the nodes on the subtree rooted at v,
Or find the minimum one, etc.

Recommended problems
(easier to harder in my opinion) :

http://www.spoj.pl/problems/DCEPC206/
http://www.spoj.pl/problems/GSS1/
http://www.spoj.pl/problems/GSS3/
http://www.spoj.pl/problems/GSS5/
https://www.spoj.pl/problems/FREQUENT/
https://www.spoj.pl/problems/PATULJCI/
https://www.spoj.pl/problems/ORDERS/
http://www.spoj.pl/problems/CTRICK/
http://www.spoj.pl/problems/BRCKTS/
https://www.spoj.pl/problems/RATING/
http://www.spoj.pl/problems/COWPIC/
http://www.spoj.pl/problems/INCSEQ/
https://www.spoj.pl/problems/INCDSEQ/
http://www.spoj.pl/problems/PT07J/
http://www.spoj.pl/problems/TRAPEZBO/

Updating an interval
(in our case adding)

The main idea is keeping an extra value
(lazy value).
When we encounter a node whose interval equals the asked one, we update it's value,

keep in the lazy value the value we should update it's children with,

and return.

For example :
Update (1,N,5) (Add 5 to every node)
We add to the root's value 5*N,
And 5 to it's Lazy value.

Update (1,N,3)
We add to the roots value 3*N (now it has 8*N)
And 3 to it's Lazy value (now it has Lazy value 8).

This equals that for every node v below, we should Update (v, 8).

Why this works?

As in the standard update function, every node whose ancestors have no lazy values has the correct value in it.

If we ask for the value of a node whose ancestors have lazy values, we can (at that time), make them have no lazy values by propagating it.

This means that when you are in a node which has a lazy value in it, and want to go deeper in this path, this node should have no lazy value in it so that you get the correct results.

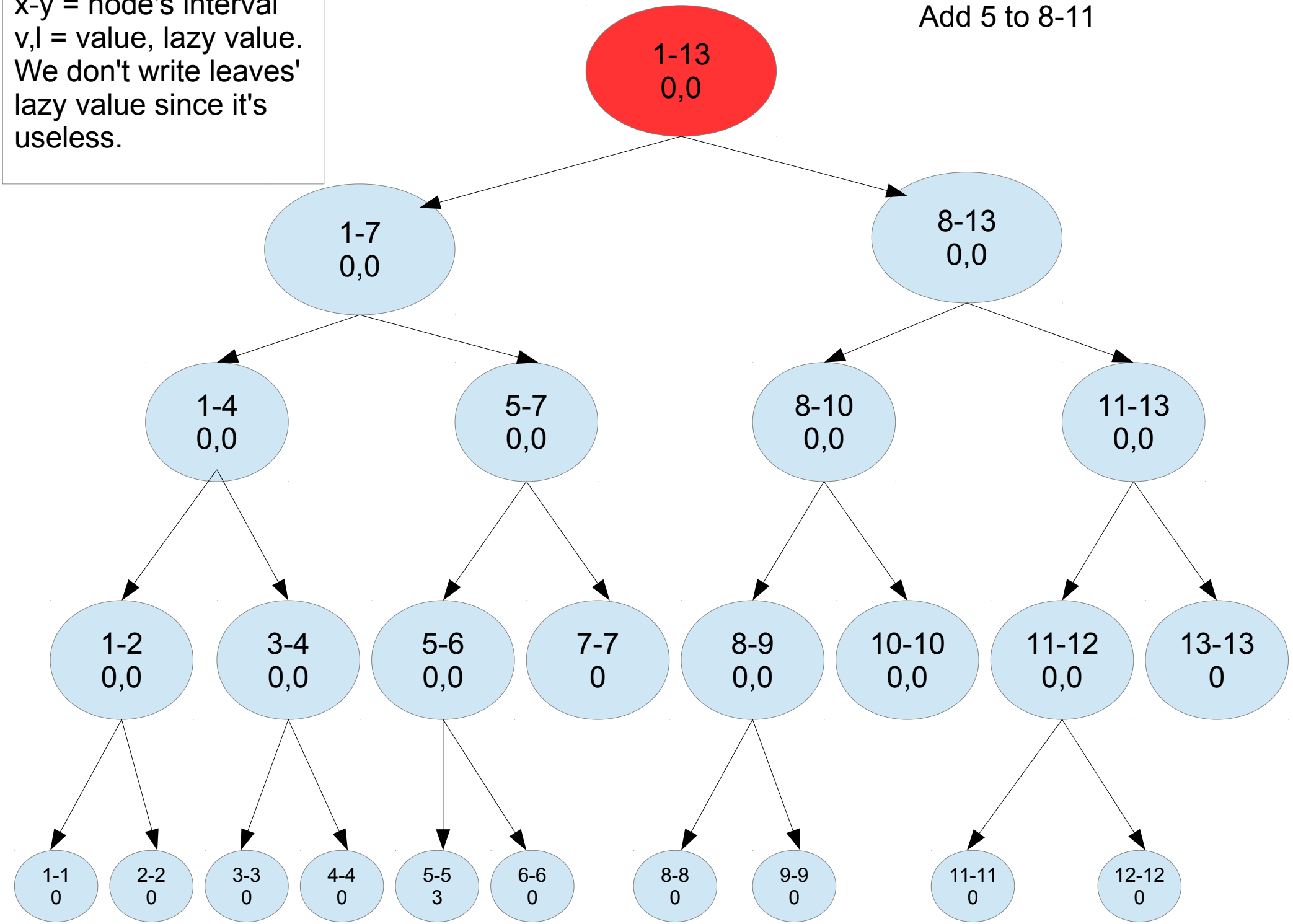
But this is easy. You just send the lazy value to both children, and set your own to zero.

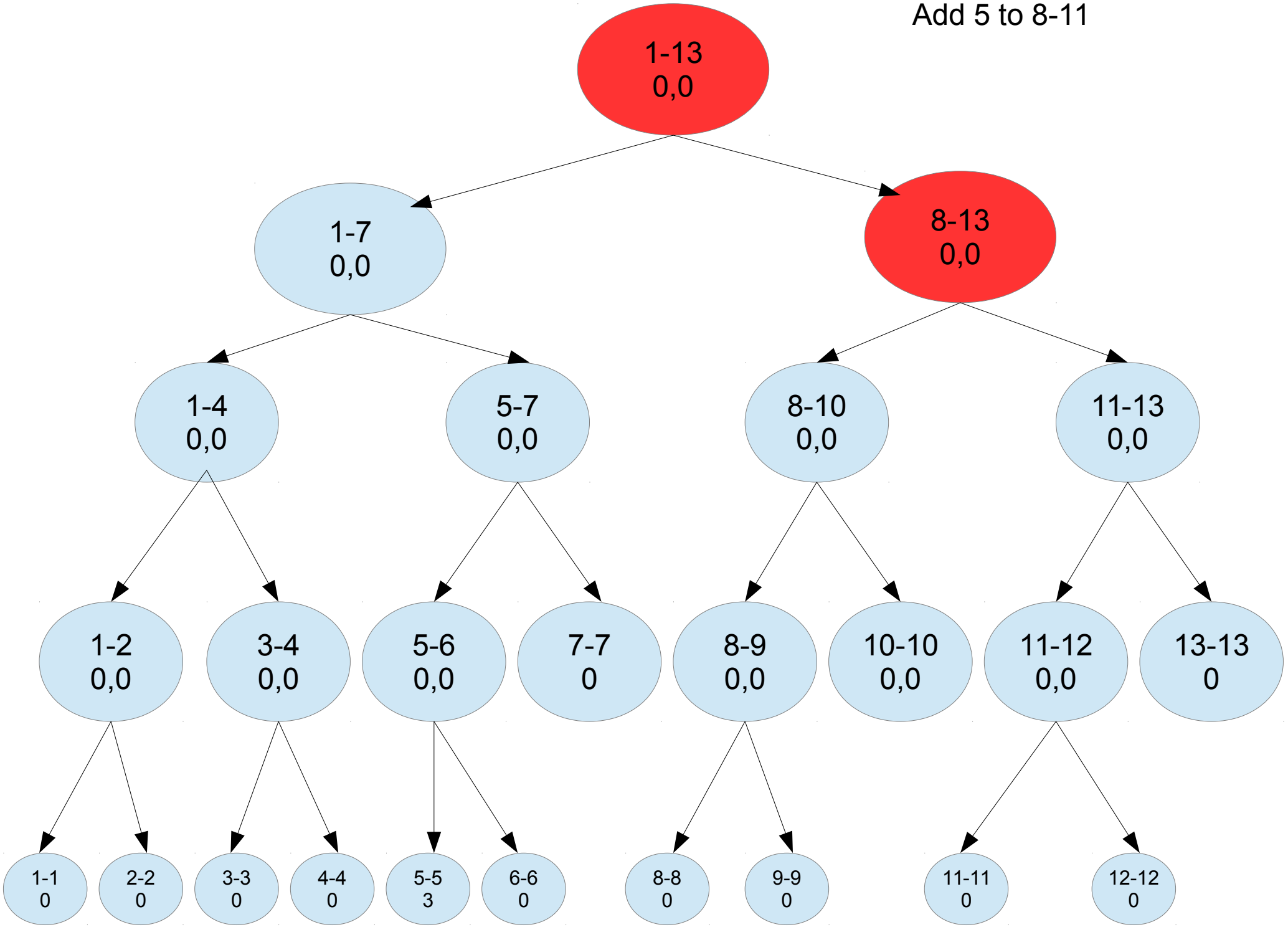
In the previous example, you send the lazy value to the two children of the root (while updating their real value) and set the root's lazy value to zero.

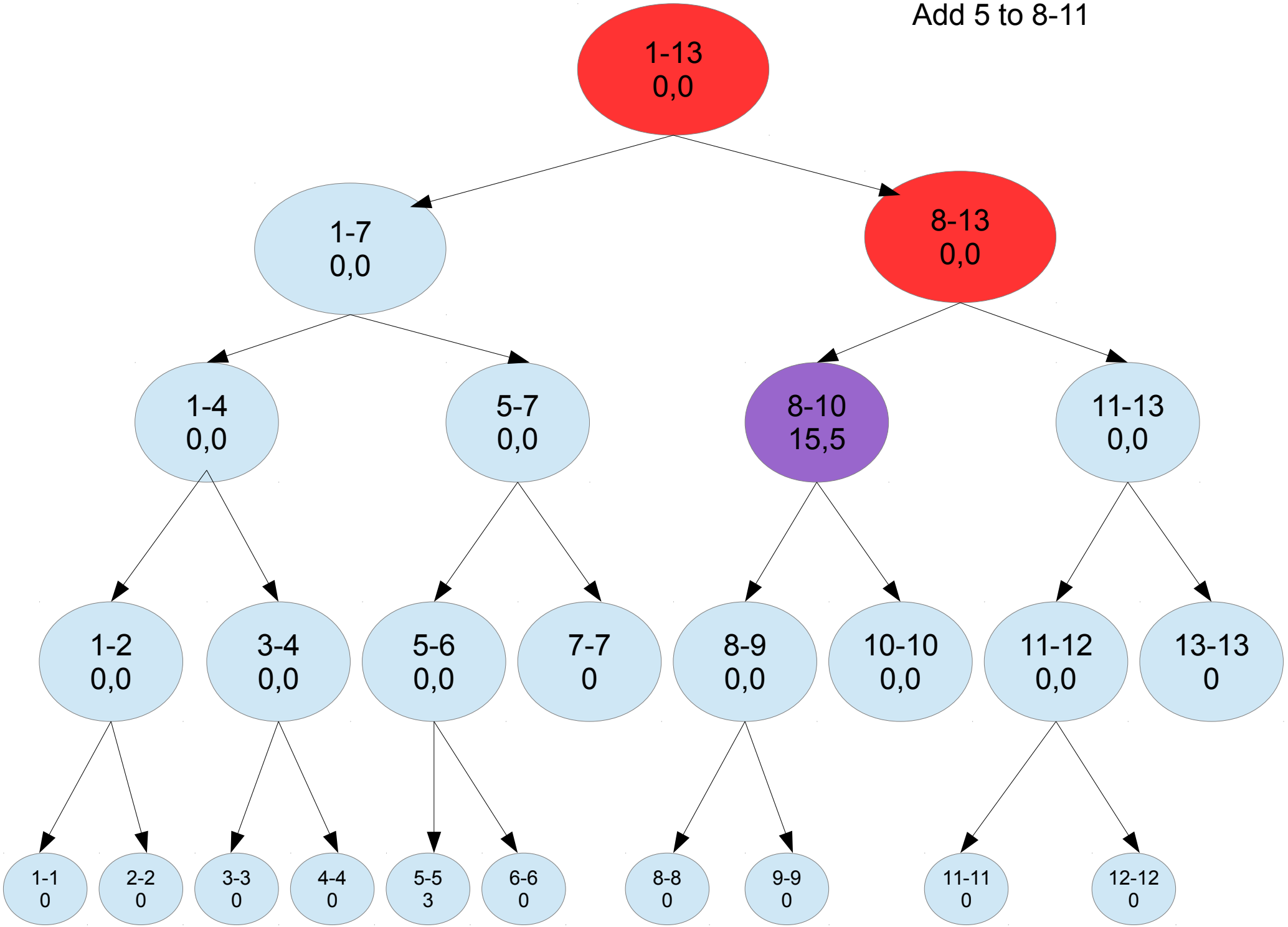
By doing so, you always send the lazy value one level deeper, until it reaches a leaf, where it has no meaning since it has no nodes below it.

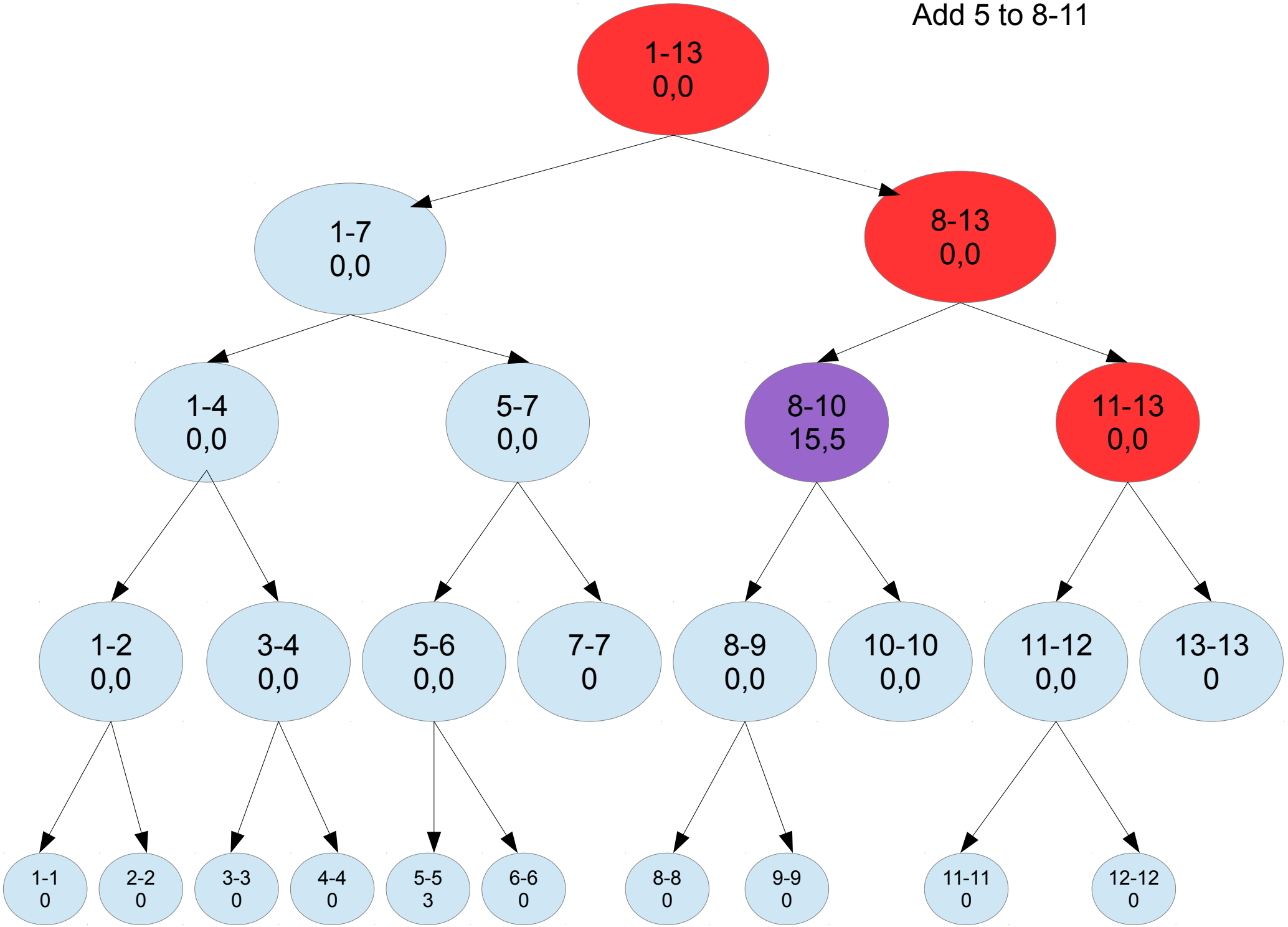
x-y = node's interval
v,l = value, lazy value.
We don't write leaves' lazy value since it's useless.

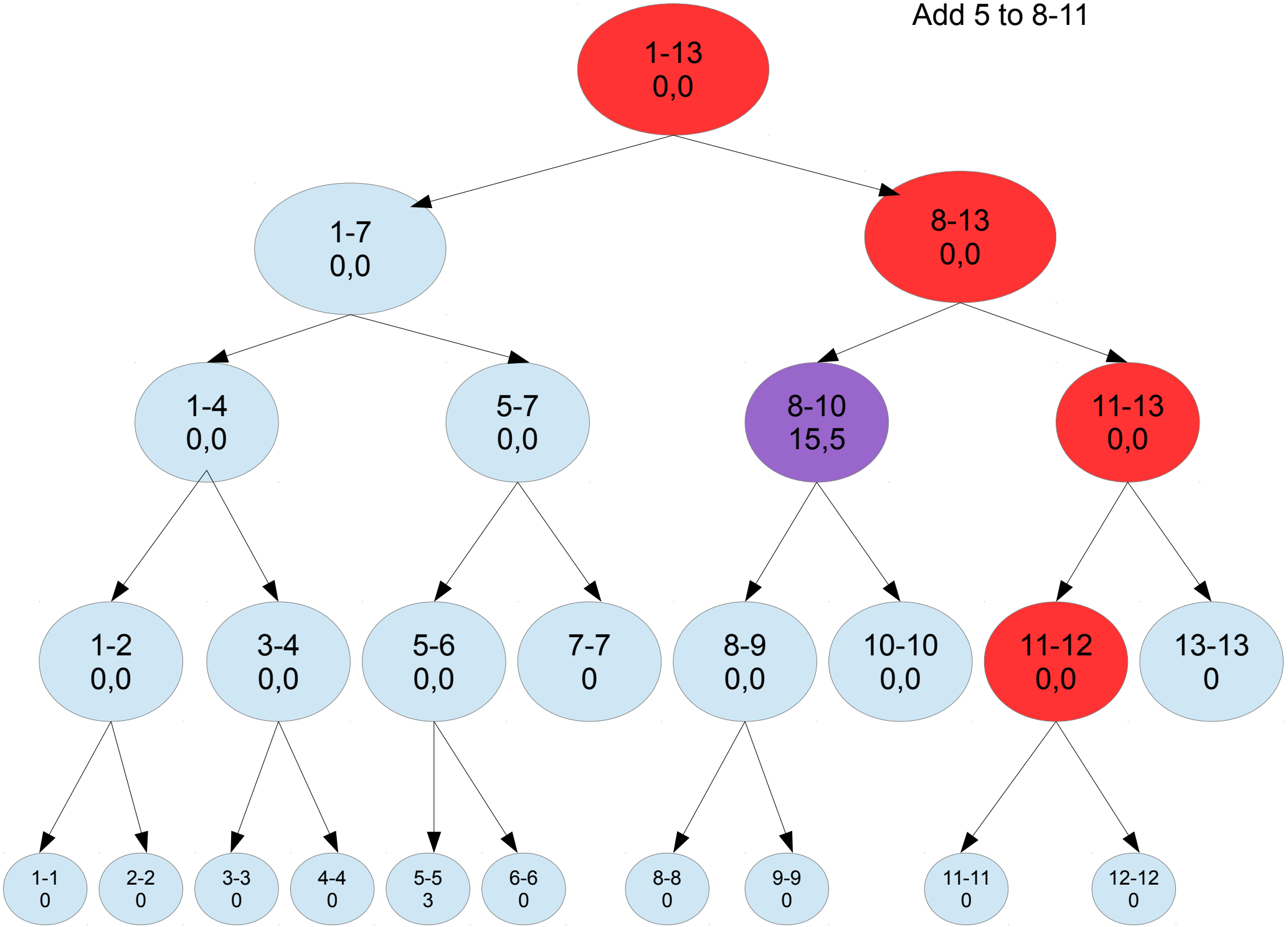
Add 5 to 8-11

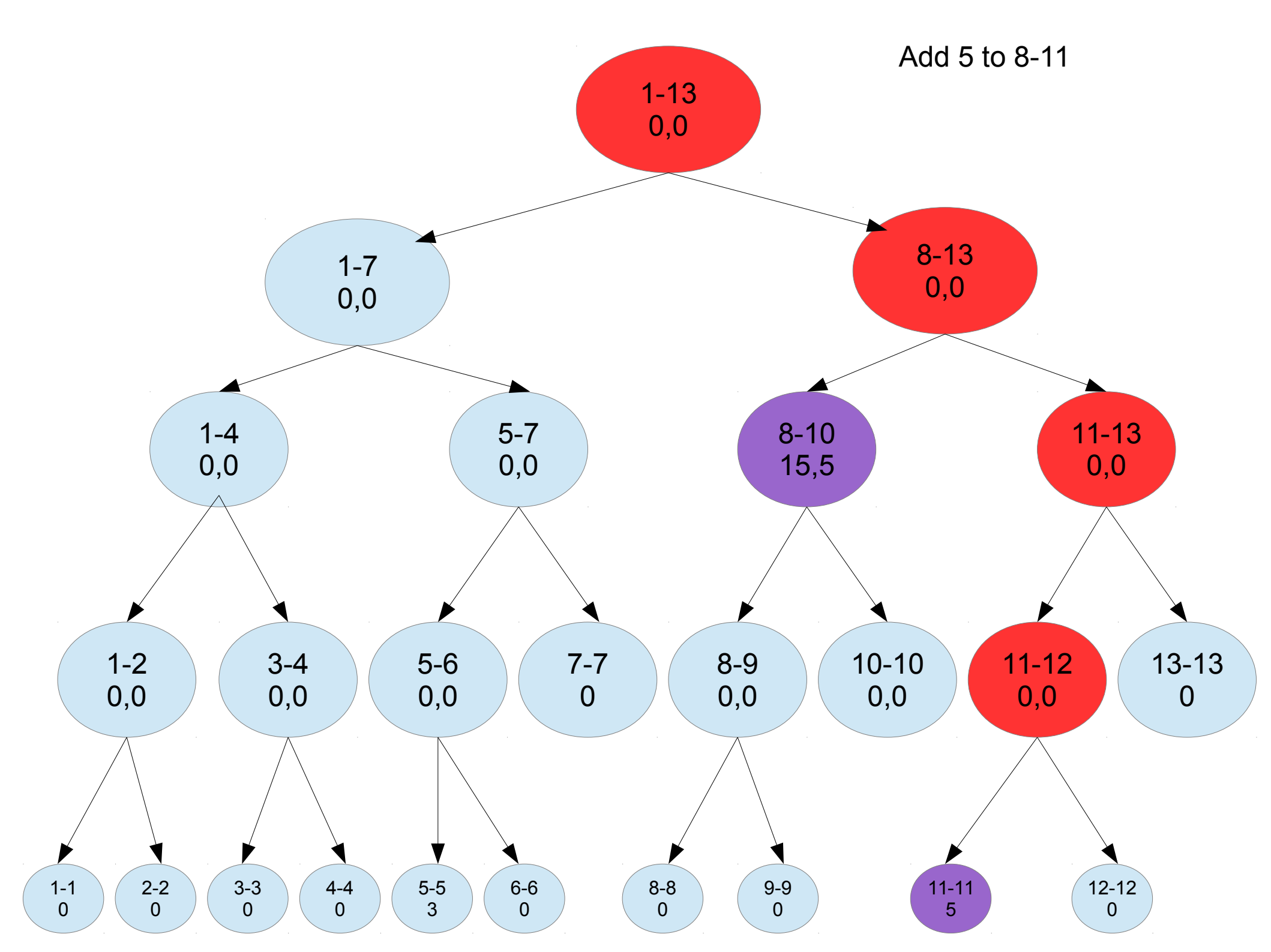


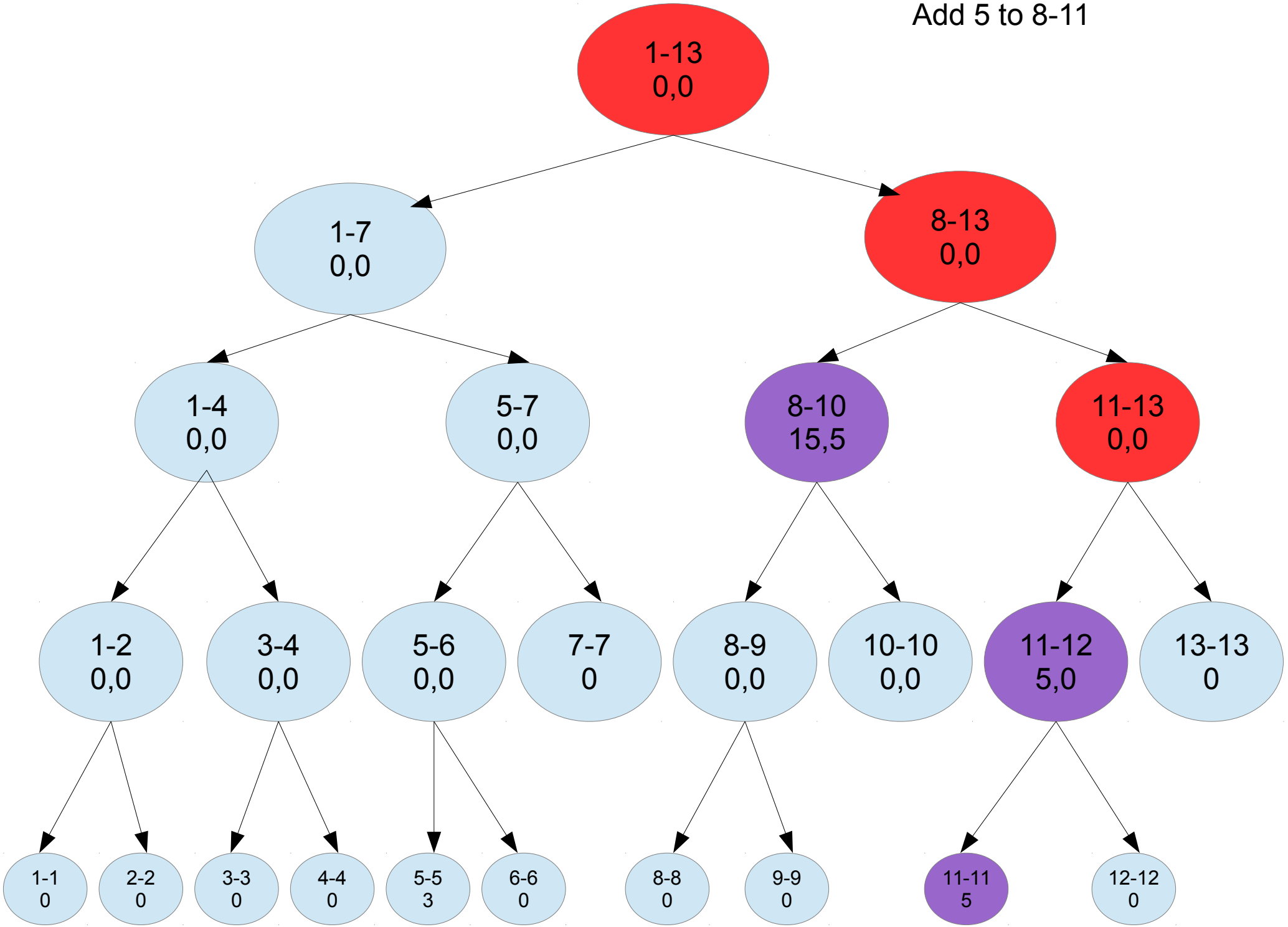

Add 5 to 8-11
1-13
0,0
1-7
0,0
8-13
0,0
1-4
0,0
5-7
0,0
8-10
0,0
11-13
0,0
1-2
0,0
3-4
0,0
5-6
0,0
7-7
0
8-9
0,0
10-10
0,0
11-12
0,0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0

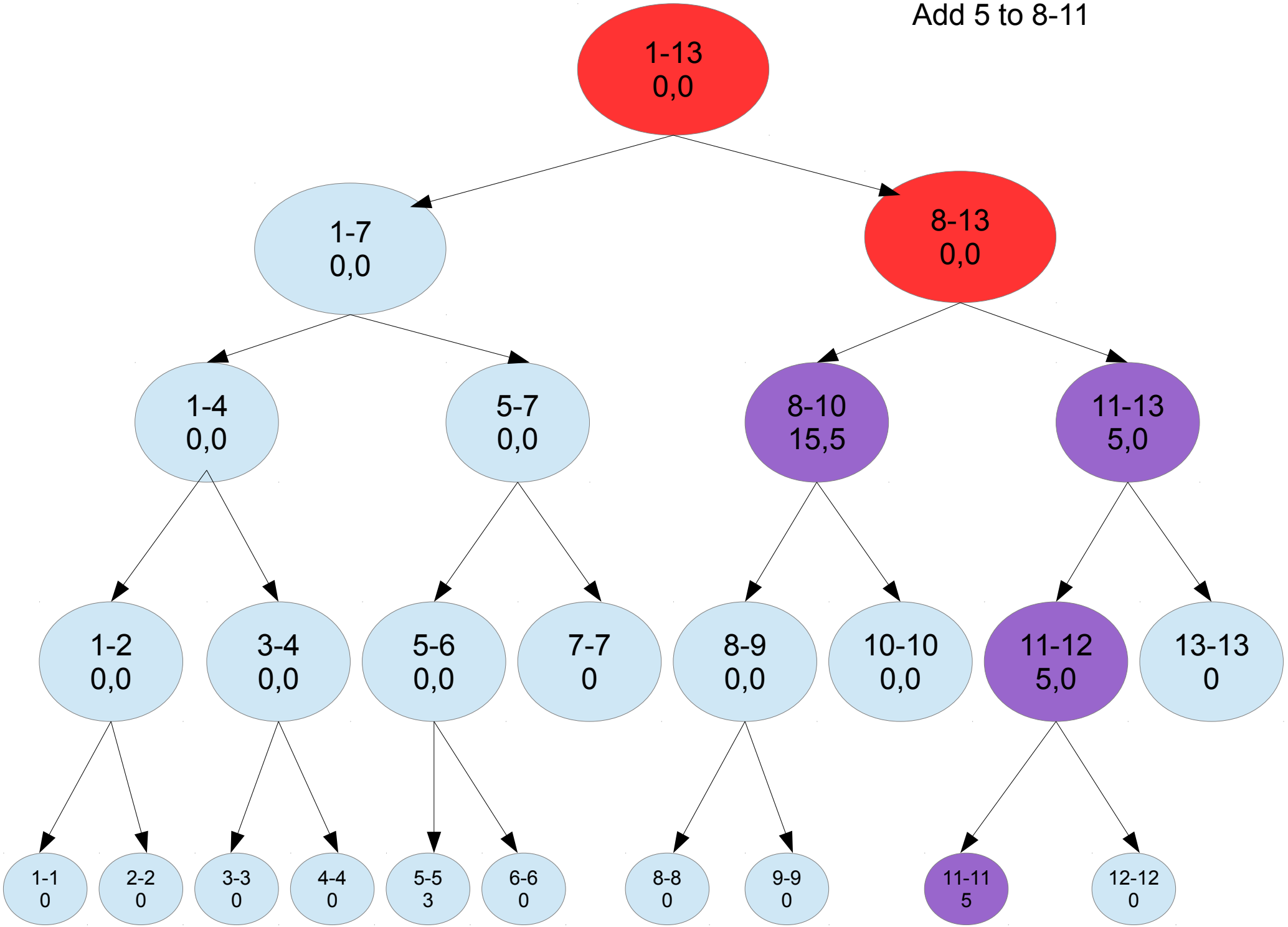

Add 5 to 8-11
1-13
0,0
1-7
0,0
8-13
0,0
1-4
0,0
5-7
0,0
8-10
15,5
11-13
0,0
1-2
0,0
3-4
0,0
5-6
0,0
7-7
0
8-9
0,0
10-10
0,0
11-12
0,0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0

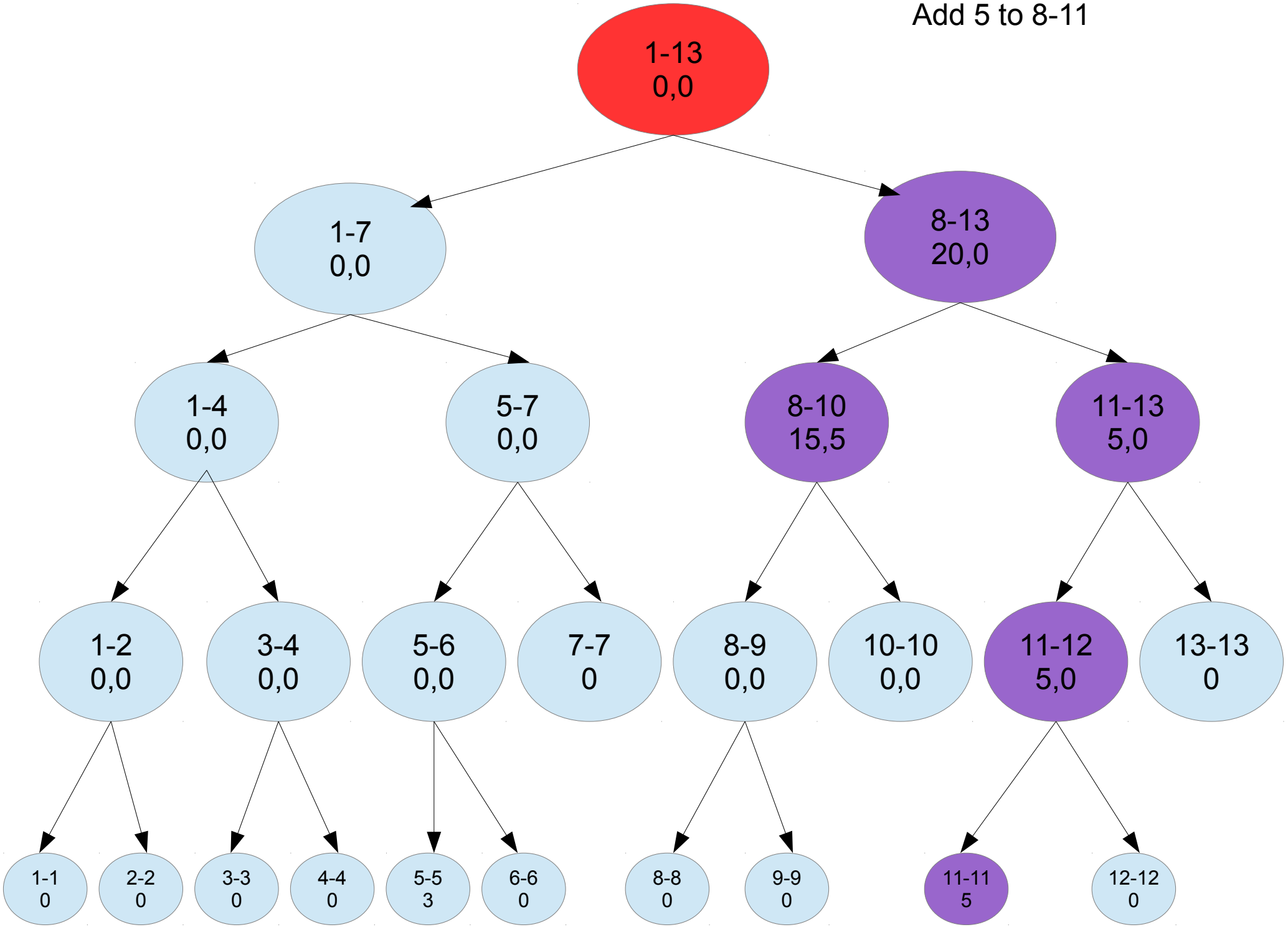

Add 5 to 8-11
1-13
0,0
1-7
0,0
8-13
0,0
1-4
0,0
5-7
0,0
8-10
15,5
11-13
0,0
1-2
0,0
3-4
0,0
5-6
0,0
7-7
0
8-9
0,0
10-10
0,0
11-12
0,0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0

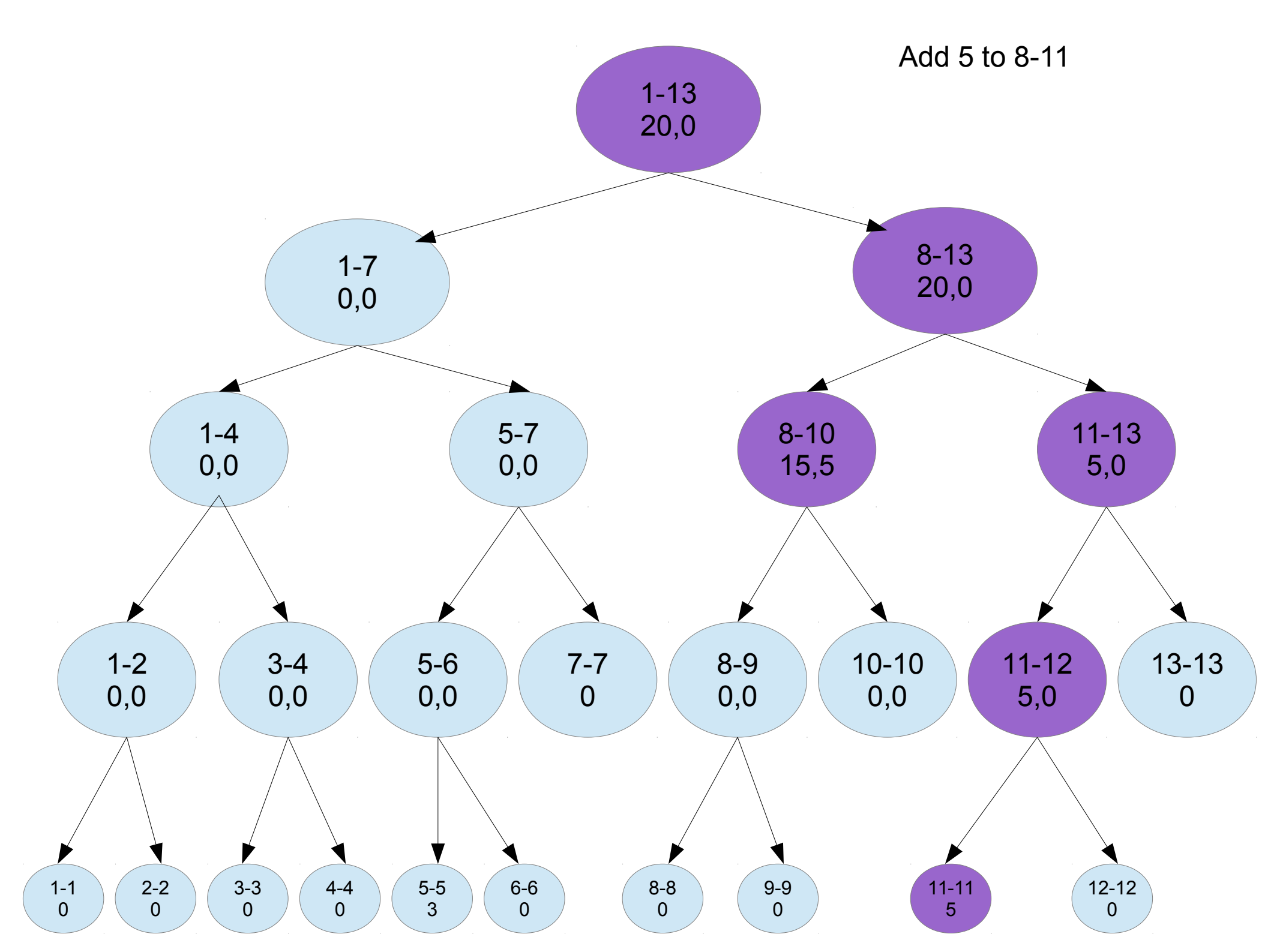

Add 5 to 8-11
1-13
0,0
1-7
0,0
8-13
0,0
1-4
0,0
5-7
0,0
8-10
15,5
11-13
0,0
1-2
0,0
3-4
0,0
5-6
0,0
7-7
0
8-9
0,0
10-10
0,0
11-12
0,0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
0
12-12
0


Add 5 to 8-11
1-13
0,0
1-7
0,0
8-13
0,0
1-4
0,0
5-7
0,0
8-10
15,5
11-13
0,0
1-2
0,0
3-4
0,0
5-6
0,0
7-7
0
8-9
0,0
10-10
0,0
11-12
0,0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
5
12-12
0


Add 5 to 8-11
1-13
0,0
1-7
0,0
8-13
0,0
1-4
0,0
5-7
0,0
8-10
15,5
11-13
0,0
1-2
0,0
3-4
0,0
5-6
0,0
7-7
0
8-9
0,0
10-10
0,0
11-12
5,0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
5
12-12
0


Add 5 to 8-11
1-13
0,0
1-7
0,0
8-13
0,0
1-4
0,0
5-7
0,0
8-10
15,5
11-13
5,0
1-2
0,0
3-4
0,0
5-6
0,0
7-7
0
8-9
0,0
10-10
0,0
11-12
5,0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
5
12-12
0


Add 5 to 8-11
1-13
0,0
1-7
0,0
8-13
20,0
1-4
0,0
5-7
0,0
8-10
15,5
11-13
5,0
1-2
0,0
3-4
0,0
5-6
0,0
7-7
0
8-9
0,0
10-10
0,0
11-12
5,0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
5
12-12
0

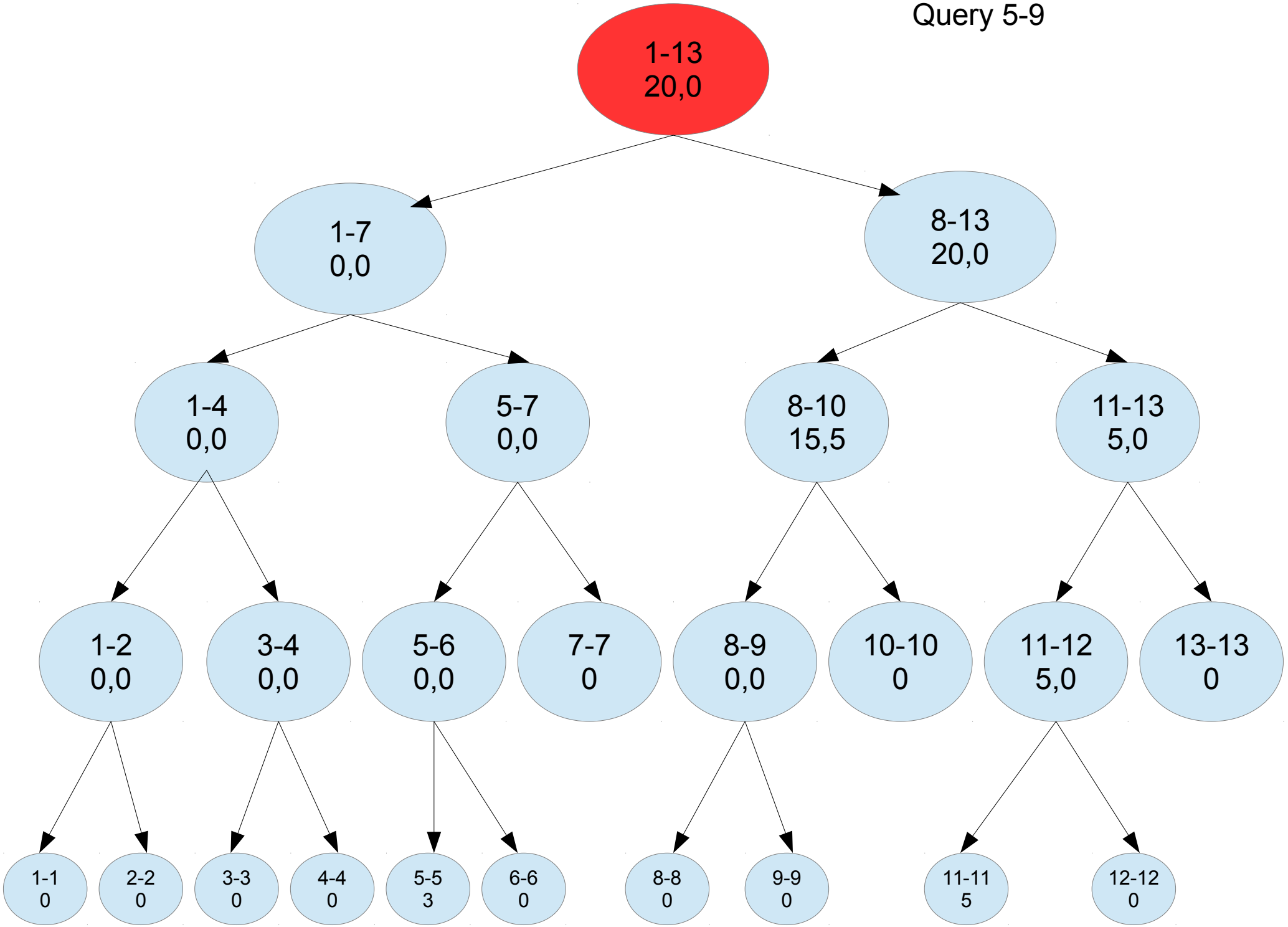

Add 5 to 8-11
1-13
20,0
1-7
0,0
8-13
20,0
1-4
0,0
5-7
0,0
8-10
15,5
11-13
5,0
1-2
0,0
3-4
0,0
5-6
0,0
7-7
0
8-9
0,0
10-10
0,0
11-12
5,0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
5
12-12
0

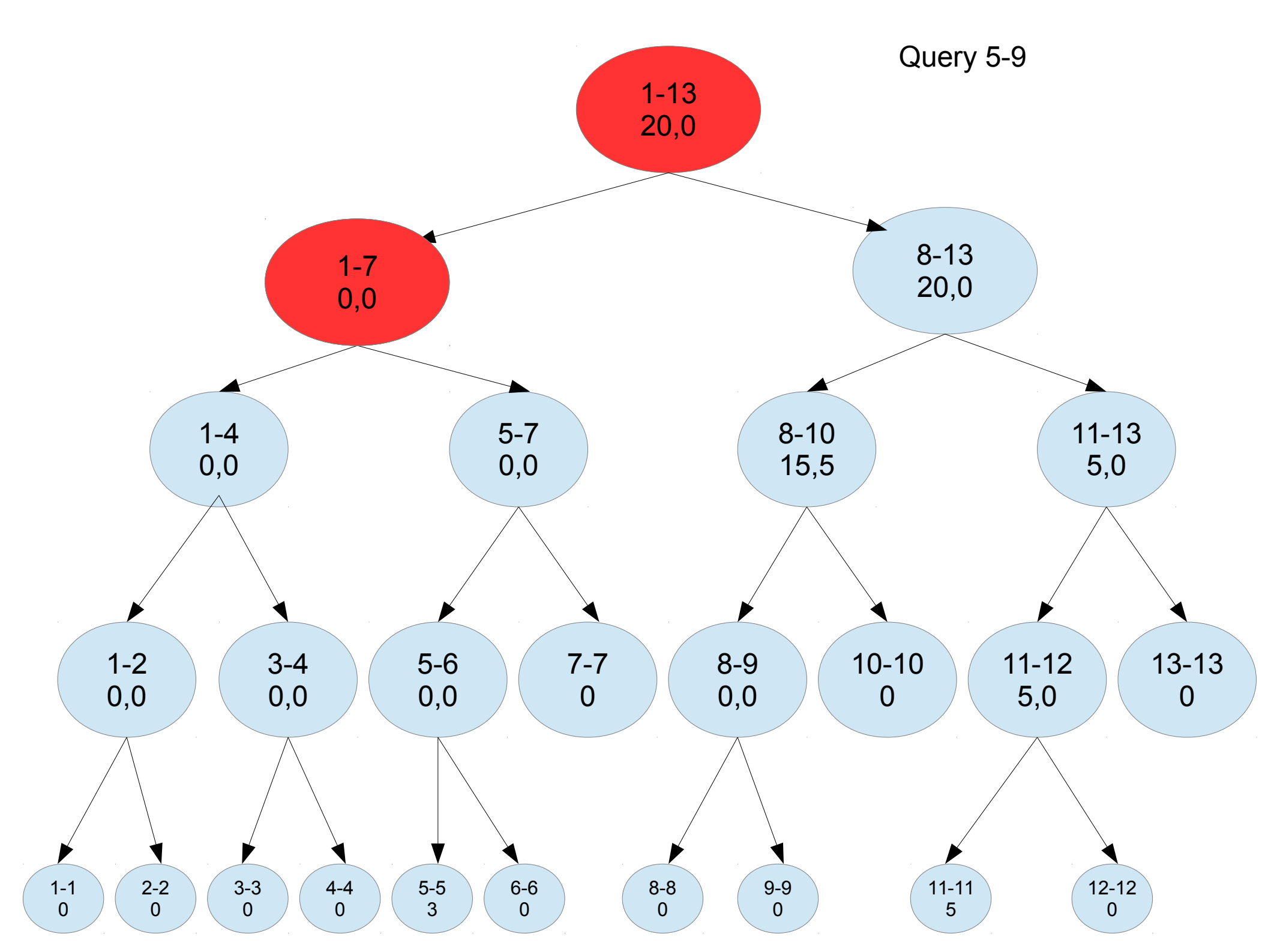
Answering queries is still the same,

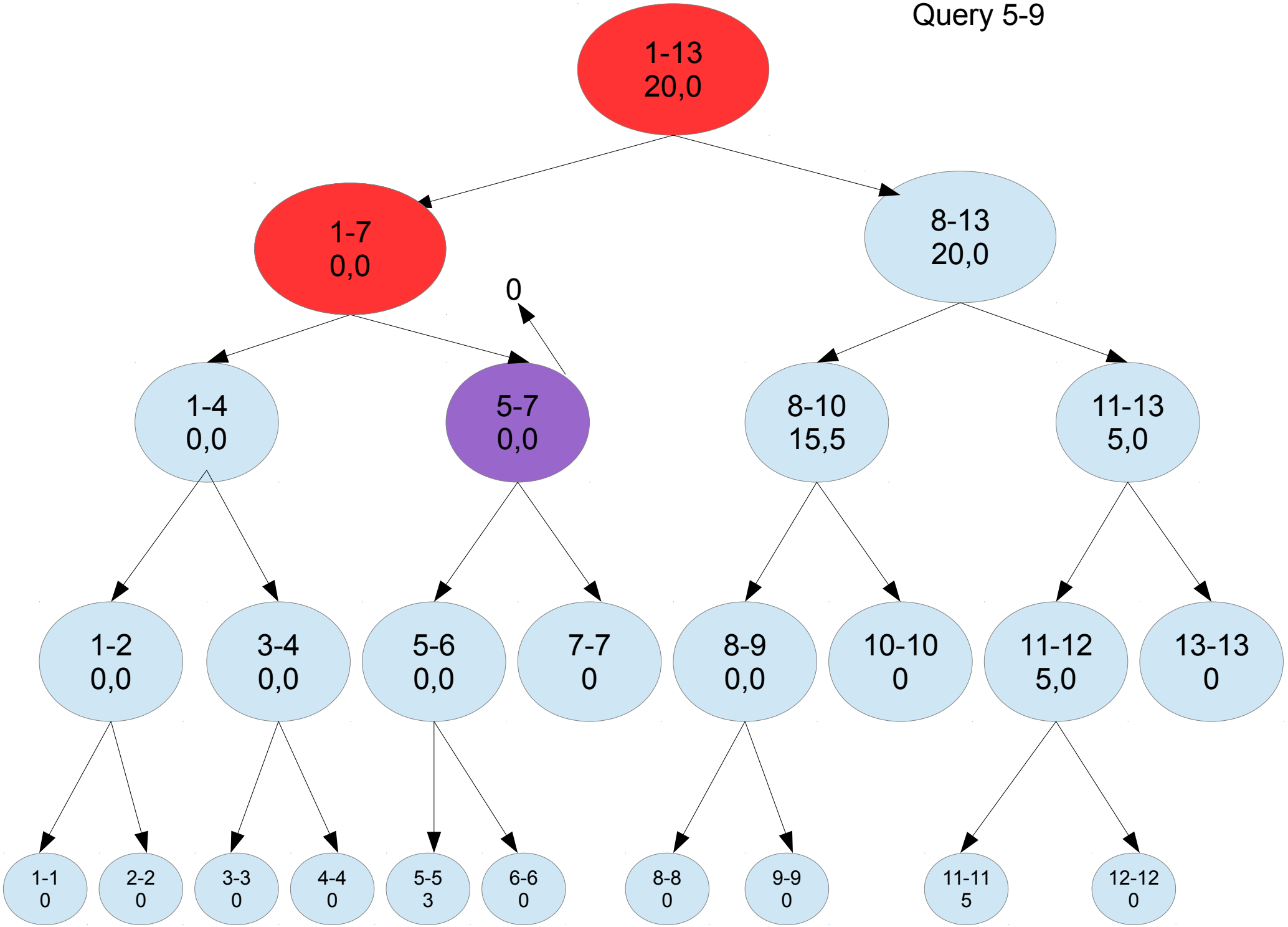
we just propagate the lazy value of the node when it has one.

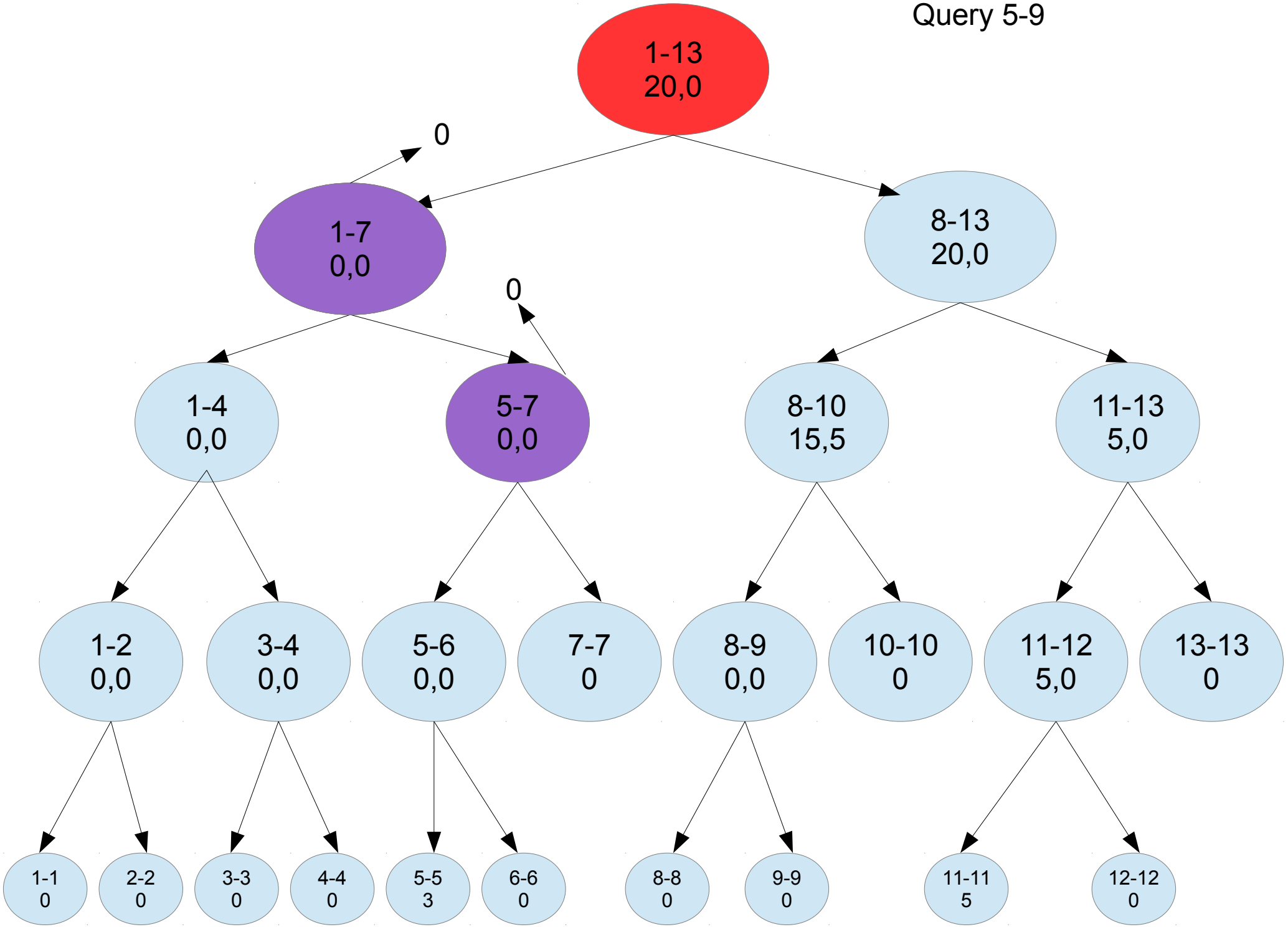
It's easy to see that both in updating an interval and in answering queries, when using lazy propagation, we visit the same nodes as in answering queries without using lazy propagation, so the complexity is still O(logN).

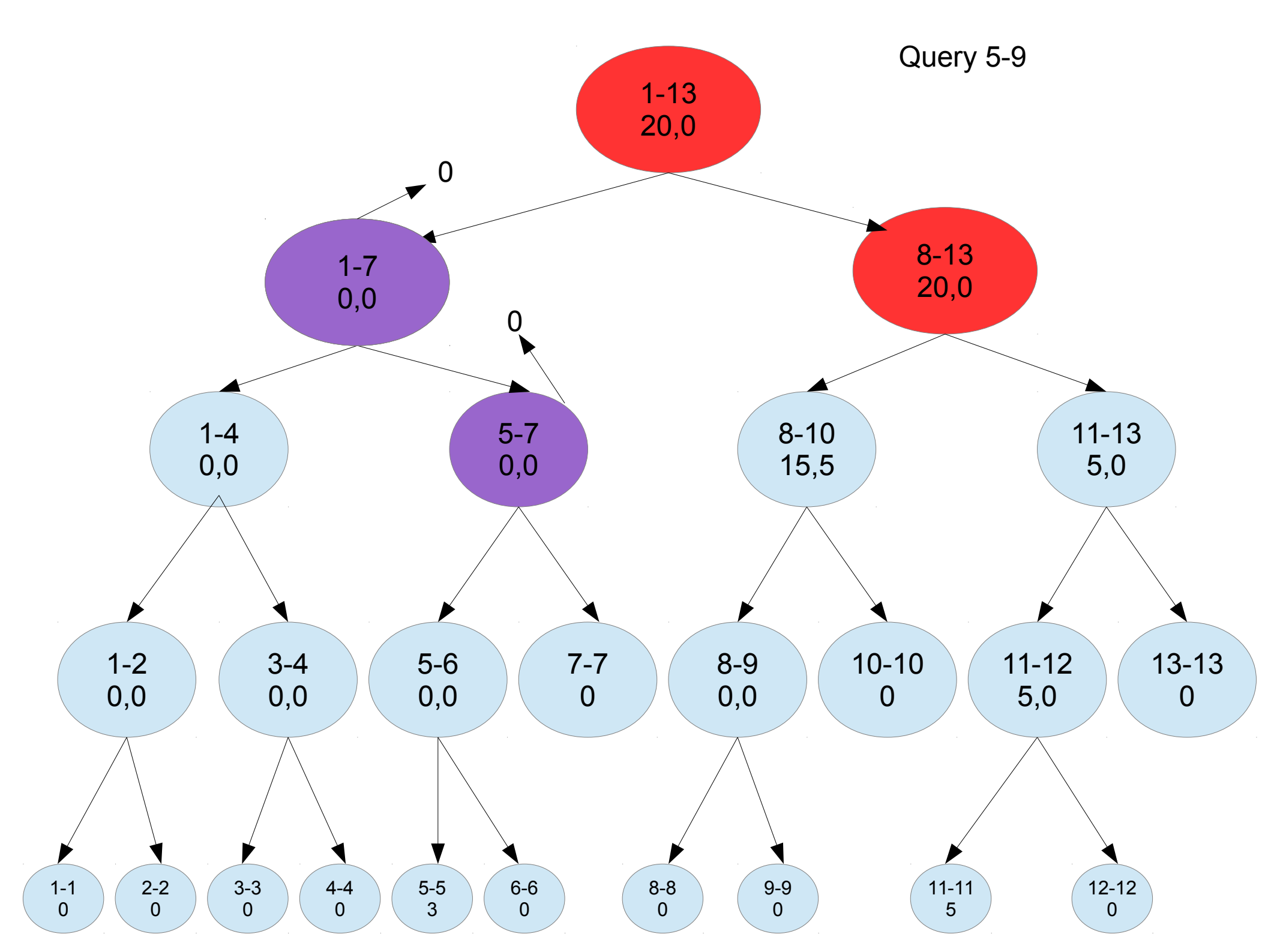
Query 5-9

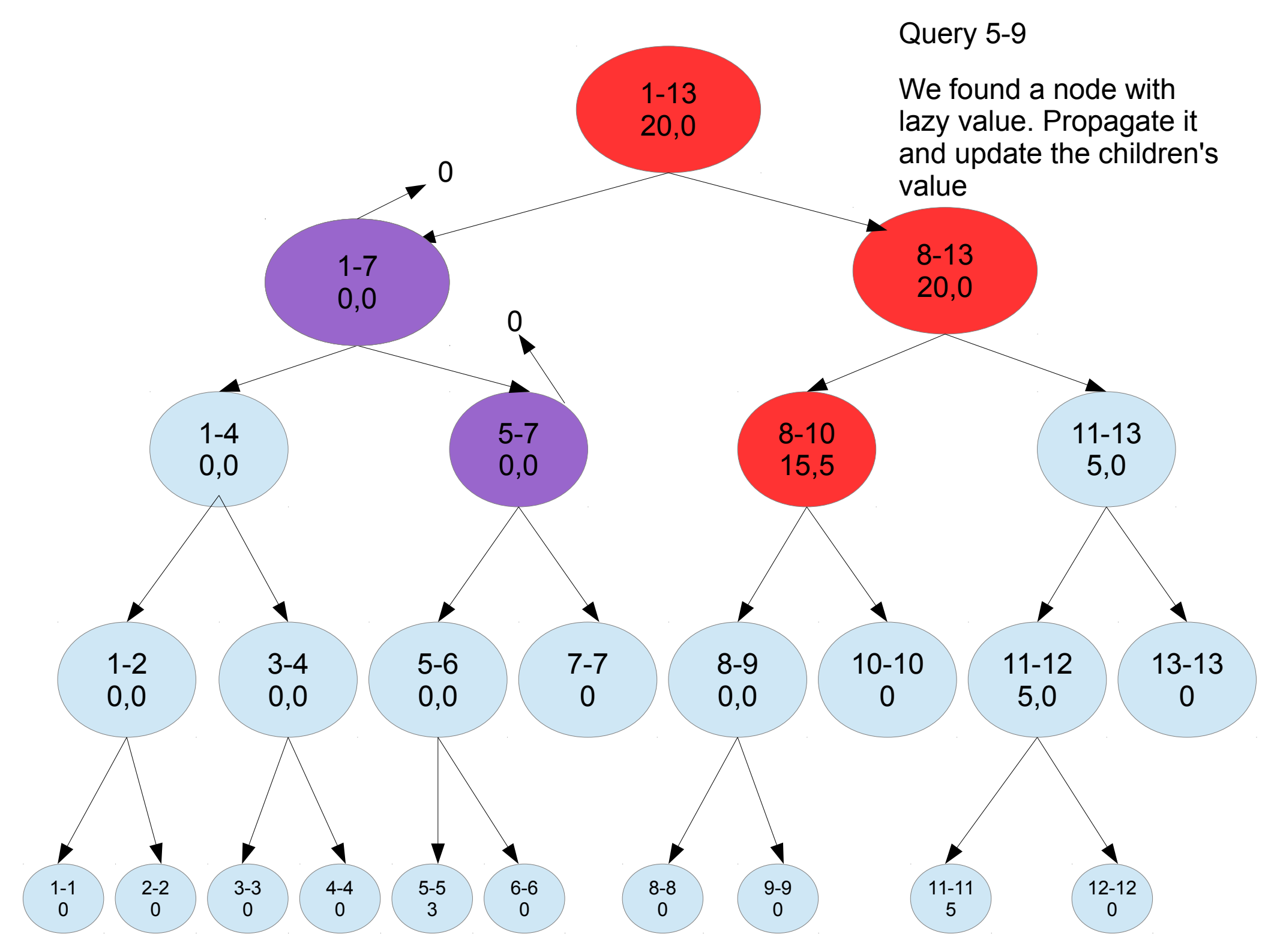

1-13
20,0
1-7
0,0
8-13
20,0
1-4
0,0
5-7
0,0
8-10
15,5
11-13
5,0
1-2
0,0
3-4
0,0
5-6
0,0
7-7
0
8-9
0,0
10-10
0
11-12
5,0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
5
12-12
0

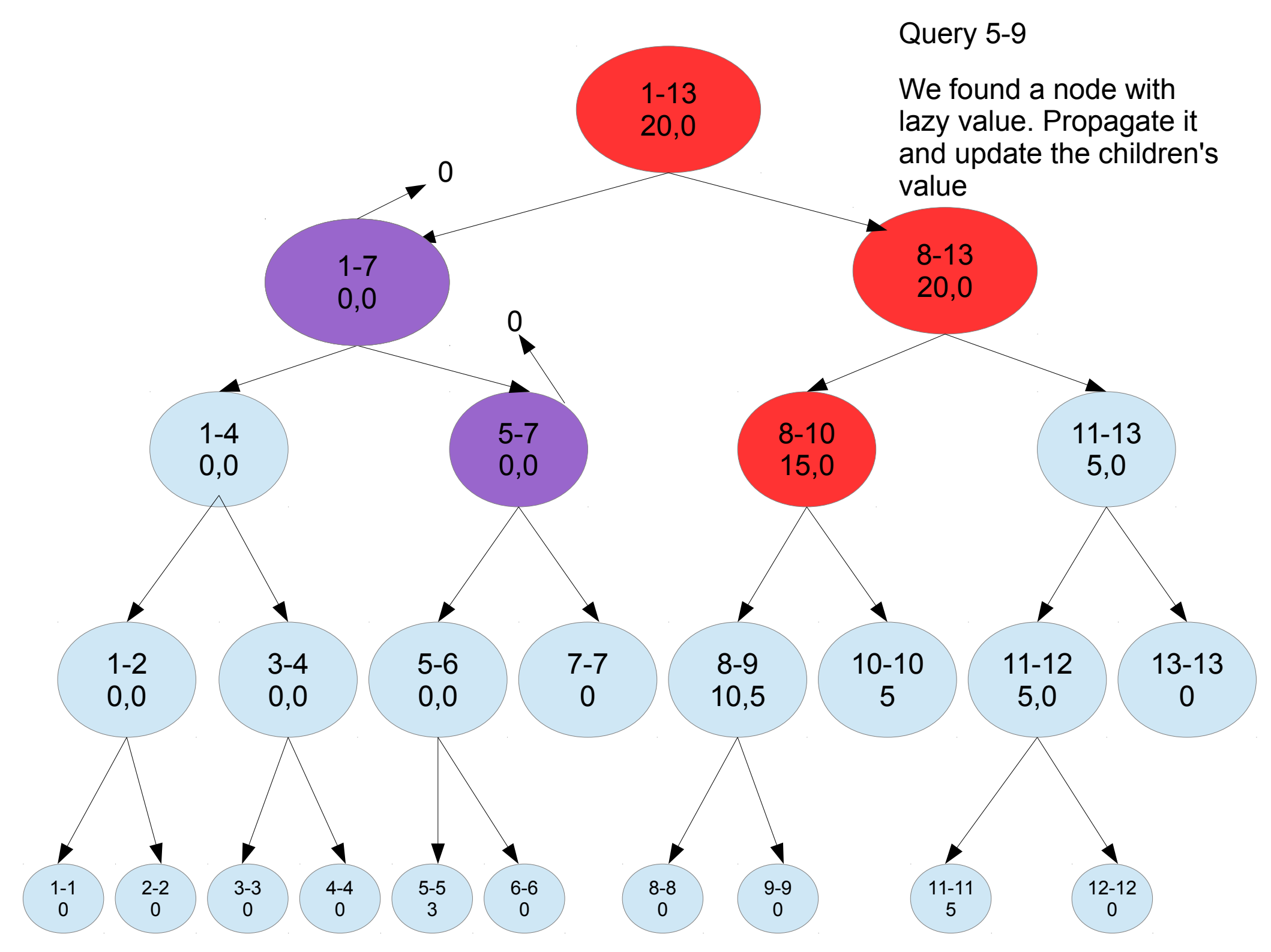

Query 5-9
1-13
20,0
1-7
0,0
8-13
20,0
1-4
0,0
5-7
0,0
8-10
15,5
11-13
5,0
1-2
0,0
3-4
0,0
5-6
0,0
7-7
0
8-9
0,0
10-10
0
11-12
5,0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
5
12-12
0

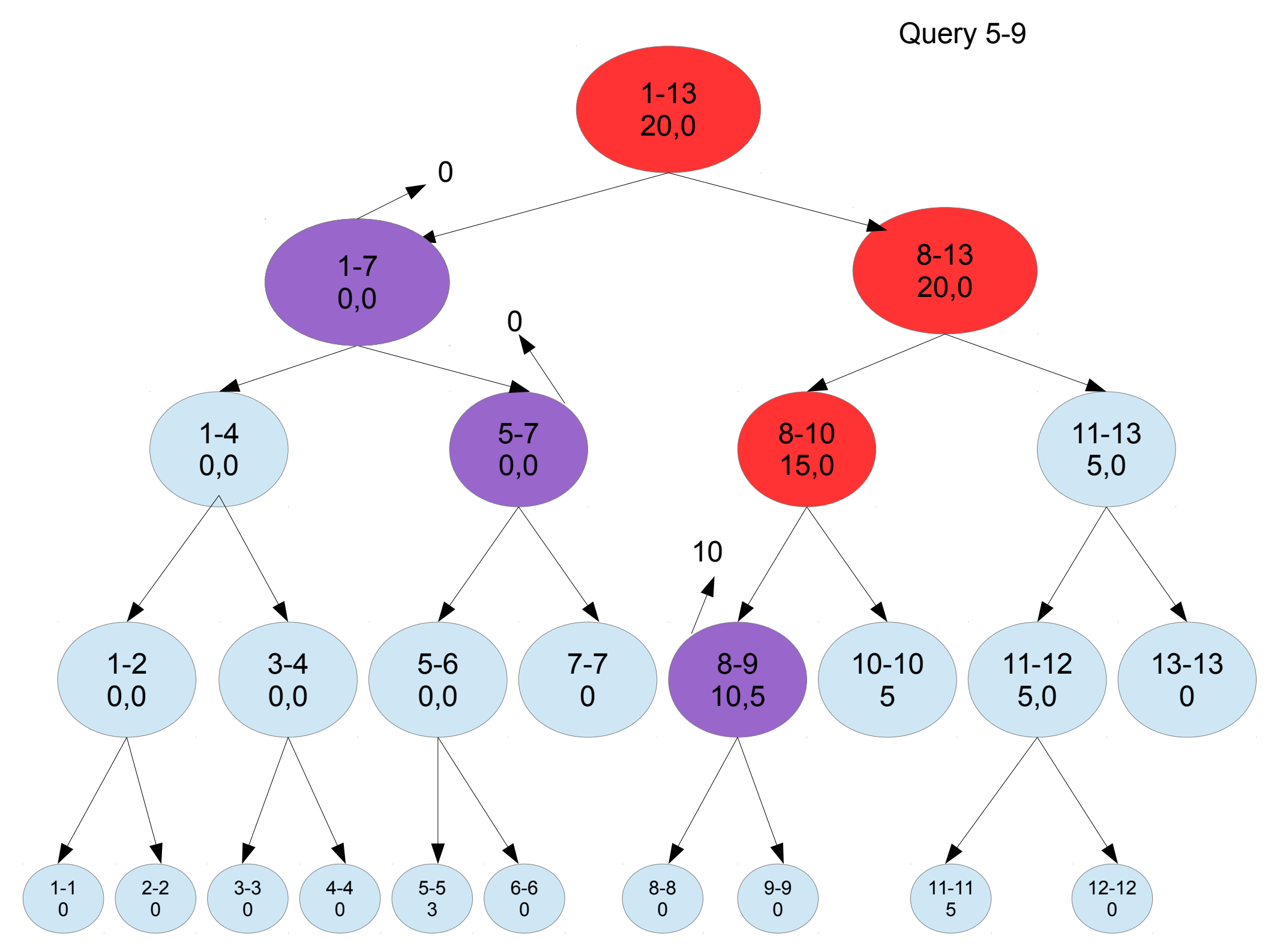

Query 5-9
1-13
20,0
1-7
0,0
8-13
20,0
0
1-4
0,0
5-7
0,0
8-10
15,5
11-13
5,0
1-2
0,0
3-4
0,0
5-6
0,0
7-7
0
8-9
0,0
10-10
0
11-12
5,0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
5
12-12
0

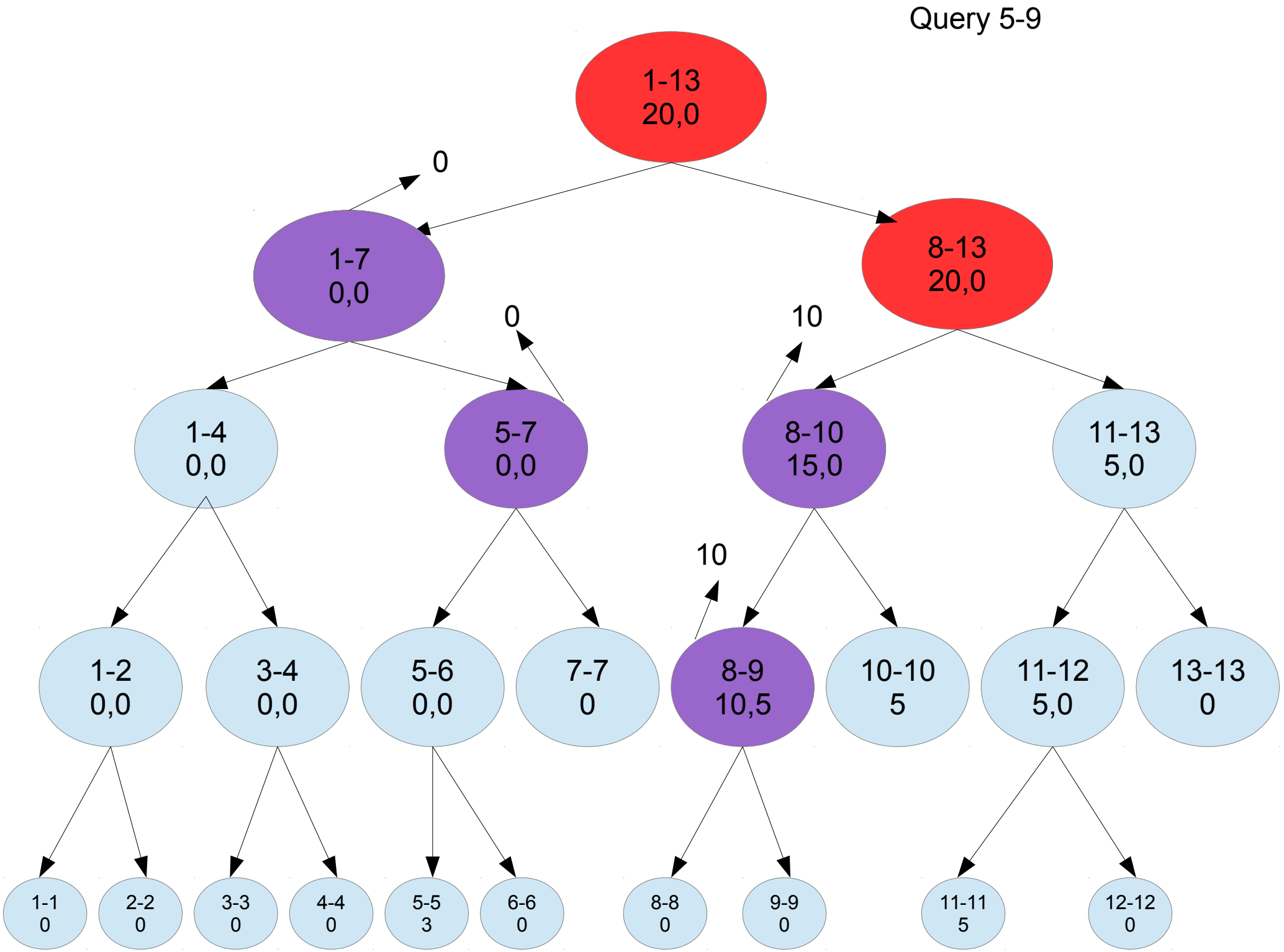

Query 5-9
1-13
20,0
0
1-7
0,0
8-13
20,0
0
1-4
0,0
5-7
0,0
8-10
15,5
11-13
5,0
1-2
0,0
3-4
0,0
5-6
0,0
7-7
0
8-9
0,0
10-10
0
11-12
5,0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
5
12-12
0

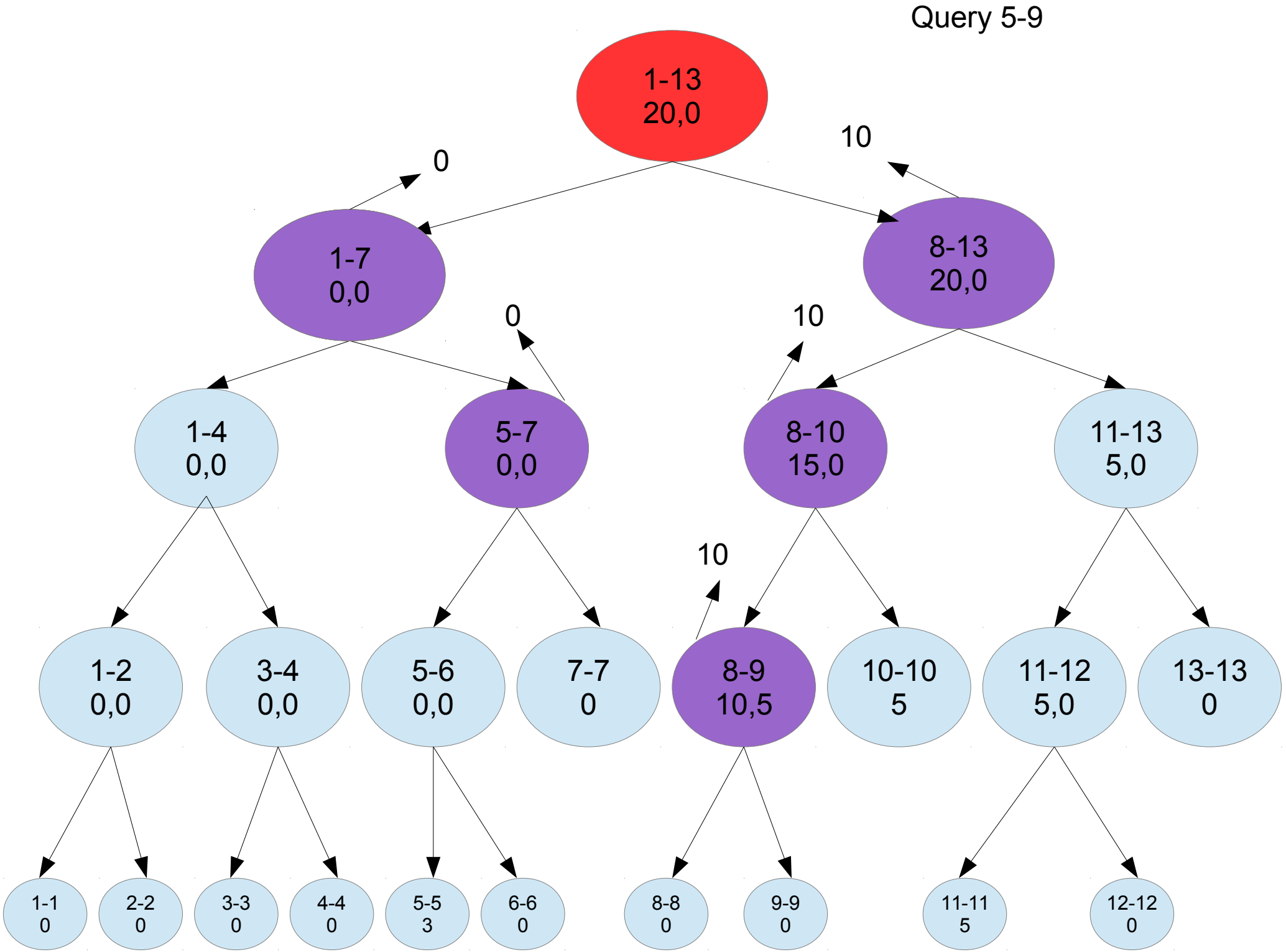

Query 5-9
1-13
20,0
0
1-7
0,0
8-13
20,0
0
1-4
0,0
5-7
0,0
8-10
15,5
11-13
5,0
1-2
0,0
3-4
0,0
5-6
0,0
7-7
0
8-9
0,0
10-10
0
11-12
5,0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
5
12-12
0

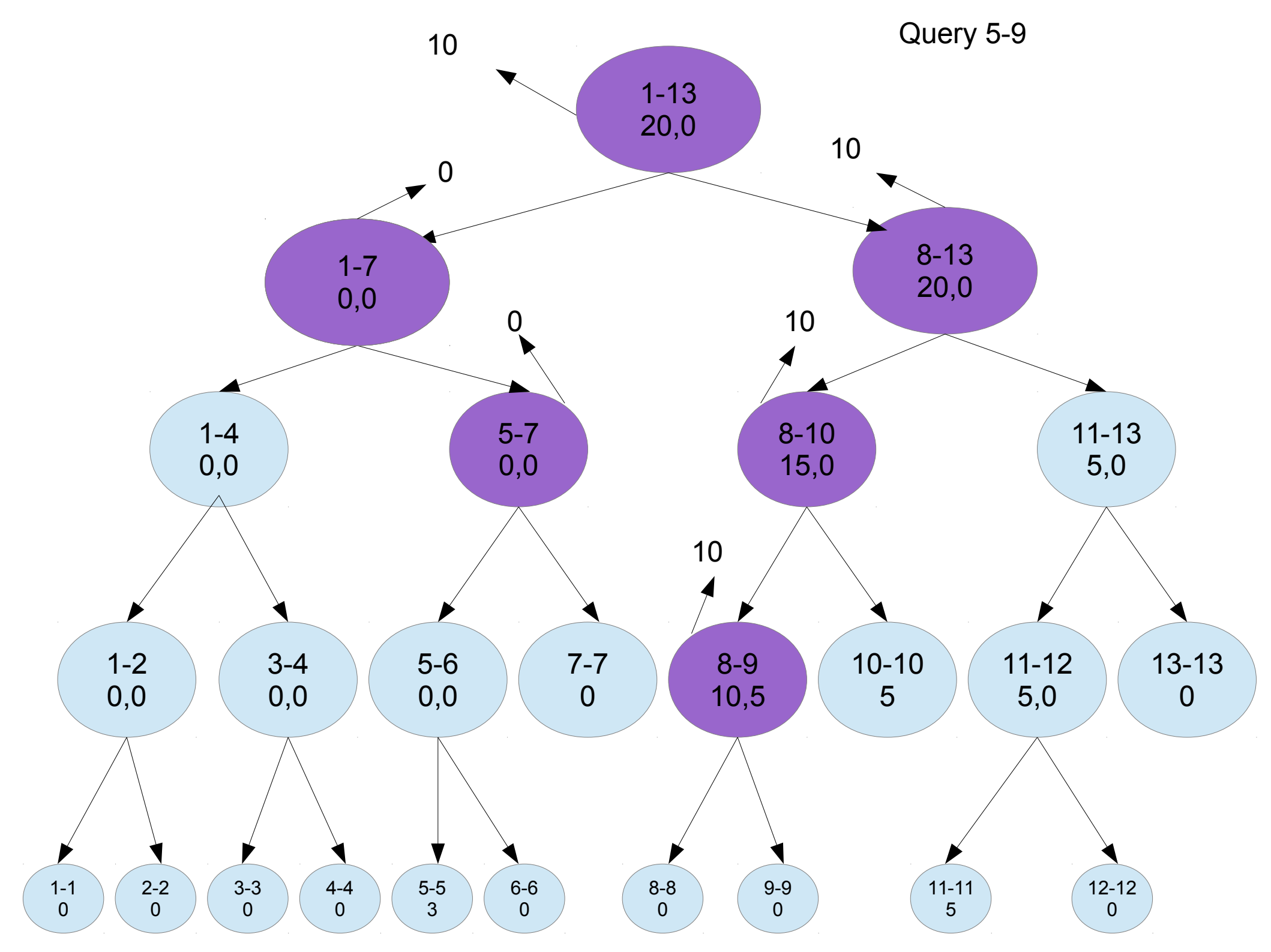

Query 5-9
We found a node with lazy value. Propagate it and update the children's value
1-13
20,0
0
1-7
0,0
8-13
20,0
0
1-4
0,0
5-7
0,0
8-10
15,5
11-13
5,0
1-2
0,0
3-4
0,0
5-6
0,0
7-7
0
8-9
0,0
10-10
0
11-12
5,0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
5
12-12
0


Query 5-9
We found a node with lazy value. Propagate it and update the children's value
1-13
20,0
0
1-7
0,0
8-13
20,0
0
1-4
0,0
5-7
0,0
8-10
15,0
11-13
5,0
1-2
0,0
3-4
0,0
5-6
0,0
7-7
0
8-9
10,5
10-10
5
11-12
5,0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
5
12-12
0


Query 5-9
1-13
20,0
0
1-7
0,0
8-13
20,0
0
1-4
0,0
5-7
0,0
8-10
15,0
11-13
5,0
10
1-2
0,0
3-4
0,0
5-6
0,0
7-7
0
8-9
10,5
10-10
5
11-12
5,0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
5
12-12
0


Query 5-9
1-13
20,0
0
1-7
0,0
8-13
20,0
0
10
1-4
0,0
5-7
0,0
8-10
15,0
11-13
5,0
10
1-2
0,0
3-4
0,0
5-6
0,0
7-7
0
8-9
10,5
10-10
5
11-12
5,0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
5
12-12
0


Query 5-9
1-13
20,0
0
10
1-7
0,0
8-13
20,0
0
10
1-4
0,0
5-7
0,0
8-10
15,0
11-13
5,0
10
1-2
0,0
3-4
0,0
5-6
0,0
7-7
0
8-9
10,5
10-10
5
11-12
5,0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
5
12-12
0


Query 5-9
10
1-13
20,0
0
10
1-7
0,0
8-13
20,0
0
10
1-4
0,0
5-7
0,0
8-10
15,0
11-13
5,0
10
1-2
0,0
3-4
0,0
5-6
0,0
7-7
0
8-9
10,5
10-10
5
11-12
5,0
13-13
0
1-1
0
2-2
0
3-3
0
4-4
0
5-5
3
6-6
0
8-8
0
9-9
0
11-11
5
12-12
0

Next, we will see the code in C++.

v = current node's id
x = starting point of the node
y = ending point of the node
start = starting point of the given interval to update
end = ending point of the given interval to update
val = value to update the nodes [start,end] with

T[v] = value of node v
L[v] = lazy value of node v

We will use the heap represantation for left-right child and father.
To refer to the node's v left child, we wil use 2*v,
To refer to the node's v right child, we will use 2*v+1,
To refer to the node's v father, we will use v/2

It's easy to see that a segment tree has 2*N-1 nodes, so we declare T[] and L[] with 4*MAXN elements when using the heap representation. We need more than 2*MAXN because we don't have a complete tree (see node 7-7 in the previous example, with id = 11), but the space for it's hypothetical children is still reserved (id=22 and id=23).

4*MAXN is guaranteed to be a good upper bound.

```
void update ( int v, int x, int y, int start, int end, int val ) {

    int mid = (x+y)/2;

    if ( x==start && y==end ) {
        L[v] = Lazy ( L[v], val );                    // In our case L[v] += val;
        T[v] = ValueofNode ( T[v], val );             // In our case T[v] += (y-x+1)*val
        return;
    }

//Propagate
    if ( L[v] != 0 ) {
        update ( 2*v, x, mid, x, mid, L[v] );
        update ( 2*v+1, mid+1, y, mid+1, y, L[v] );
        L[v] = 0;
    }

    if ( end <= mid ) {
        update ( 2*v, x, mid, start, end, val );
    }
    else if ( start > mid ) {
        update ( 2*v+1, mid+1, y, start, end, val );
    }
    else {
        update ( 2*v, x, mid, start, mid, val );
        update ( 2*v+1, mid+1, y, mid+1, end, val );
    }

    T[v] = Function ( T[2*v], T[2*v+1] );             // In our case T[v] = T[2*v] + T[2*v+1];
}
```

```
int query ( int v, int x, int y, int start, int end ) {
    int mid = (x+y)/2;

    if ( x==start && y==end ) {
        return T[v];
    }

// Propagate
    if ( L[v] != 0 ) {
        update ( 2*v, x, mid, x, mid, L[v] );
        update ( 2*v+1, mid+1, y, mid+1, y, L[v] );
        L[v] = 0;
    }

    if ( end <= mid ) {
        return query ( 2*v, x, mid, start, end );
    }
    else if ( start > mid ) {
        return query ( 2*v+1, mid+1, y, start, end );
    }
    else {
        return Function ( query (2*v  , x    , mid, start, mid) )
            query (2*v+1, mid+1, y  , mid+1, end) );     // return query(2*v,...) + query(2*v+1,...)
    }
}
```

We can use another representation for getting left and right child.
It requires more code, but it can solve more problems.

We keep two more arrays left[v] and right[v] which gives the id of the left and right child of node v. It begins filled with 0.
When we need to visit the left child, instead of 2*v, we call the function Left (v), which returns left[v] if it is non-zero, or gives it a new value (the first one that hasn't been used already).

The good thing is that you don't need to create every node.
Since each operation visits O(logN) nodes (even if we use lazy propagation), at most O (logN) nodes will be created.

So K * logN * c, where K = The number of operations and c = a small constant (about 4 to make sure), nodes will be created (actually it's much less but that's a pretty easy to understand upper_bound).

So if K = 100.000 operations and N = 1.000.000.000, we can't use the heap representation, since it would require 4*N = 4.000.000.000 nodes.
But using this representation, we only need 9.600.000, which is 400 times smaller.

A very useful application of this, is building an online order-statistic-tree for contests where you don't have much time to build a red-black-tree.
When you want to insert value X, you just add 1 to position X.
To find X's order, you just need the sum (1...X-1).
To find the node with order K, you search for it at the left child if it's sum is >= K, or you search at the right child for node with order K-left_value.

Recommended Problems :

http://www.spoj.pl/problems/HAYBALE/
http://www.spoj.pl/problems/LITE/
http://www.spoj.pl/problems/HORRIBLE/
http://www.spoj.pl/problems/HELPR2D2/
http://www.spoj.pl/problems/NKMOU/
http://www.spoj.pl/problems/RPAR/
http://www.spoj.pl/problems/SEGSQRSS/
https://www.spoj.pl/problems/CASHIER/
http://www.spoj.pl/problems/TEMPLEQ/

Other problems which involve segment trees,
Plus some precomputation (usually that's sorting the queries and answering them in this order).

http://acm.timus.ru/problem.aspx?space=1&num=1028
http://acm.timus.ru/problem.aspx?space=1&num=1613
http://www.spoj.pl/problems/DQUERY/
http://www.spoj.pl/problems/KQUERY/
http://www.spoj.pl/problems/CCOST/
http://codeforces.com/problemset/problem/220/B

http://boi2012.dms.rs/index.php?action=show&data=tasks
Task : The best teams

http://www.spoj.pl/problems/GSS2/
http://www.codeforces.com/problemset/problem/91/E