5800
data structures

apr8/apr11 2022

shelat

Dictionary

data structure

Dictionary

insert(key, value)

delete(key)

lookup(key)

findnext(key)

Dictionary

standard solution: hashtable

insert(key, value)

delete(key)

lookup(key)

findnext(key)

Hashtables are tricky

```python

import time
import sys
import d

dd = {}

# make a dictionary with elements from the list
for l in d.list:
    dd[l] = l

def lookup(v):
    start = time.time()
    t = 0
    for j in range(10000):
        if v in dd:
            t = t +1
    end = time.time()
    print(end - start)
    return t

```

Hashtables are tricky

```python

import time
import sys
import d

dd = {}

# make a dictionary with elements from the list
for l in d.list:
    dd[l] = l

def lookup(v):
    start = time.time()
    t = 0
    for j in range(10000):
        if v in dd:
            t = t +1
    end = time.time()
    print(end - start)
    return t

```

This is a trivial lookup experiment. Looking up 1 key takes 2000x longer.

```
MacBook-Pro-2:hashing abhi$ python3 bad.py
size of dictionary: 43689
Starting experiment to lookup 1000:
0.0005161762237548828
Starting experiment to lookup 100000:
1.0303189754486084
MacBook-Pro-2:hashing abhi$
```

Hashtables are tricky

```python

import time
import sys
import d

dd = {}

# make a dictionary with elements from the list
for l in d.list:
    dd[l] = l

def lookup(v):
    start = time.time()
    t = 0
    for j in range(10000):
        if v in dd:
            t = t +1
    end = time.time()
    print(end - start)
    return t

```

This is a trivial lookup experiment.
Looking up 1 key takes 2000x longer.

```
MacBook-Pro-2:hashing abhi$ python3 bad.py
size of dictionary: 43689
Starting experiment to lookup 1000:
0.0005161762237548828
Starting experiment to lookup 100000:
1.0303189754486084
MacBook-Pro-2:hashing abhi$
```

Worst case performance: $O(n)$

Dictionary

new constraint: keys belong to limited range:

$$\{1, \ldots, n\}$$

insert(key, value)

delete(key)

lookup(key)

findnext(key)

A simple solution: bit vector

Maintain an array of bits

insert(key, value)

delete(key)

lookup(key)

findnext(key)

Can we do better than O(n) findnext?

van emde Boas Q

THE BIG IDEA:

van emde Boas Q

THE BIG IDEA:

Use recursion for a data structure.

A data structure that handles 1..n can be designing using several smaller versions of the same structure.

VEB queue

VEB$_{(N)}$

VEB queue

VEB(N)

SZ, MIN, MAX

VEB queue

VEB(N)

SZ, MIN, MAX

BASE CASE: 1 BIT VECTOR.

VEB queue

VEB(N)

SZ, MIN, MAX

BASE CASE: 1 BIT VECTOR.

NORMAL CASE:

MAP

$\mathrm{VEB}(\sqrt{n})$

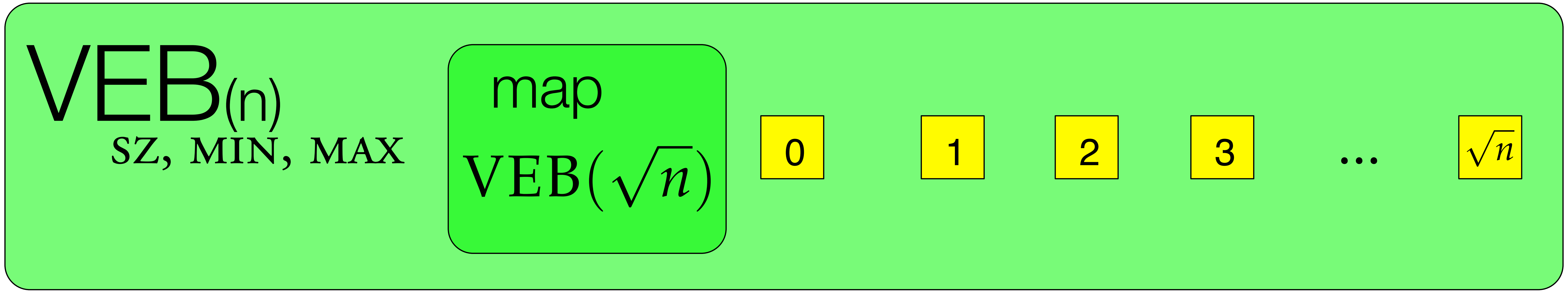
Pointers to recursive, smaller instances of VEB.

0 1 2 3 … $\sqrt{n}$

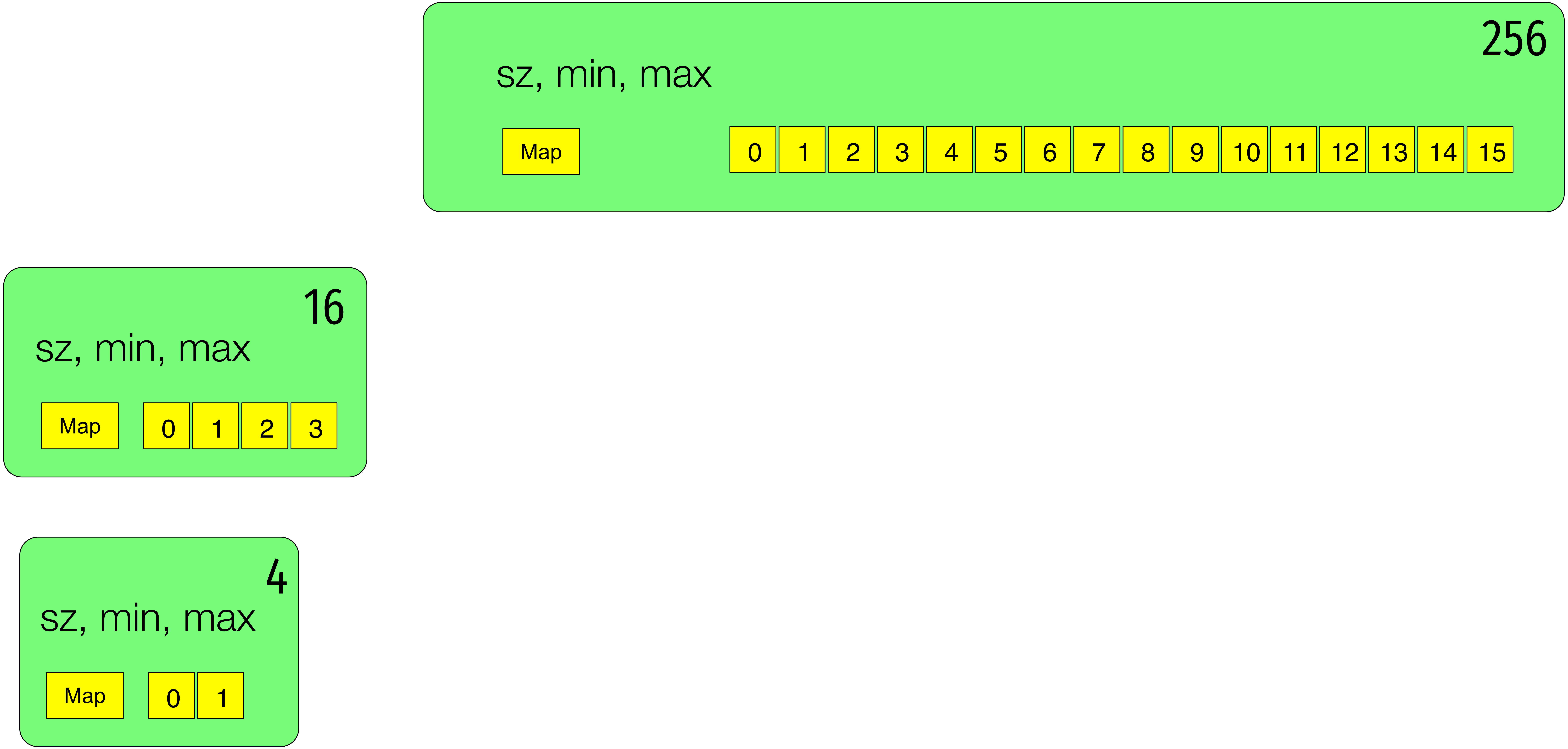
Keeps track of which ptrs are not empty.

EXAMPLE:

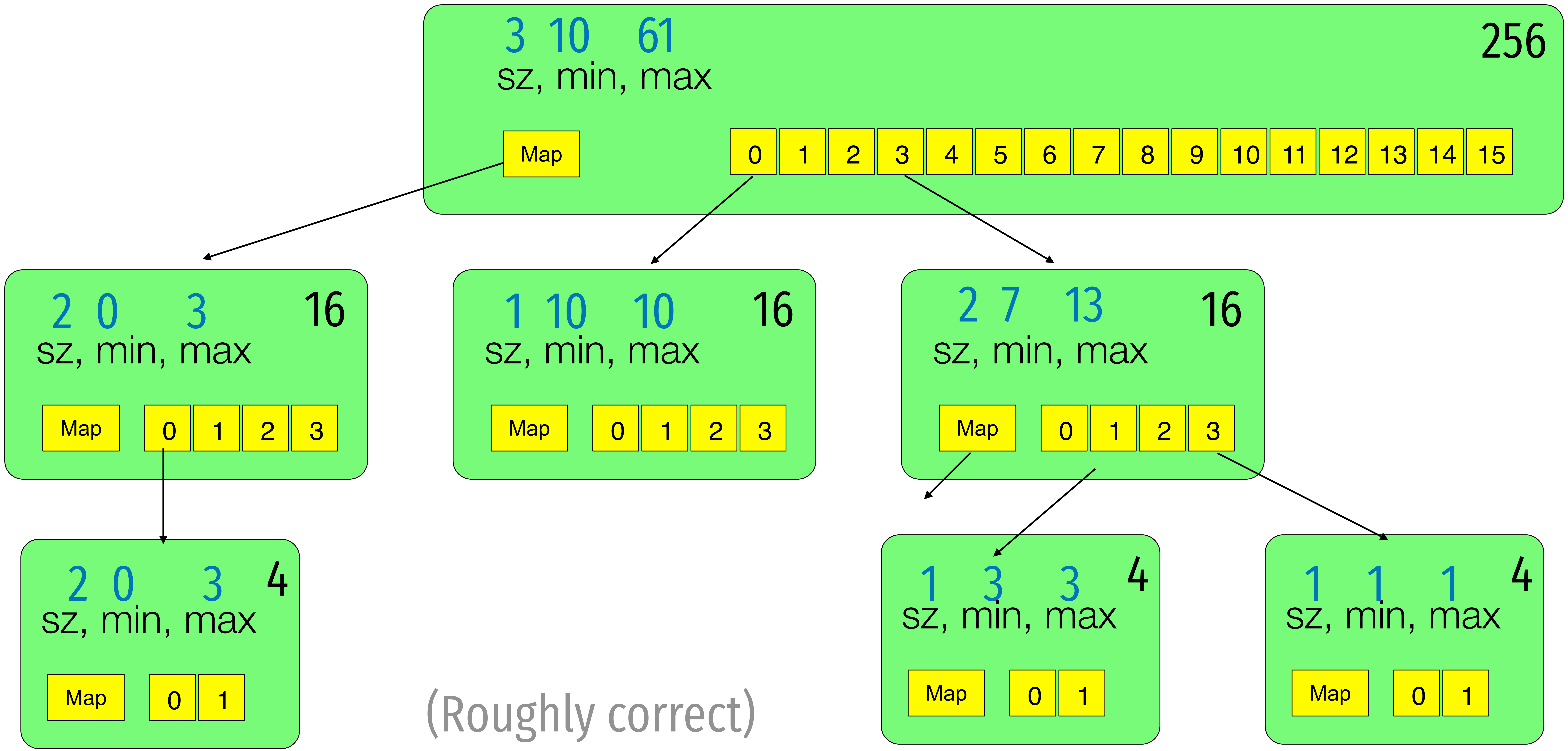
$n = 256$



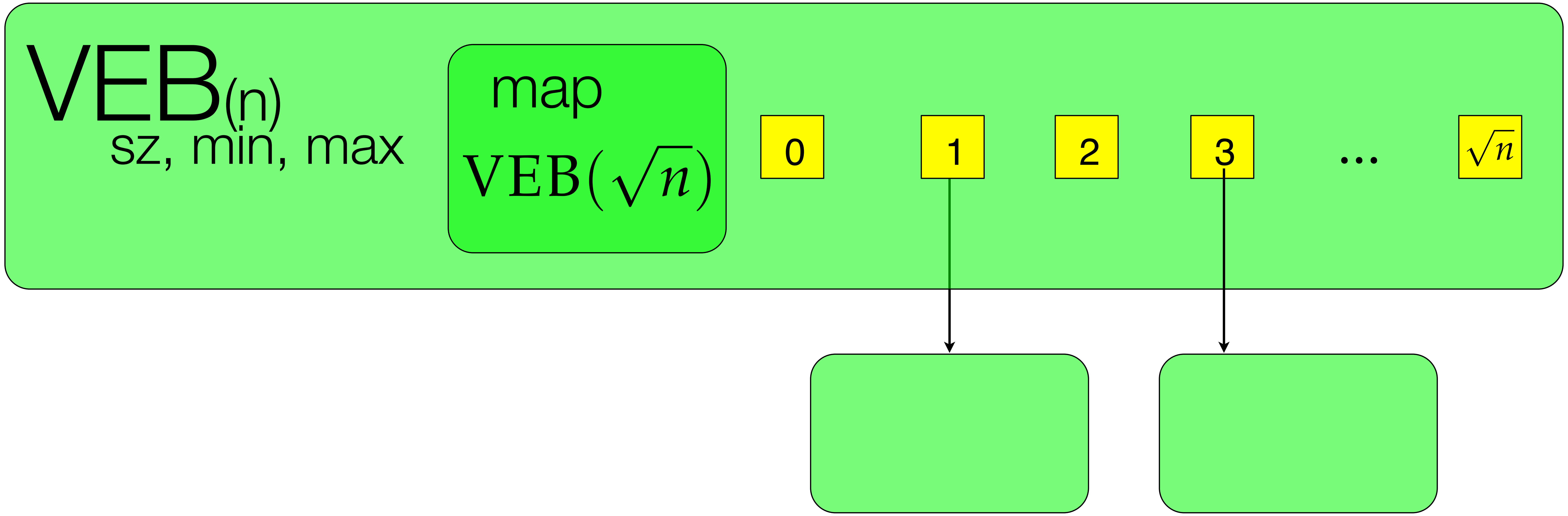
Example n=256, keys={10,55,61}



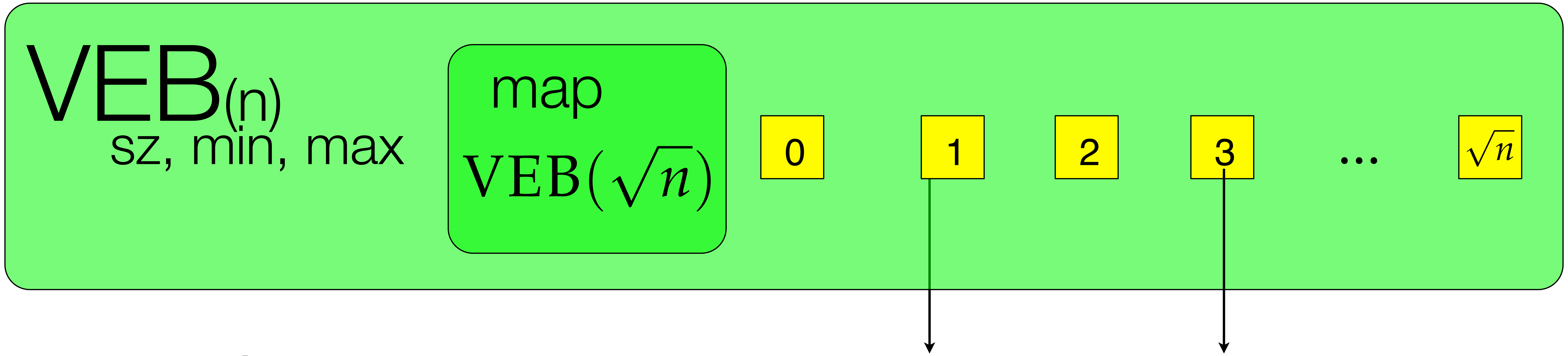
Example n=256, keys={10,55,61}



(Roughly correct)



LOOKUP(i)



LOOKUP(i)

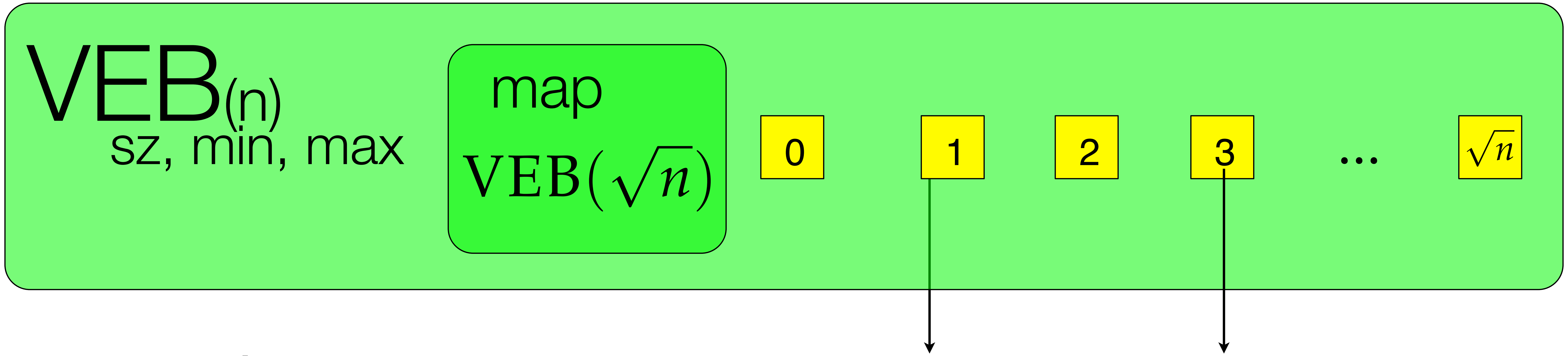
WRITE $i = a\sqrt{n} + b$

IF <BASE CASE>: CHECK BIT VECTOR

IF SIZE = 0 OR [a].SIZE = 0 THEN RETURN FALSE

ELSE RETURN [a].LOOKUP(b)

(Almost right, we will have to slightly change this later.)



LOOKUP(i)

WRITE $i = a\sqrt{n} + b$

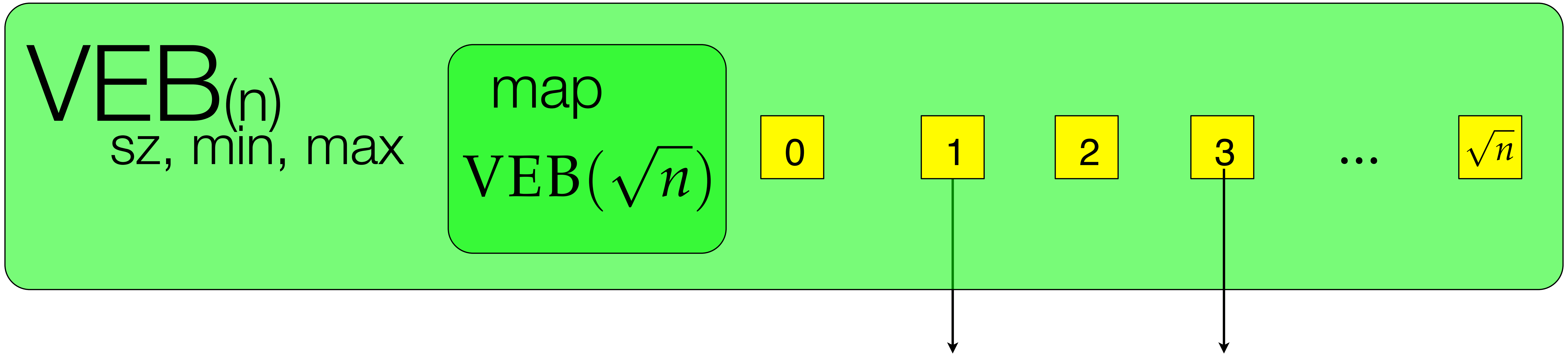
IF <BASE CASE>: CHECK BIT VECTOR

IF SIZE = 0 OR a .SIZE = 0 THEN RETURN FALSE

ELSE RETURN a .LOOKUP(b)

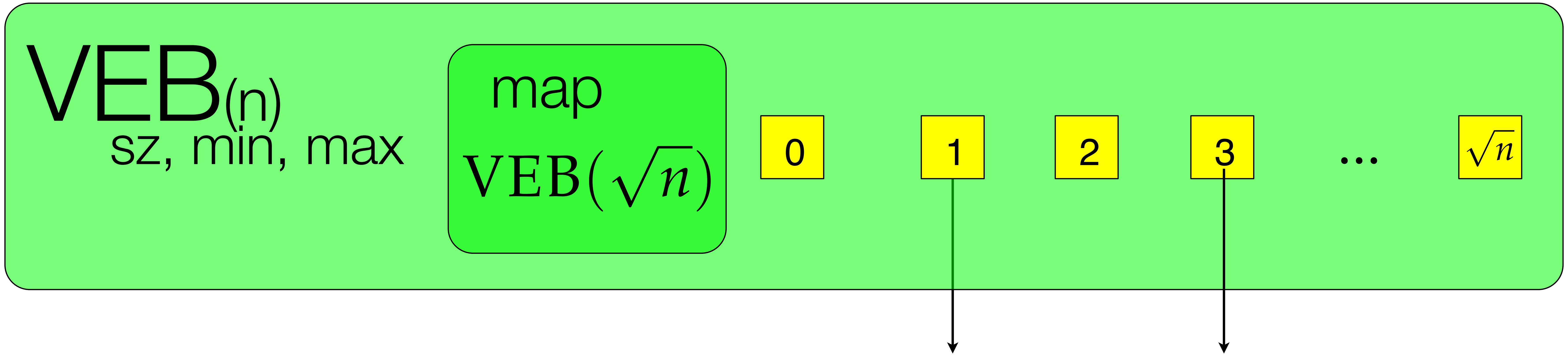
Running time: $T(n) = T(\sqrt{n}) + \Theta(1) = \Theta(\log \log n)$

(Almost right, we will have to slightly change this later.)



FINDNEXT(i)

IDEA:



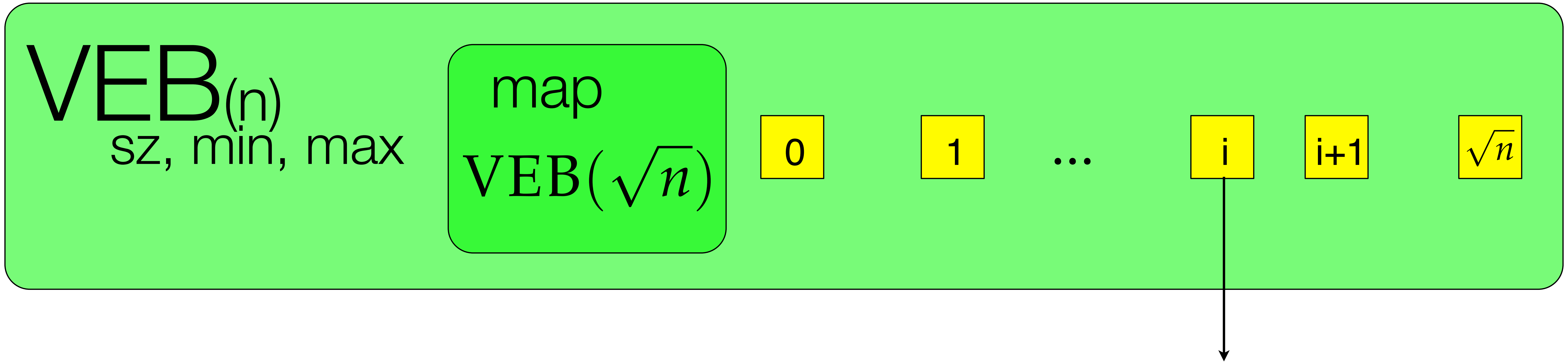
FINDNEXT(i)

IDEA:

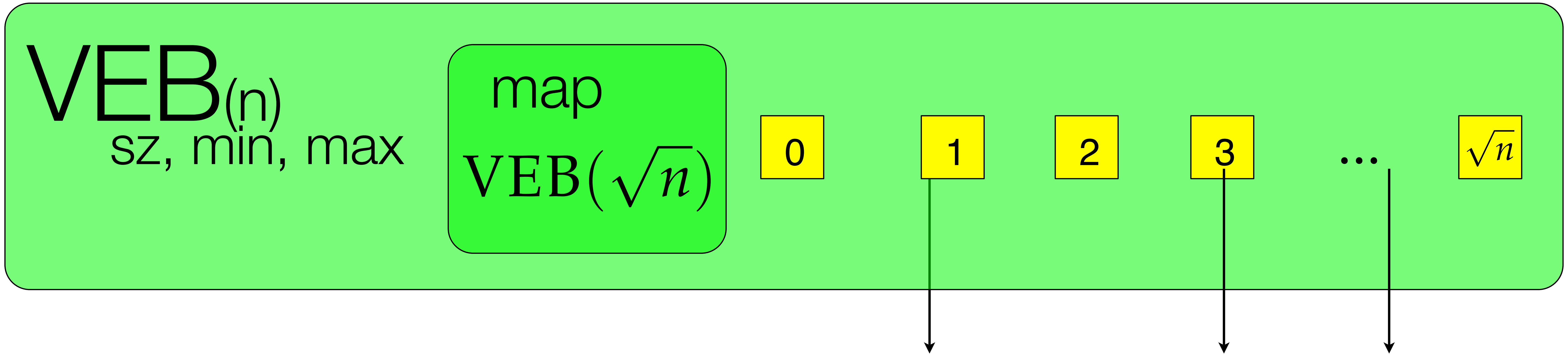
Write $i = a\sqrt{n} + b$ as usual.

Case 1: Bucket a has the next value.
Recursively use $\text{findnext}_a(b)$

Case 2: Bucket a does not have the next value.
Use $x = \text{findnext}_{map}(a)$, return x.min.



FINDNEXT(i)



FINDNEXT(i)

WRITE $i = a\sqrt{n} + b$

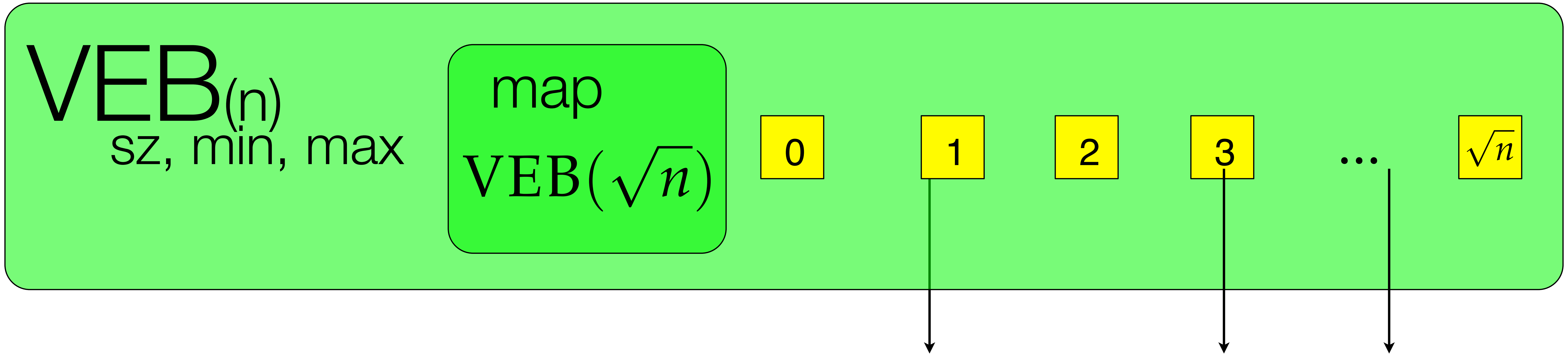
<BASE CASE IF SIZE IS ZERO>

IF a .MAX $> b$ THEN

RETURN a .FINDNEXT(b)

ELSE

RETURN MAP .FINDNEXT(a) .MIN



FINDNEXT(i)

WRITE $i = a\sqrt{n} + b$

<BASE CASE IF SIZE IS ZERO>

IF a .MAX $> b$ THEN

RETURN a .FINDNEXT(b)

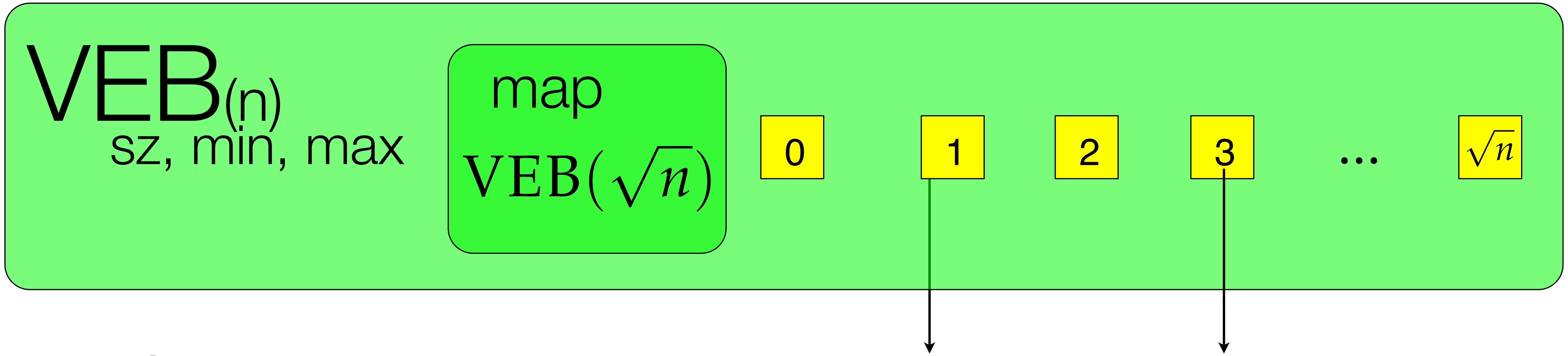
ELSE

RETURN MAP .FINDNEXT(a) .MIN

Running time:

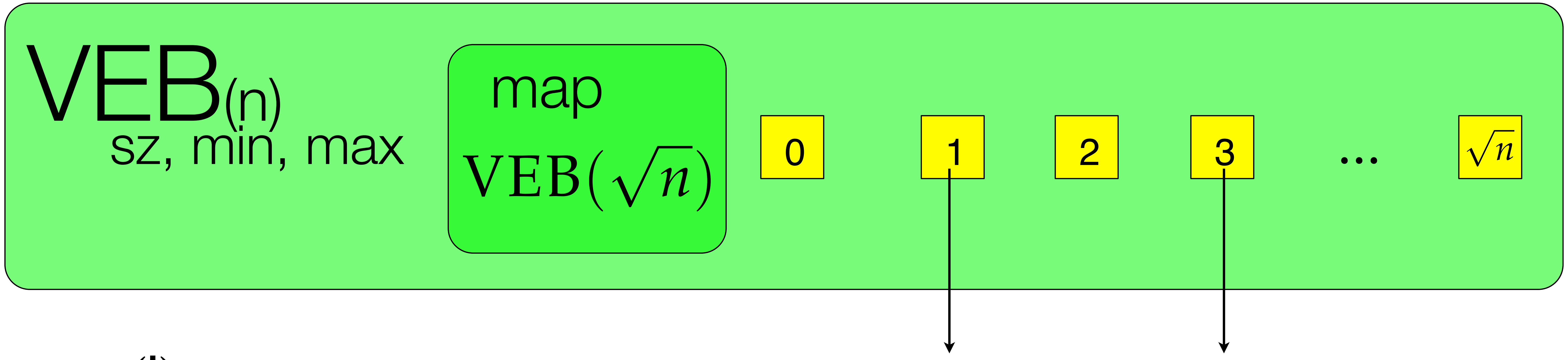
$$T(n) = T(\sqrt{n}) + \Theta(1)$$

$$\Theta(\log \log n)$$



INSERT(i)

WRITE $i = a\sqrt{n} + b$

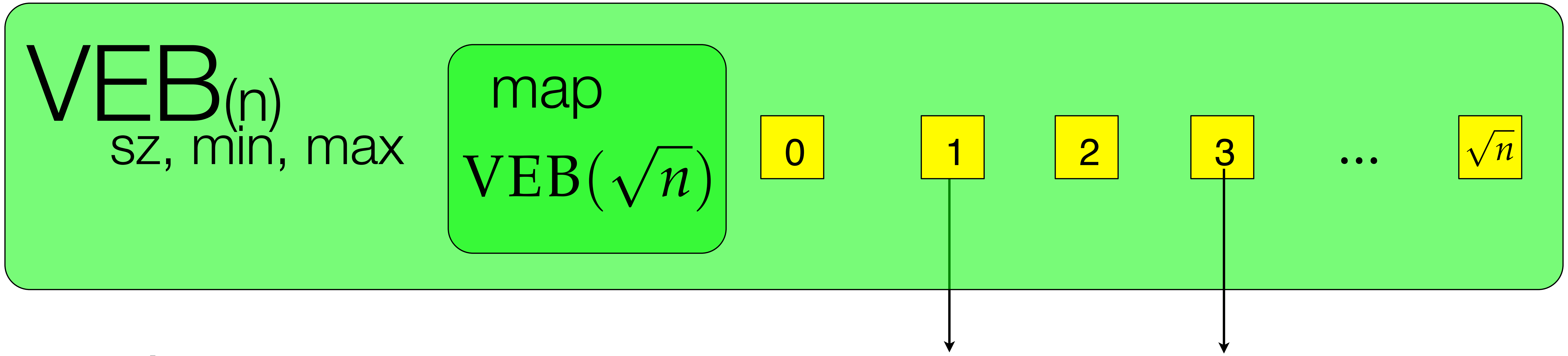


INSERT(i)

WRITE $i = a\sqrt{n} + b$

A.INSERT(B)

MAP.INSERT(A)



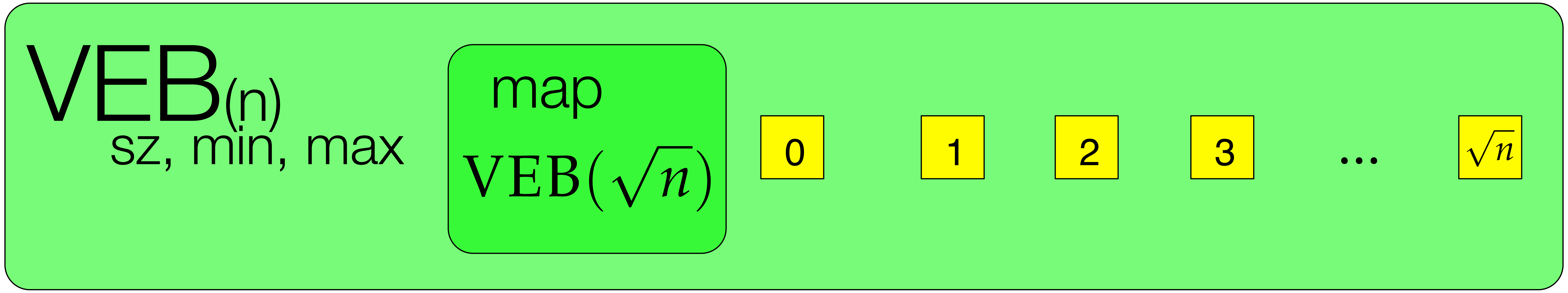
INSERT(i)

WRITE $i = a\sqrt{n} + b$

A.INSERT(B)

MAP.INSERT(A)

WHAT IS THE PROBLEM WITH THIS?



insert(i)

write $i = a\sqrt{n} + b$

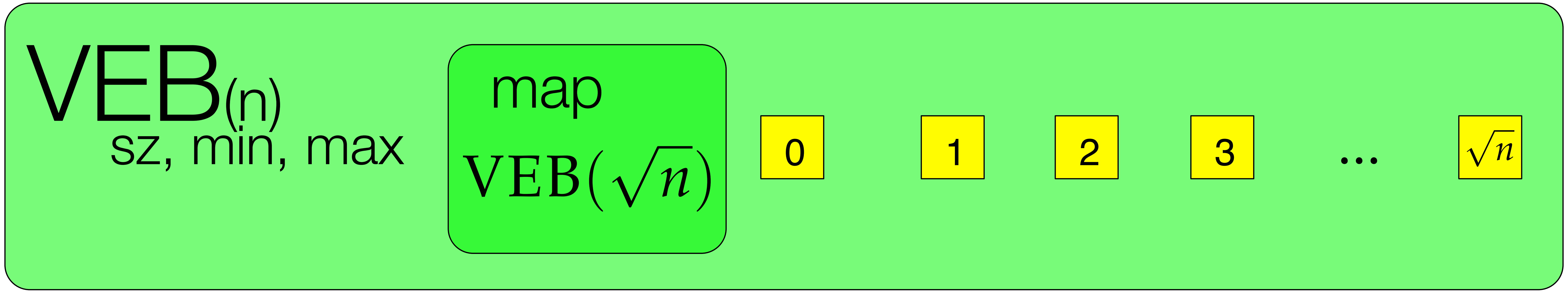
a.insert(b)

map.insert(a)

what is the problem with this?

how can we get around the problem of inserting twice?

answer: LAZY inserts. how many times do we need to insert into MAP?

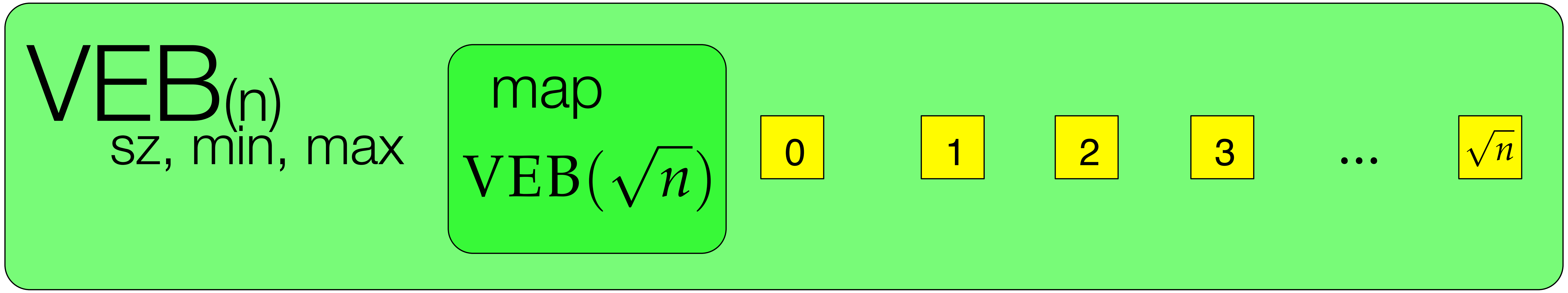


INSERT(i)

WRITE $i = a\sqrt{n} + b$

IF SZ==0 THEN

ELSE



INSERT(i)

IF SZ==0 THEN UPDATE SZ=1,MIN=MAX=i

ELSE

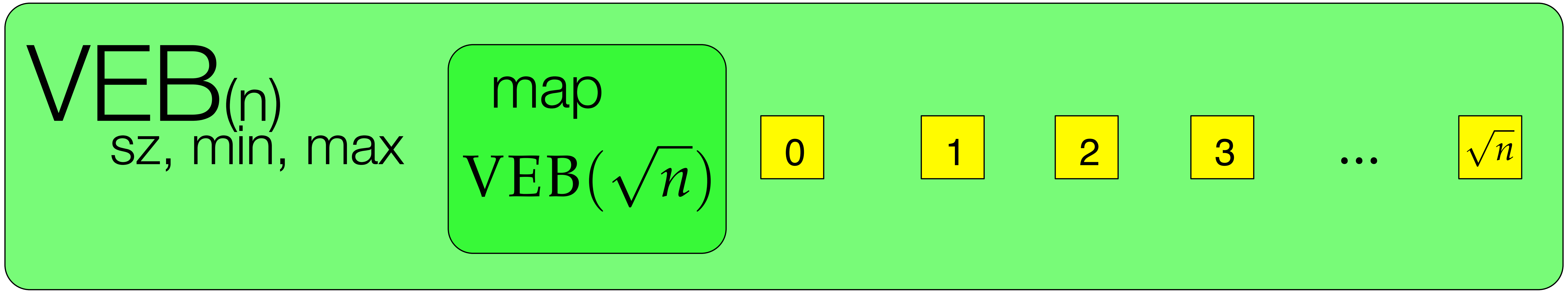
IF MIN>i SWAP(i,MIN)

WRITE $i = a\sqrt{n} + b$

IF a .SZ==0 THEN MAP .INSERT(a).

a .INSERT(b)

UPDATE SZ, MAX



INSERT(i)

IF SZ==0 THEN UPDATE SZ=1,MIN=MAX=i

ELSE

IF MIN>i SWAP(i,MIN)

WRITE $i = a\sqrt{n} + b$

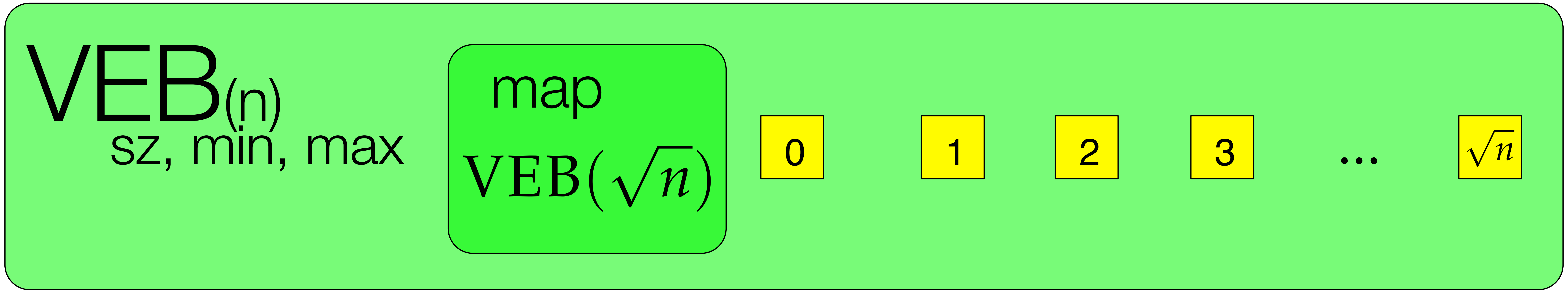
IF a .SZ==0 THEN MAP .INSERT(a).

a .INSERT(b)

UPDATE SZ, MAX

If a is empty:

then 1 full recursive call + 1 base case



INSERT(i)

IF SZ==0 THEN UPDATE SZ=1,MIN=MAX=i

ELSE

IF MIN>i SWAP(i,MIN)

WRITE $i = a\sqrt{n} + b$

IF a .SZ==0 THEN MAP .INSERT(a).

a .INSERT(b)

UPDATE SZ, MAX

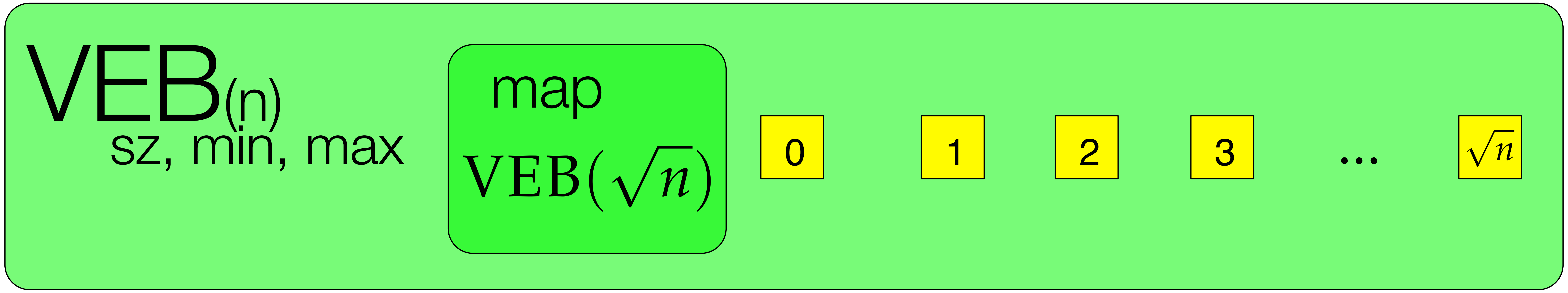
If a is empty:

then 1 full recursive call + 1 base case

If a is not empty:

Then this line does not run

but 1 full recursive call is made



INSERT(i)

IF SZ==0 THEN UPDATE SZ=1,MIN=MAX=i

ELSE

IF MIN>i SWAP(i,MIN)

WRITE $i = a\sqrt{n} + b$

IF a .SZ==0 THEN MAP .INSERT(a).

a .INSERT(b)

UPDATE SZ, MAX

If a is empty:

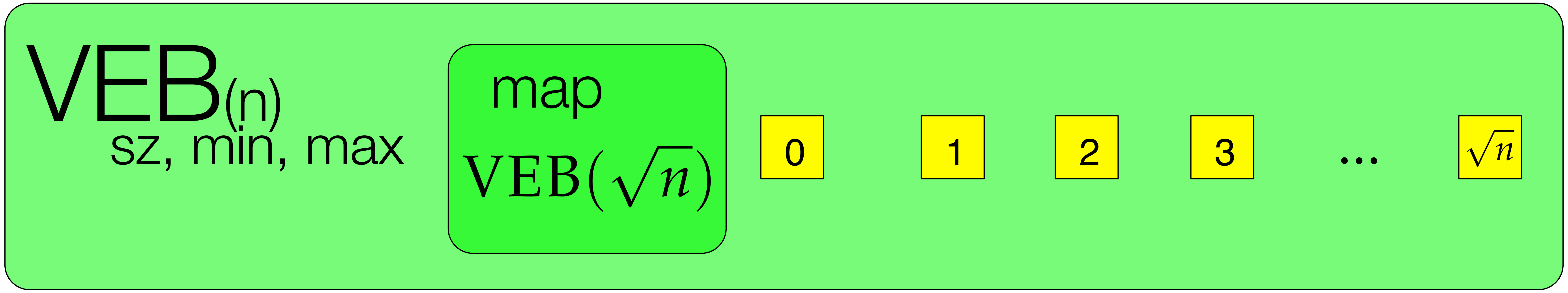
then 1 full recursive call + 1 base case

If a is not empty:

Then this line does not run

but 1 full recursive call is made

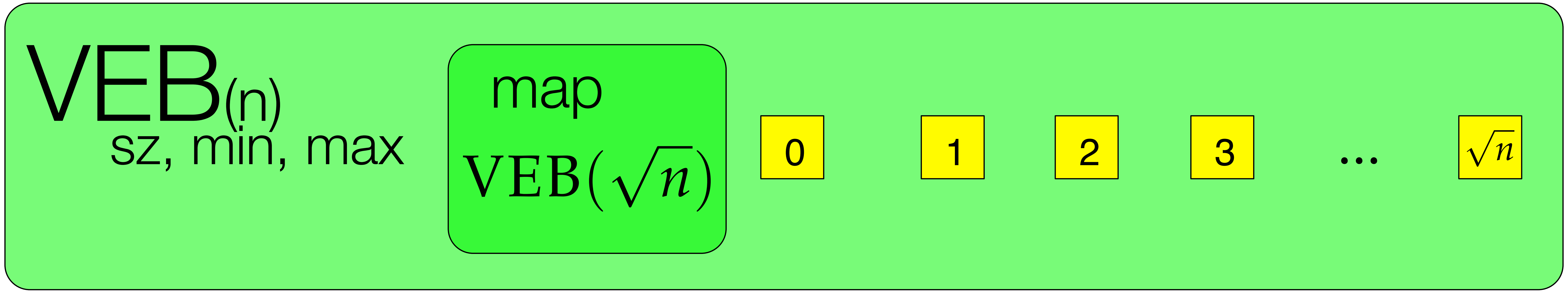
$$T(n) = T(\sqrt{n}) + \Theta(1)$$



LOOKUP(i)

WRITE $i = a\sqrt{n} + b$

We need to fix the Lookup to work with Lazy inserts.



LOOKUP(i)

WRITE $i = a\sqrt{n} + b$

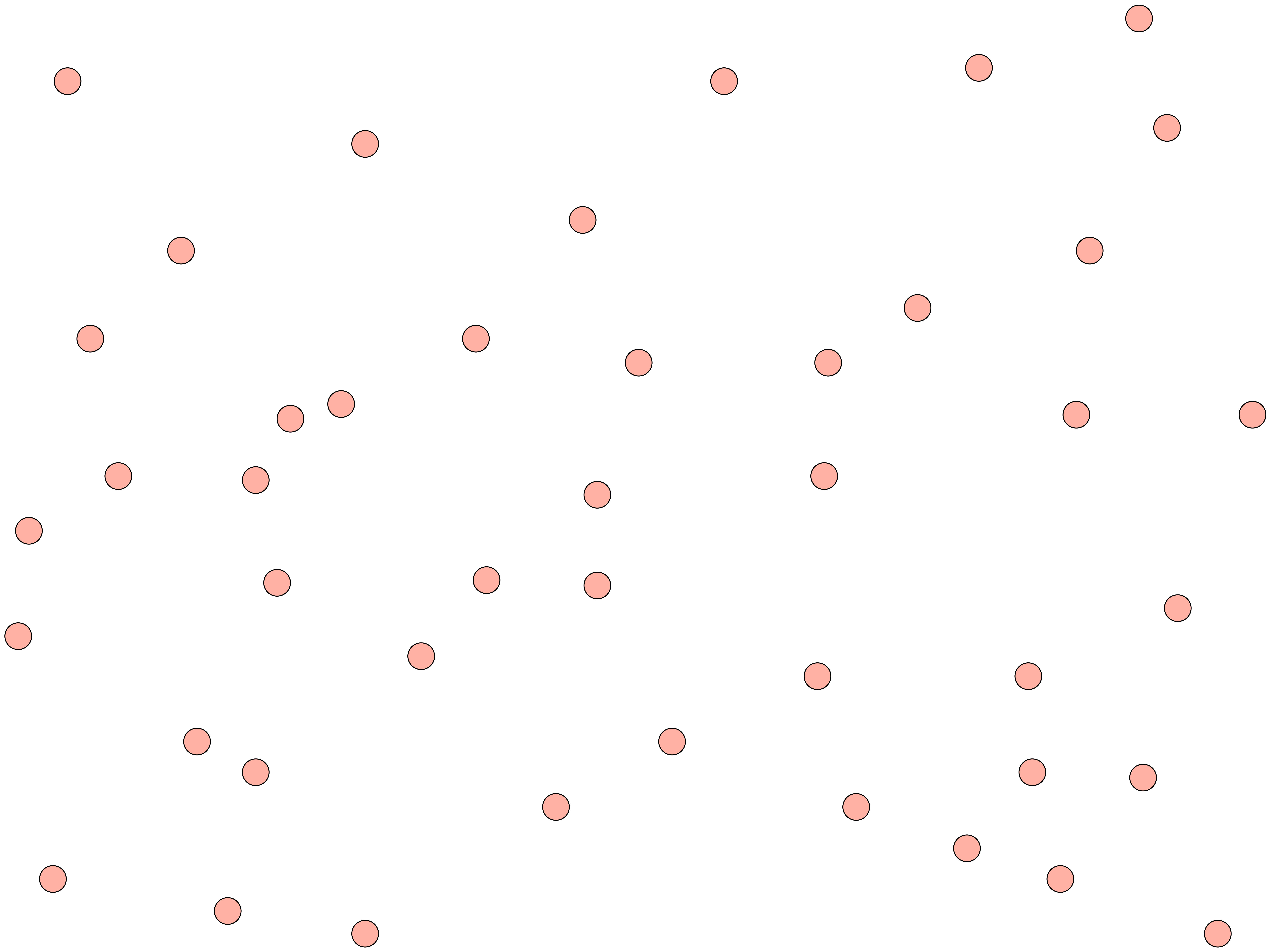
IF SIZE==0 RETURN FALSE

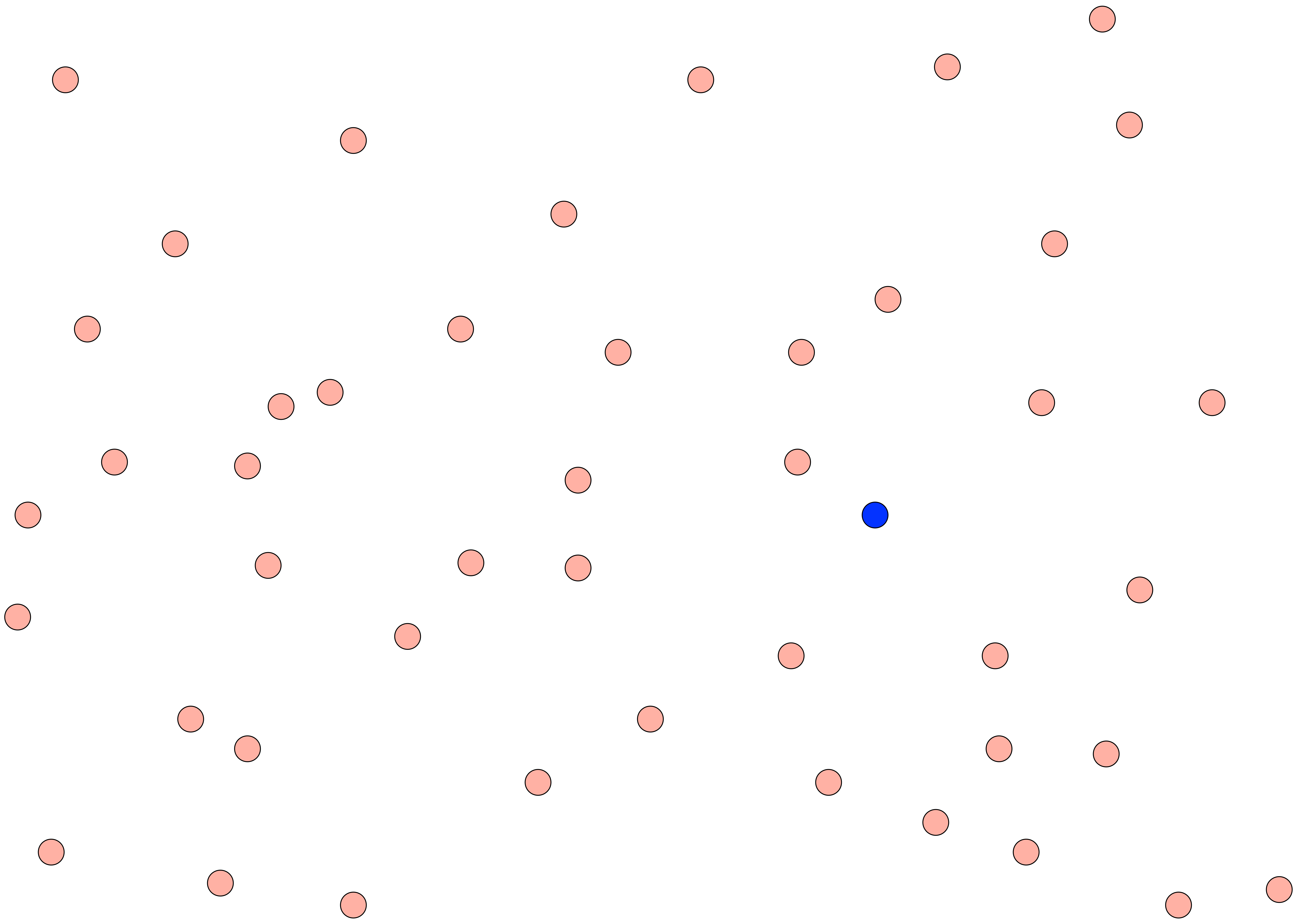
IF i==MIN RETURN TRUE

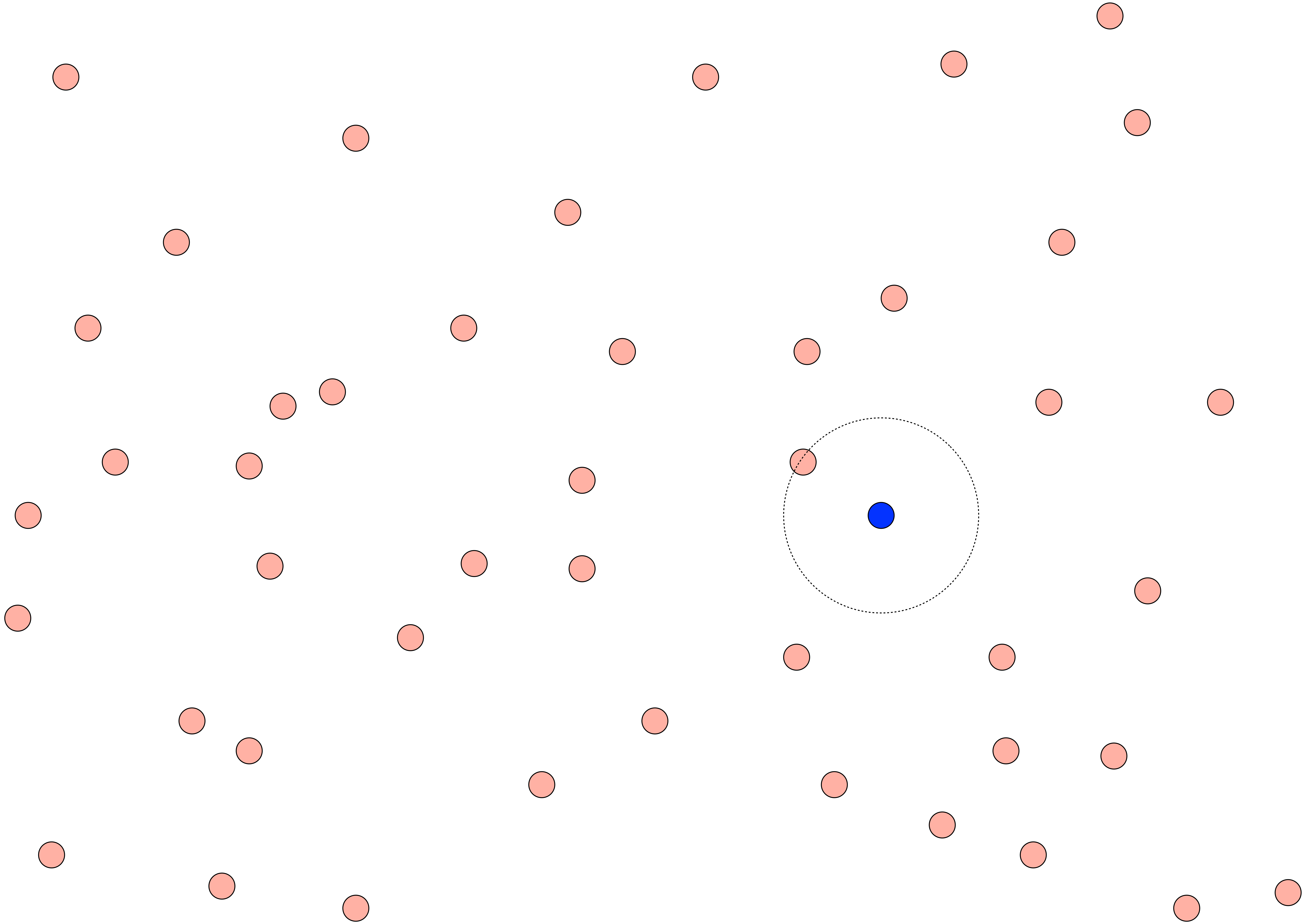
ELSE RETURN a.LOOKUP(b)

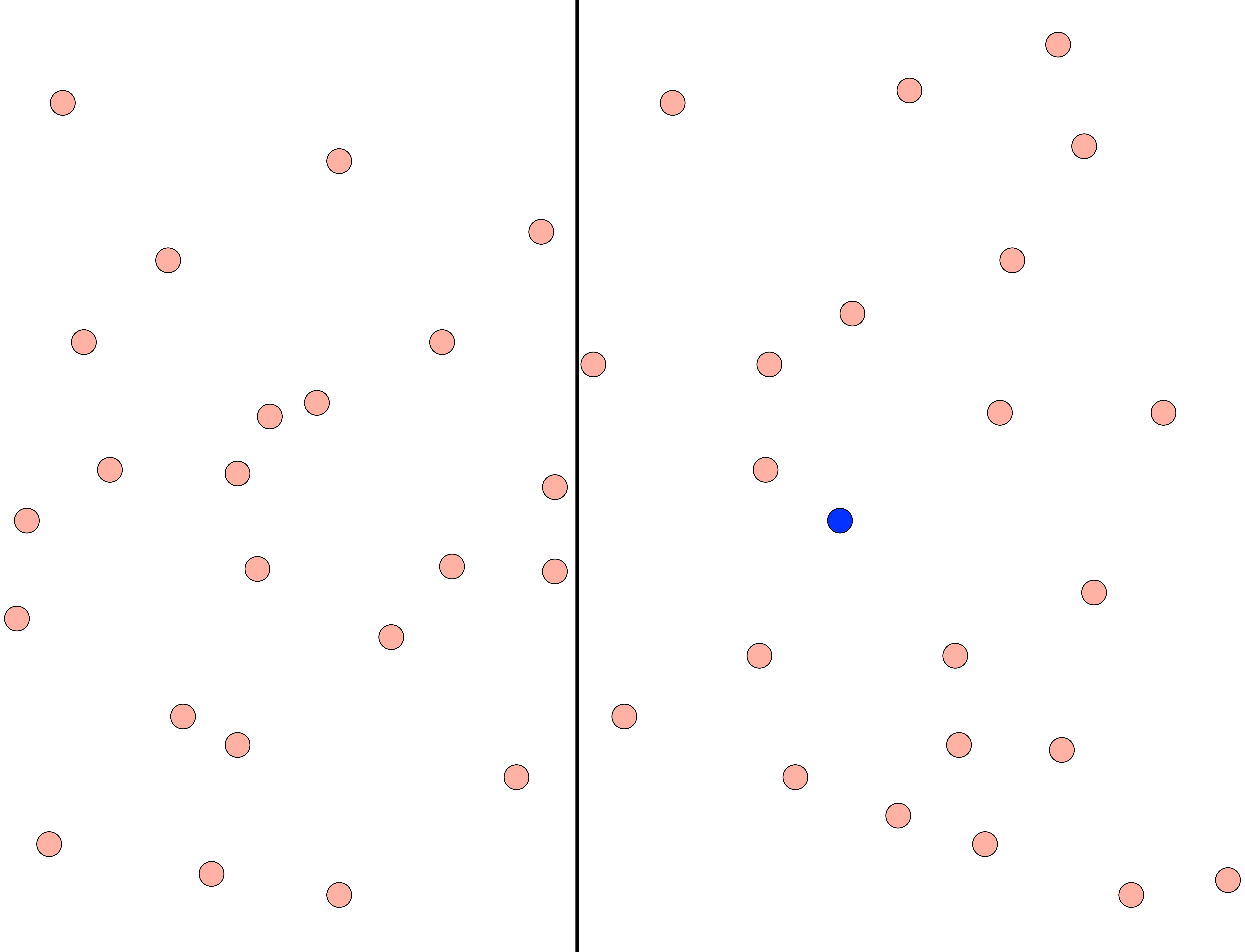
We need to fix the Lookup to work with Lazy inserts.

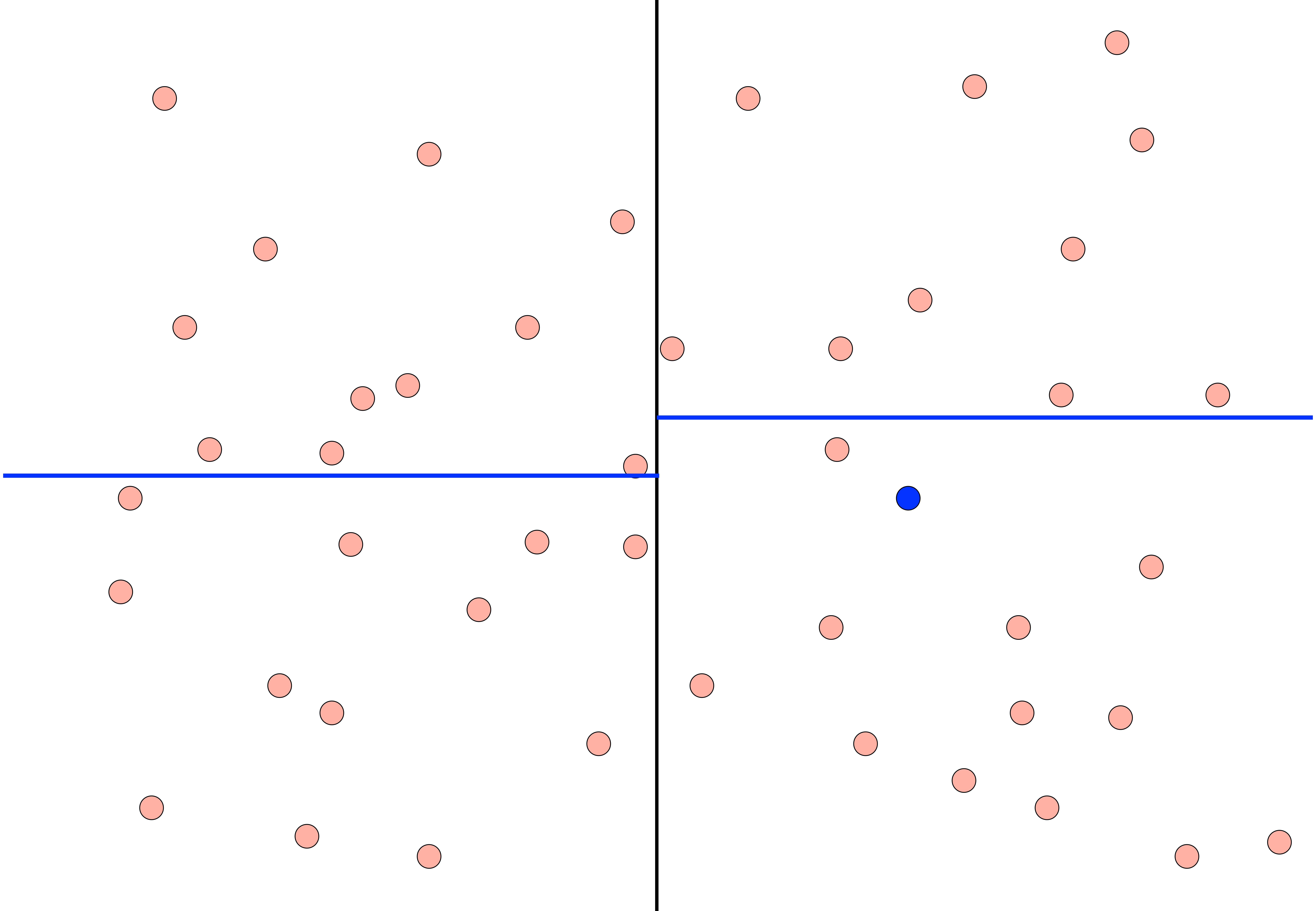
Nearest neighbor queries



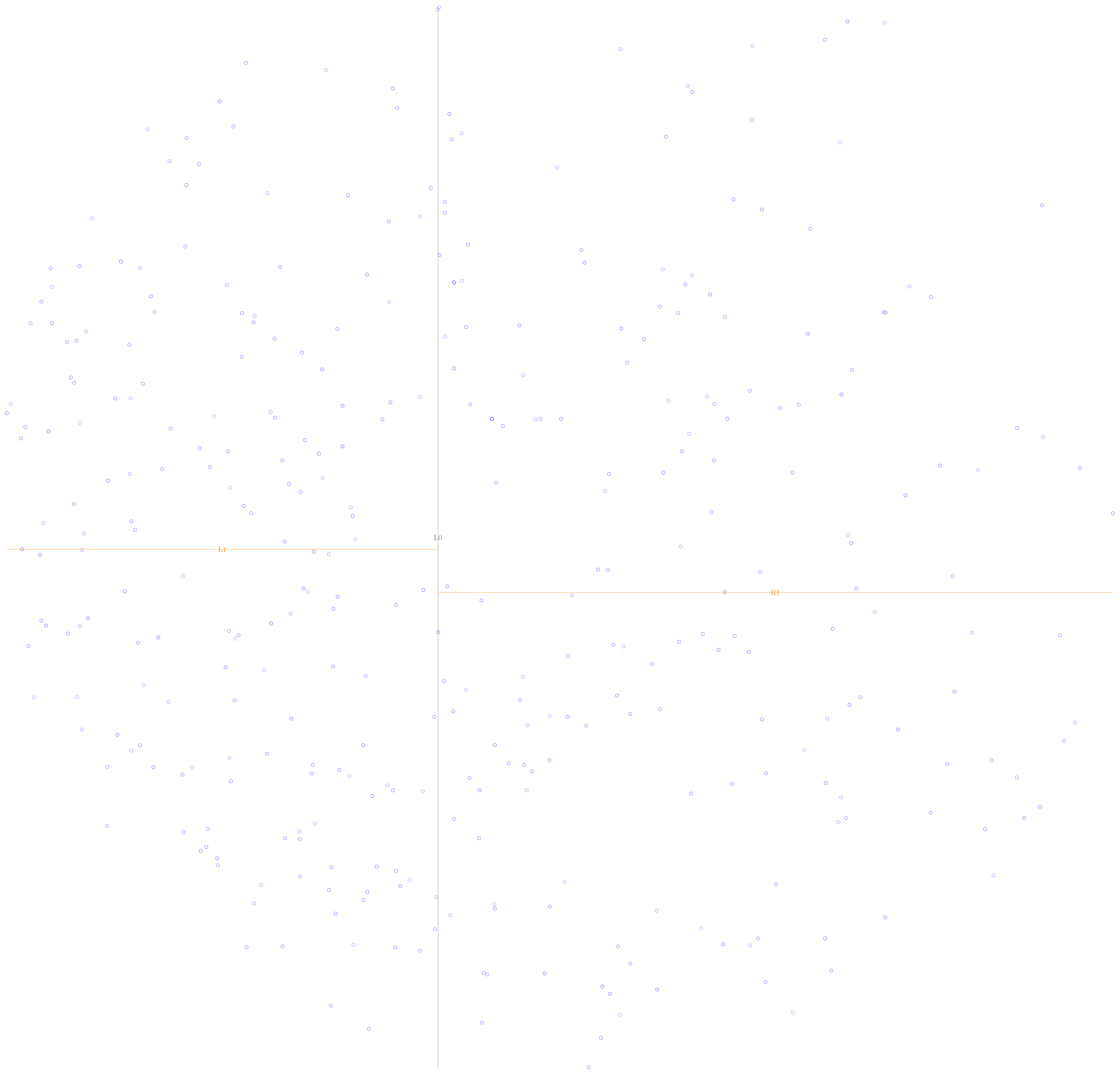


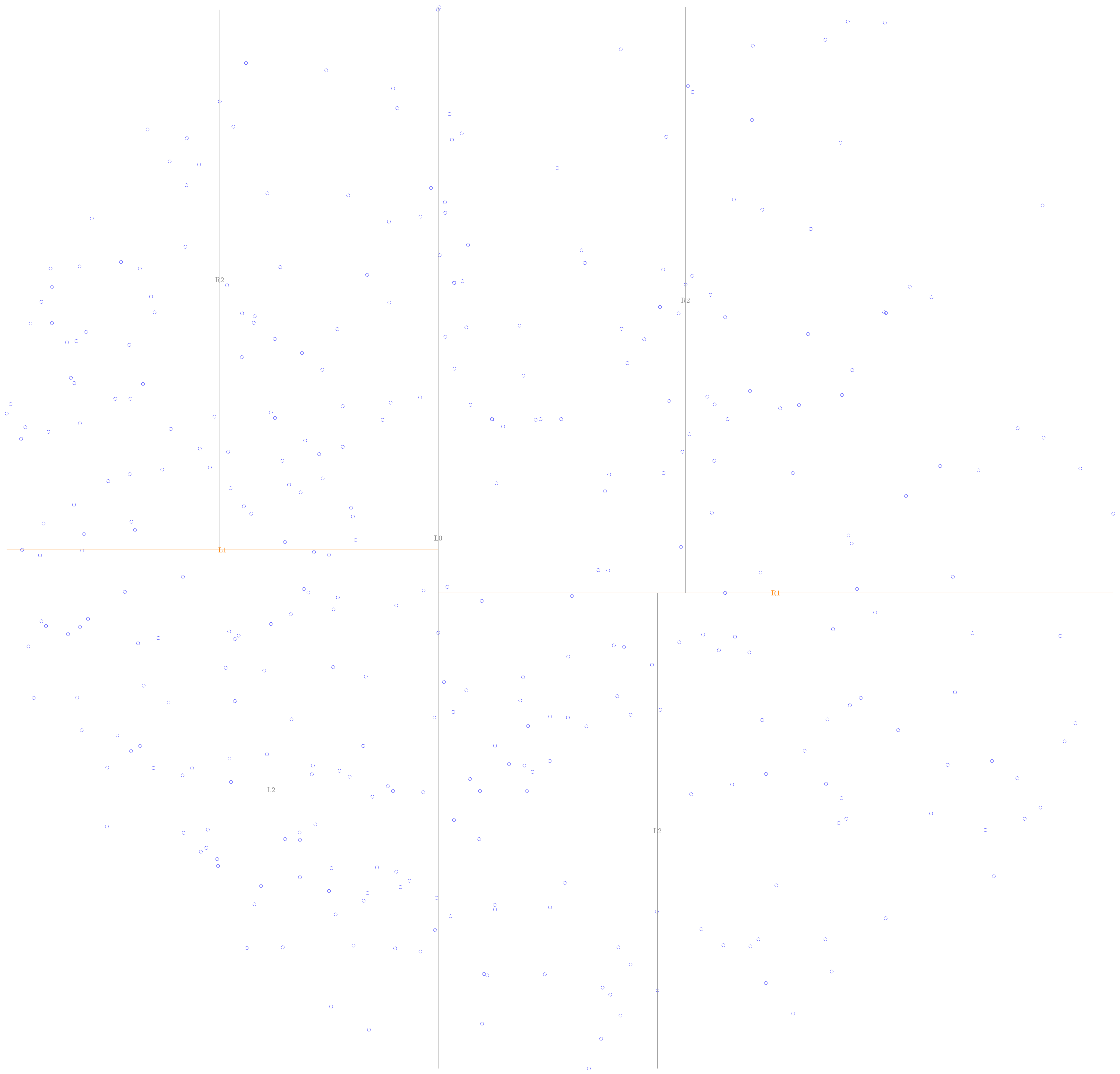


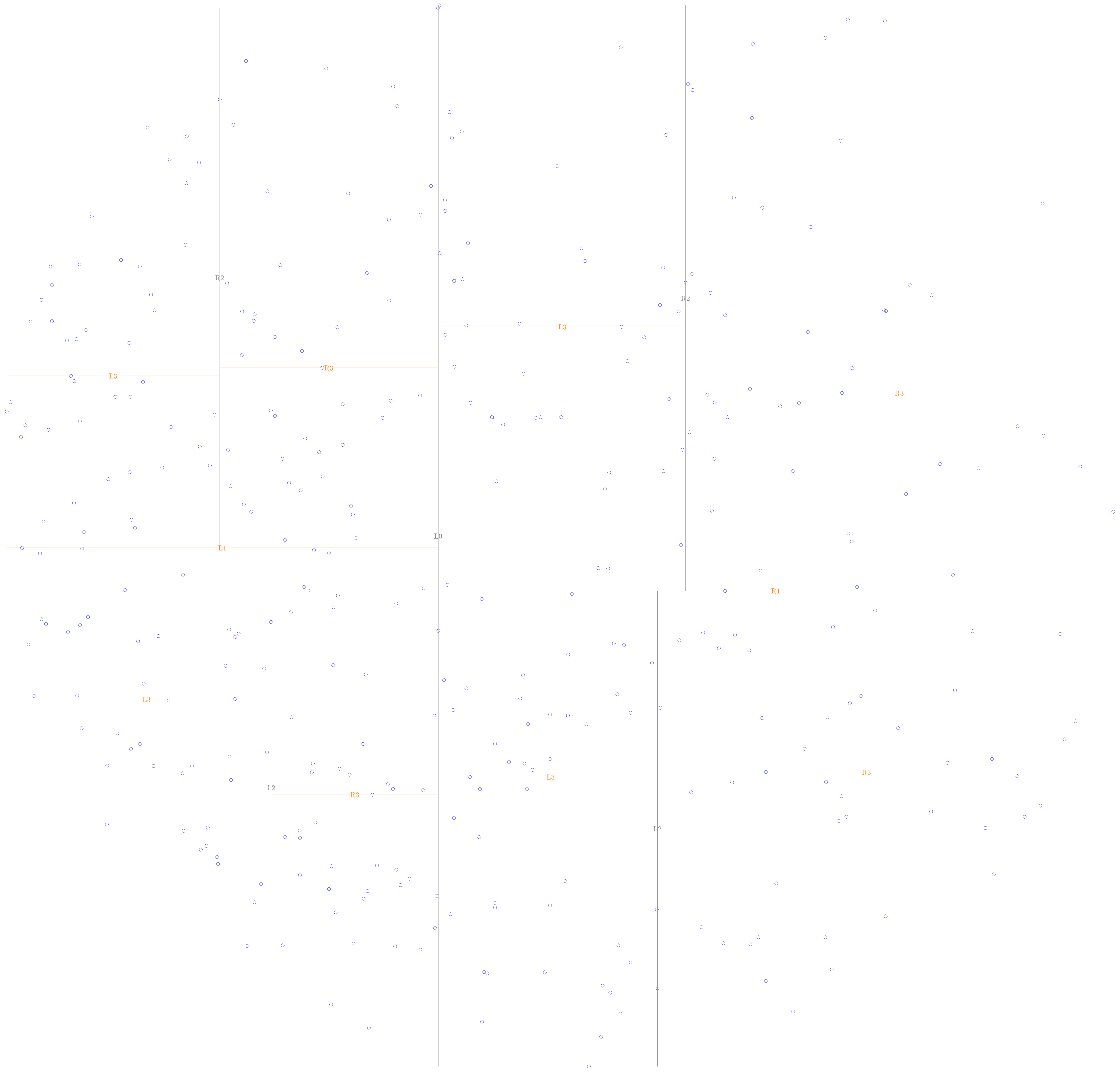




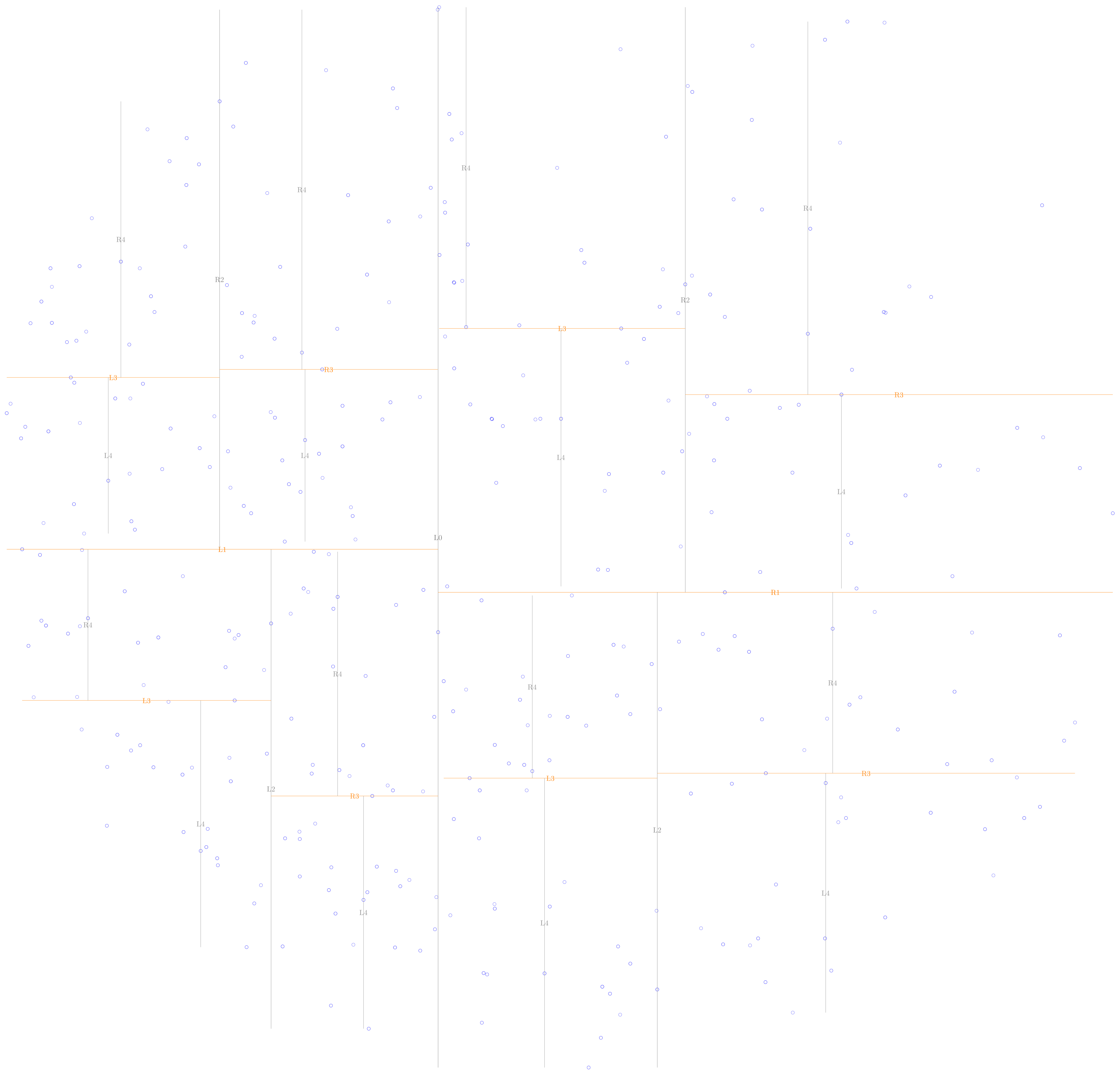

L0

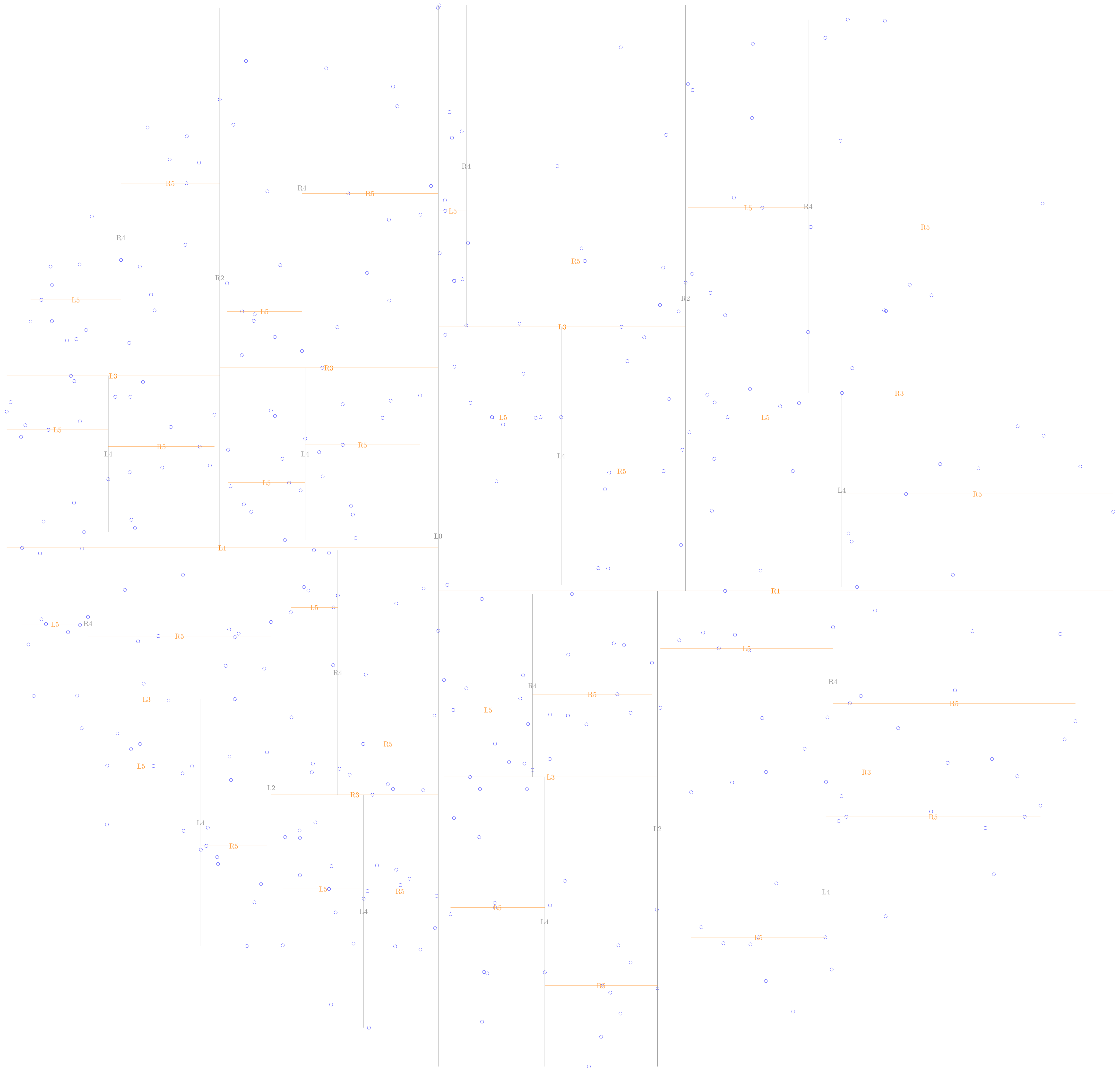

L0
L1
R1


R2
R2
L0
L1
R1
L2
L2



R2
L3
R3
L3
R2
R3
L0
L1
R1
L3
L2
R3
L3
R3
L2

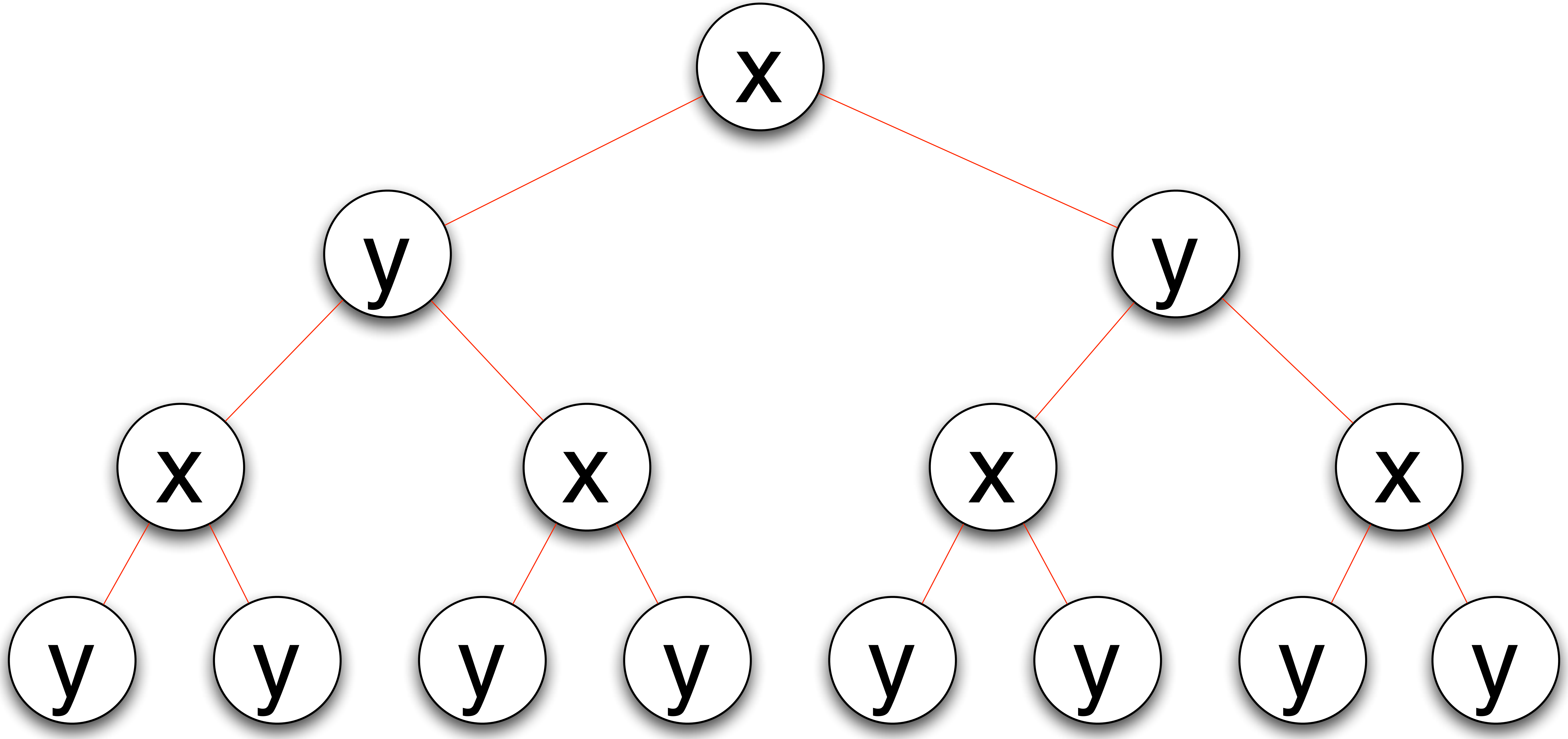


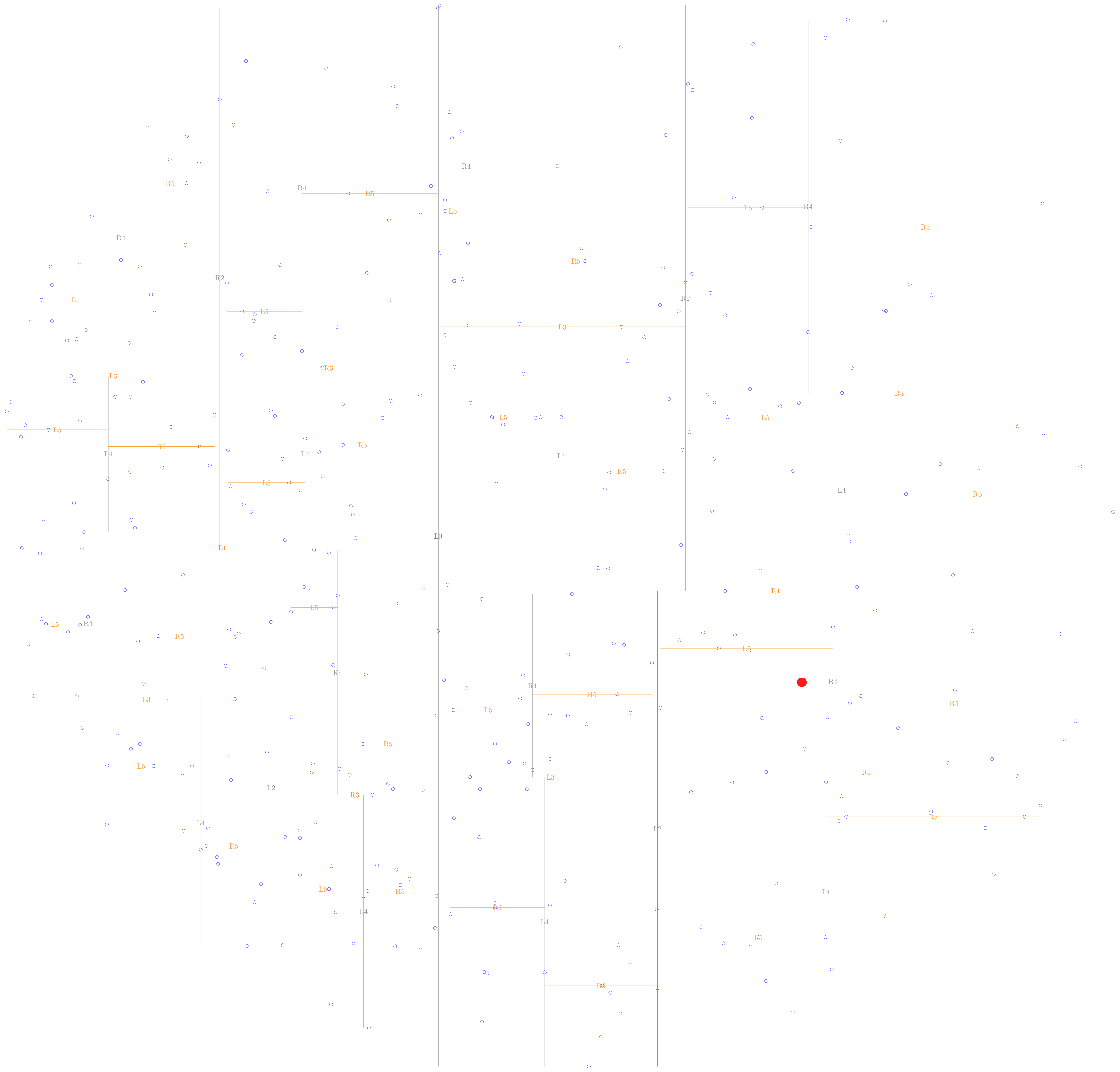


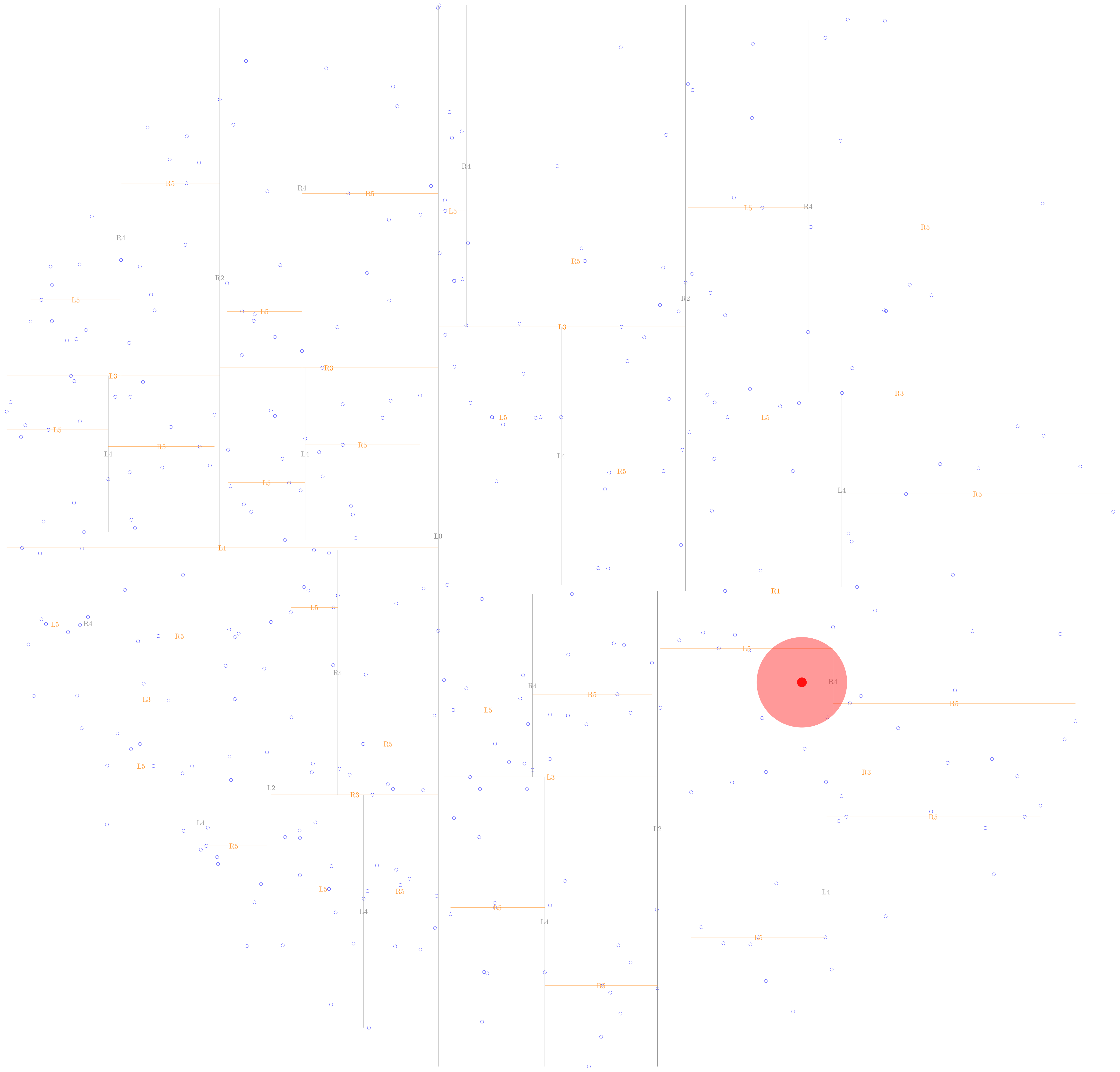
KD-Tree

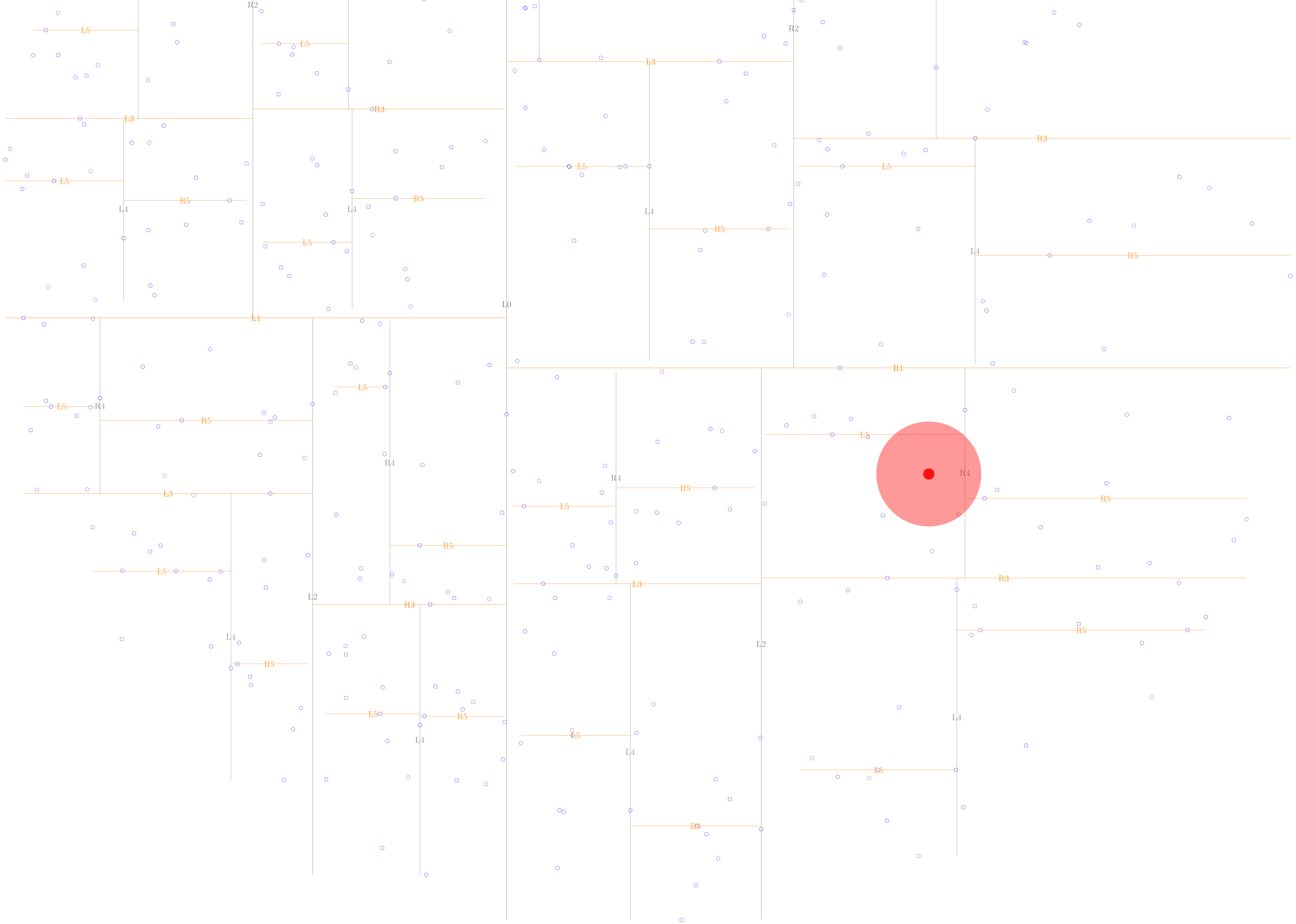
Each node in tree maintains variable “box”

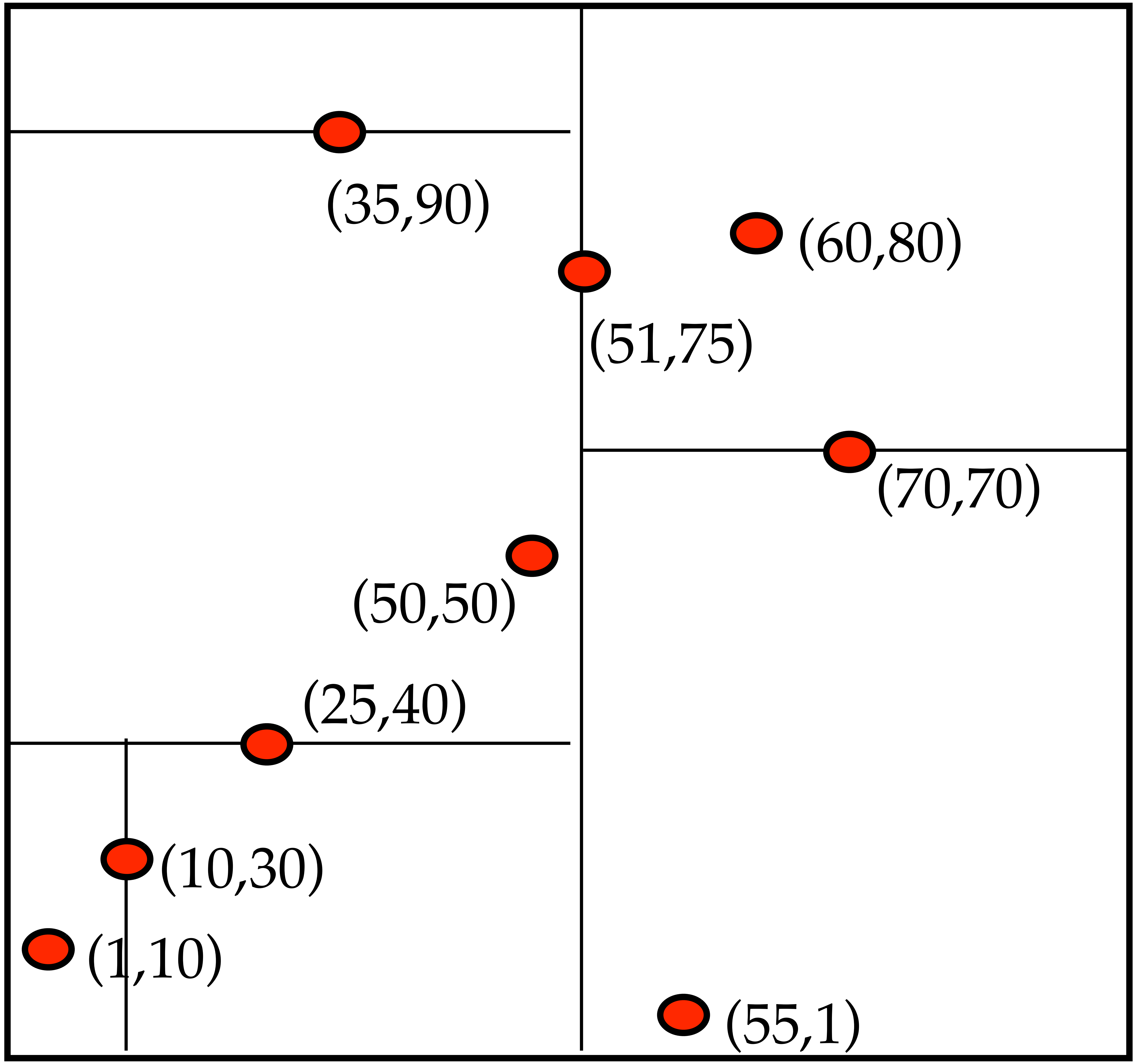
```
node {
 rect box
 point split
 node* left
 node* right
}
```


x
y
y
x
x
x
x
y
y
y
y
y
y
y
y






L0
L1
R1
L2
R2
L3
R3
L4
R4
L5
R5


(35,90)
(60,80)
(51,75)
(70,70)
(50,50)
(25,40)
(10,30)
(1,10)
(55,1)

```
NN(q, tree, dir, closest-so-far)

    if empty(tree) or  dist(q, tree.box)>closest return

    if dist(q,tree.root) < closest { update closest}

    if q.dir < tree.dir {
     NN(q, tree.left, nextdir, closest)
     NN(q, tree.right, nextdir, closest)
    } else {
     NN(q, tree.left, nextdir, closest)
     NN(q, tree.right, nextdir, closest)
    }
```