5800
data structures

apr8/apr11 2022
shelat

Dictionary

data structure

Key-value mapping.

Insert (Key, value) : creates the association b/w key-value.

Lookup (Key) : returns (key, value) if key was previously inserted into the data structure.

Delete(Key) : removes key.

Findnext (Key) : Finds the "next" lexographic key (pair) in the data structure that is greater than (key)

Dictionary

insert(key, value)

delete(key)

lookup(key)

findnext(key)

Dictionary

standard solution: hashtable

- insert(key, value)
- delete(key)
- lookup(key)
- findnext(key)

key value

d = { }

d['hello'] = 170

expected O(1) time, but worse case O(n) time

findnext: O(n) scan of all the keys.

Python DICTIONARY

$N = 2^k$

hash function h

$h(key) :\to [0, N-1]$

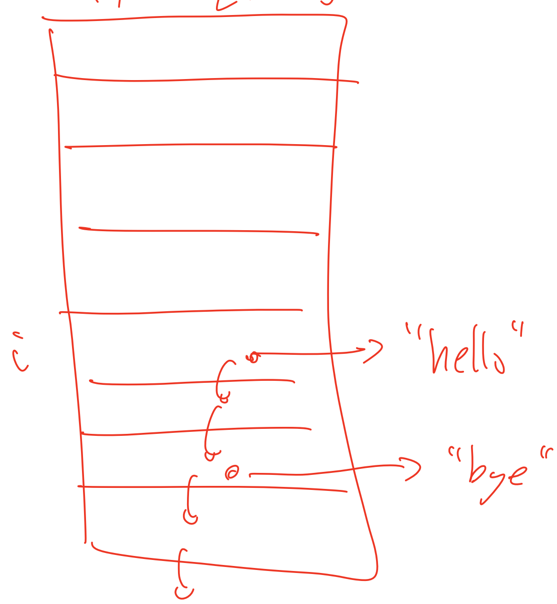
h("hello") = i

h("bye") = i

might be

$(5*i + p + 1) \% N.$

TABLE [1...N]



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

```

empty dictionary

insert (k, v) into the dictionary

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

2003

```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) : O(loglog n)

delete(key) "

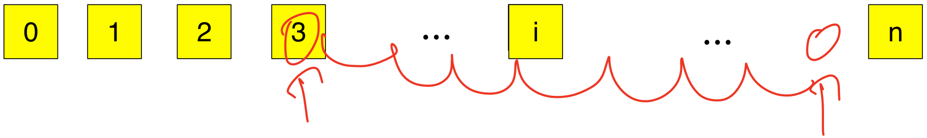
lookup(key) "

findnext(key) "

worst case performance!!

A simple solution: bit vector

Maintain an array of bits



insert(key, value) : easy O(1)

delete(key) O(1)

lookup(key) O(1)

findnext(key)

min(·) → O(n)

max(·)

Can we do better than O(n) findnext?

van emde Boas Q

THE BIG IDEA:

① recursive data structure

the data structure that handles the universe [1...n] consists of many smaller data structures that handle the universe [1...$\sqrt{n}$]

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.

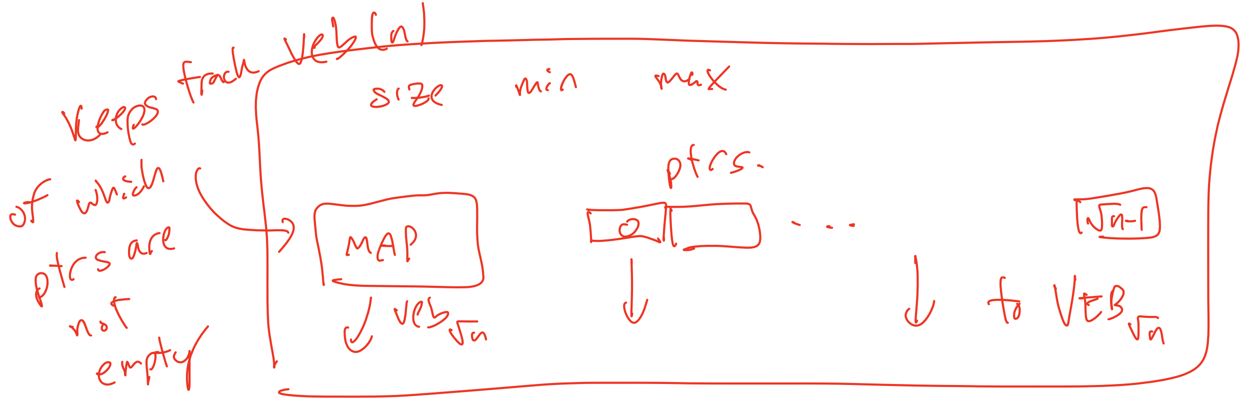
VEB queue

$N = 2^k$

VEB(N) size min max

Base case: 2 bit vector for 1...2

recursive case:



VEB queue

VEB(N)

sz, min, max

VEB queue

VEB(N)

SZ, MIN, MAX

BASE CASE: ~~1~~ 4 BIT VECTOR.

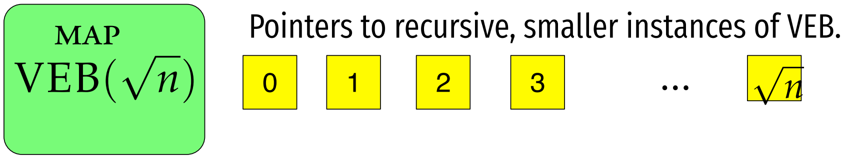
VEB queue

VEB(N)

SZ, MIN, MAX

BASE CASE: ~~1~~ 4 BIT VECTOR.

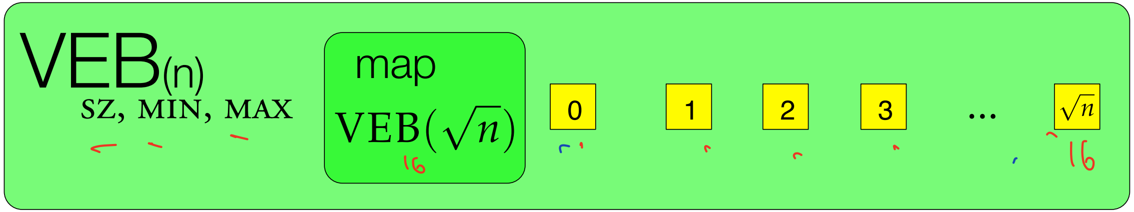
NORMAL CASE:



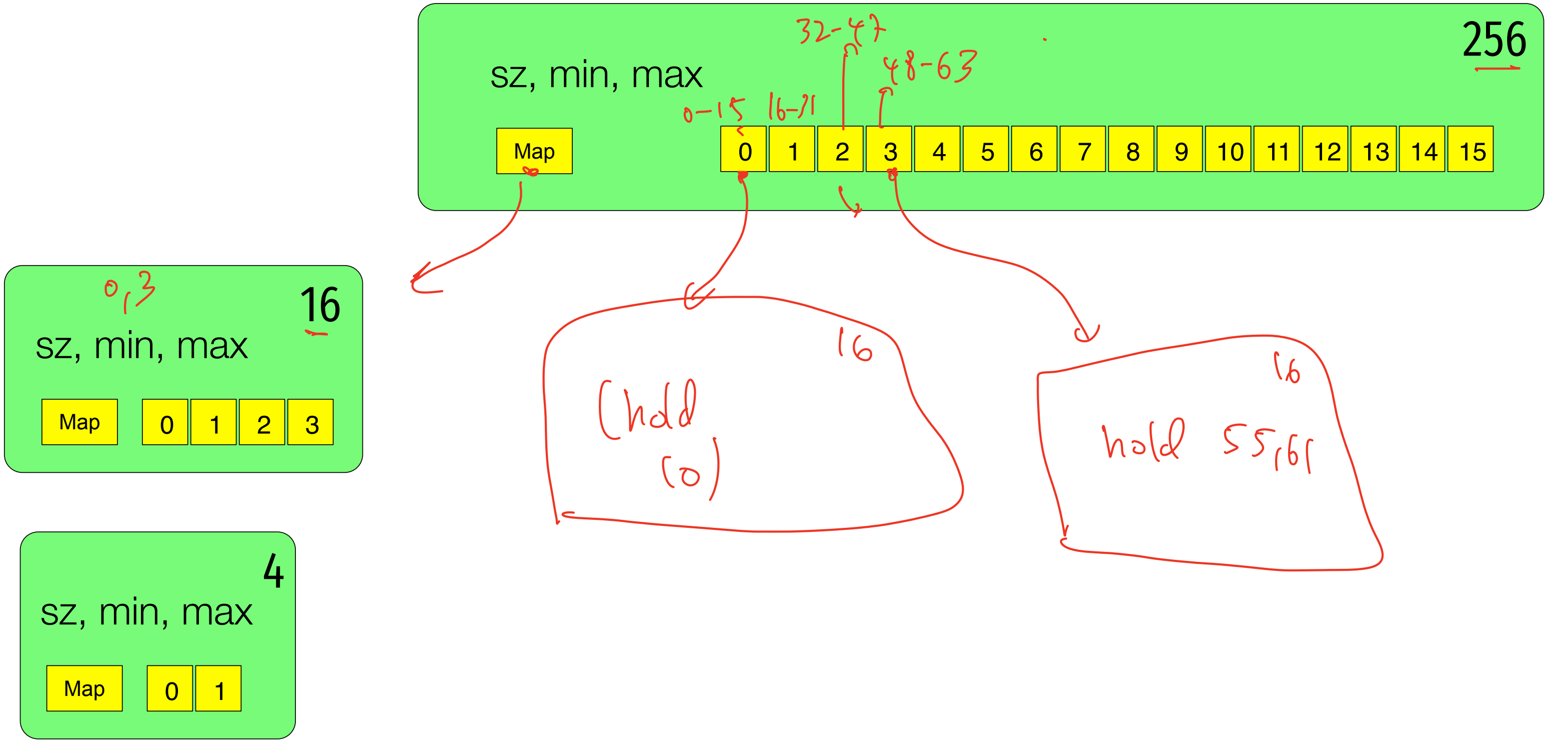
Keeps track of which ptrs are not empty.

EXAMPLE:

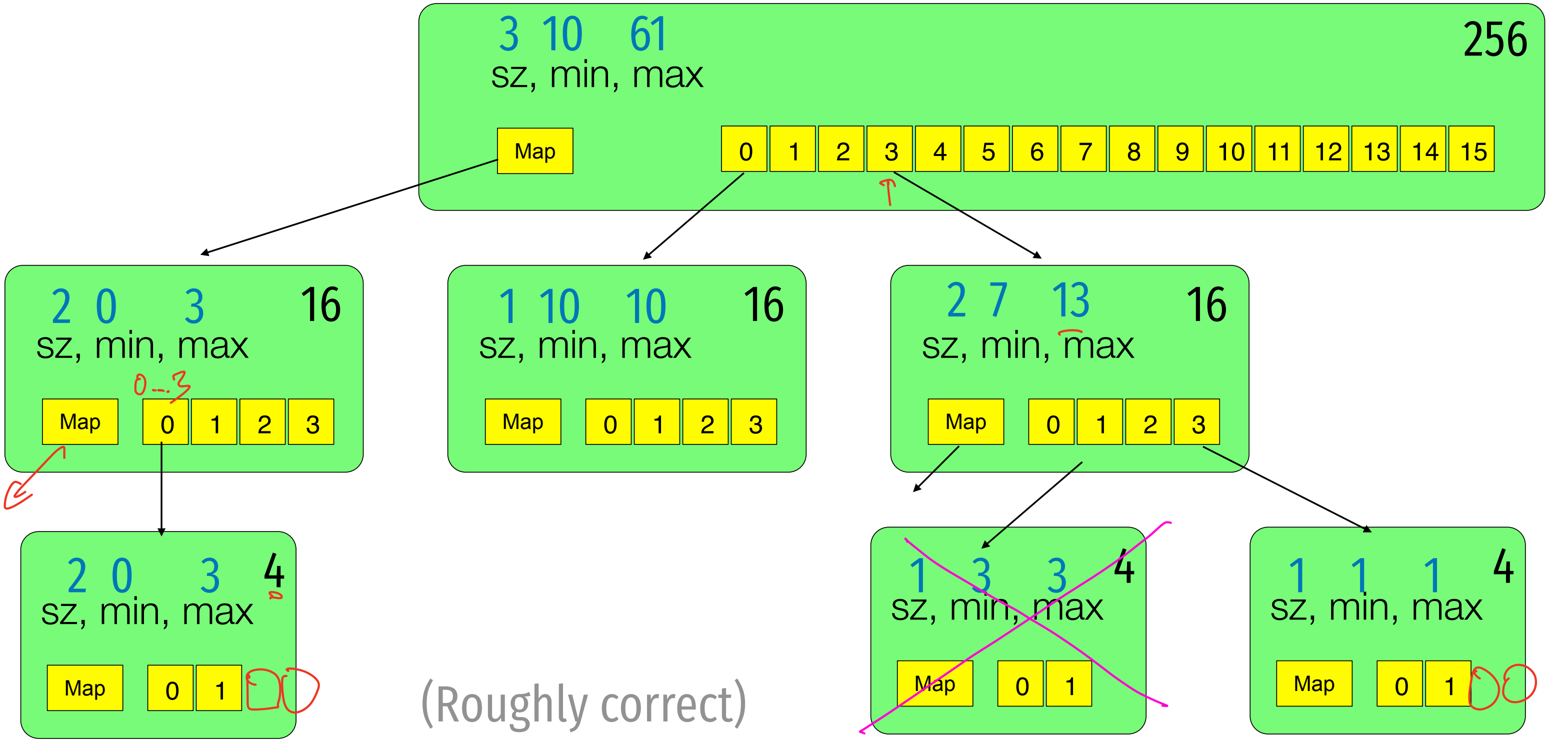
$n = 256$



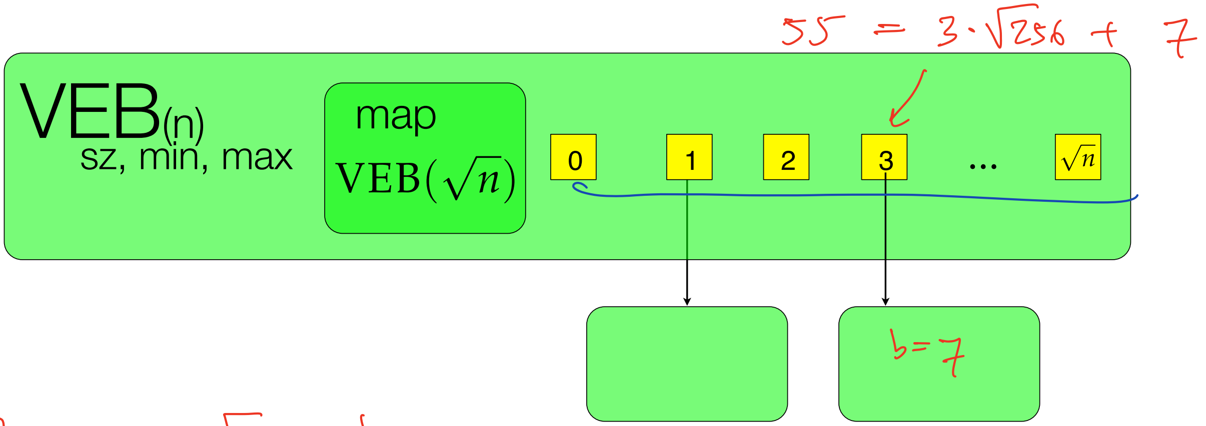
Example n=256, keys={10,55,61}



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



(Roughly correct)



LOOKUP(i)

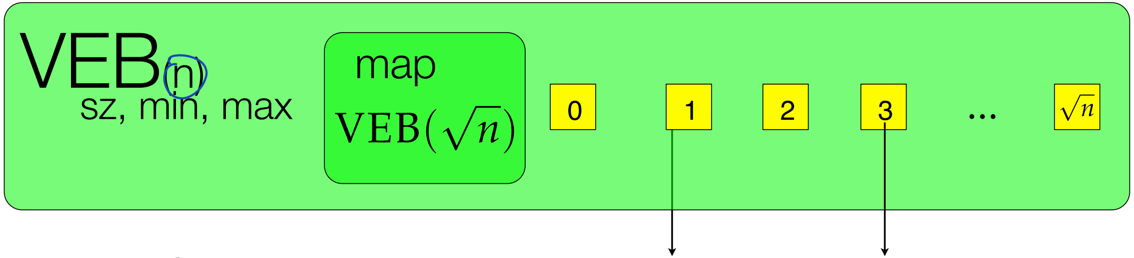
O(1) write $i = a\cdot\sqrt{n} + b$ → such that $a, b \in [0..\sqrt{n}-1]$

O(1) if BASE case then check the bit vector.

If size = 0 or a.size = 0 return false.

else return a.Lookup(b)

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



N = size of the universe.

LOOKUP(i)

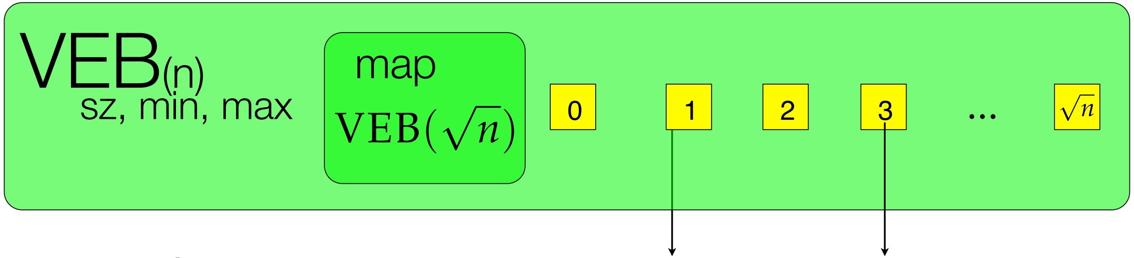
WRITE $i = a\sqrt{n} + b$

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

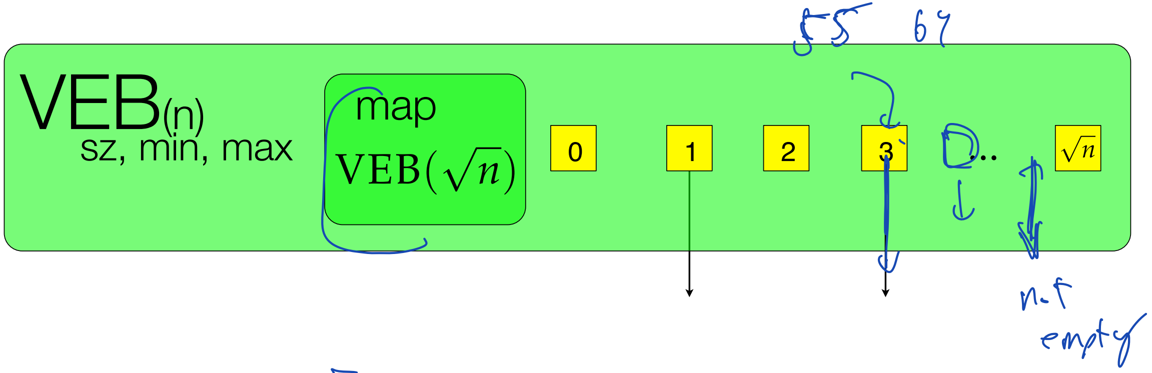
IF <BASE CASE>: CHECK BIT VECTOR

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.)

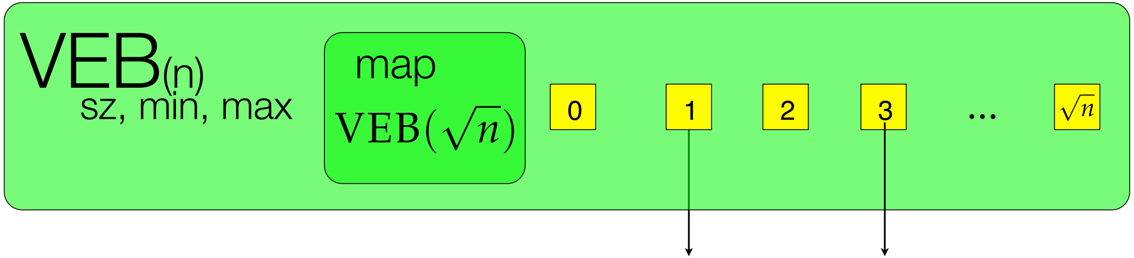


FINDNEXT(i)

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

2 cases. (1) either bucket a contains the next key

(2) or the next bucket at this level contains the next key



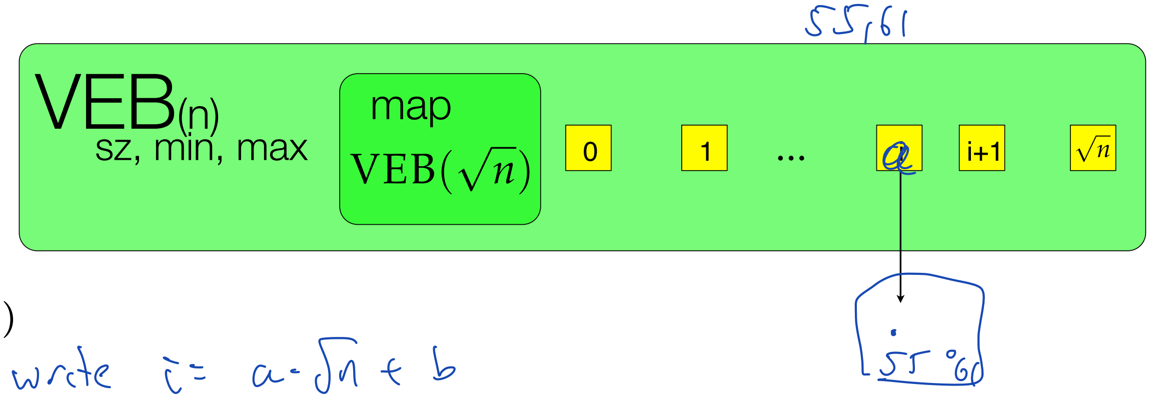
FINDNEXT(i)

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)

$\Theta(1)$ write $i = a \cdot \sqrt{n} + b$

< handle base case with scanning >

If a.max ≥ b then

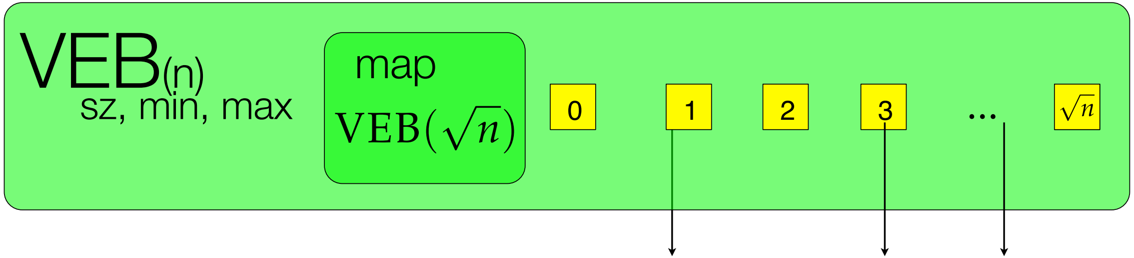
  return a.findnext(b)

else

  return map.findnext(a).min

if no such integer return ∞.

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



FINDNEXT(i)

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

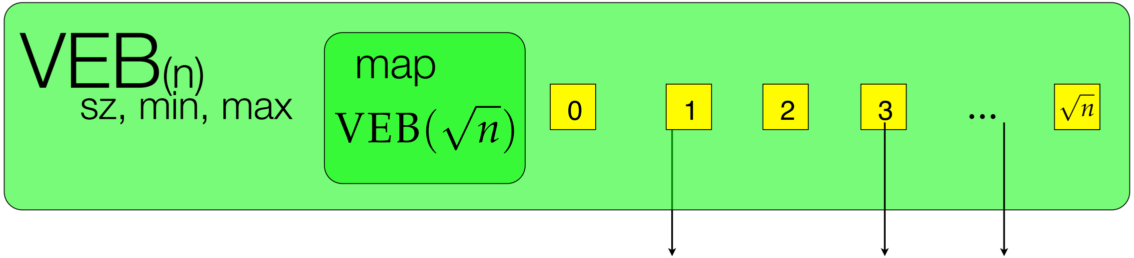
<BASE CASE IF SIZE IS ZERO>

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)

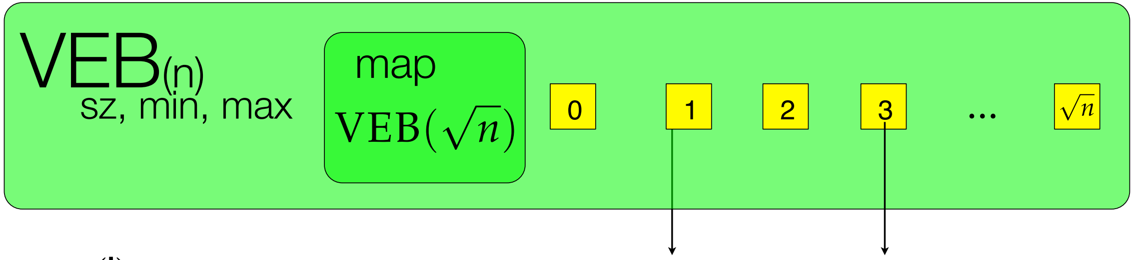
ELSE

RETURN MAP .FINDNEXT(a) .MIN

Running time:

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

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

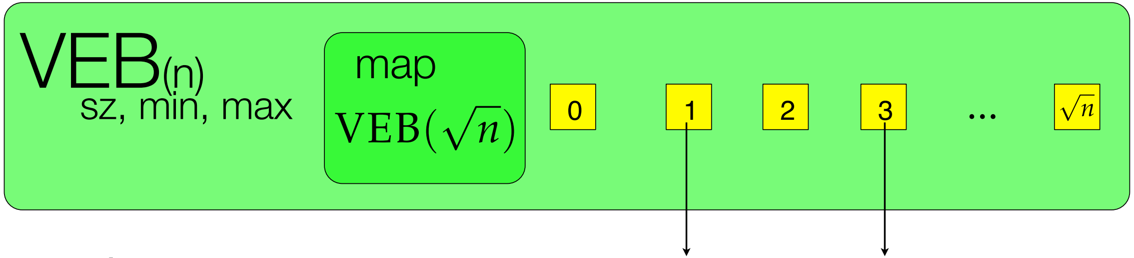


INSERT(i)

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

a.insert(b)

map.insert(a)



INSERT(i)

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

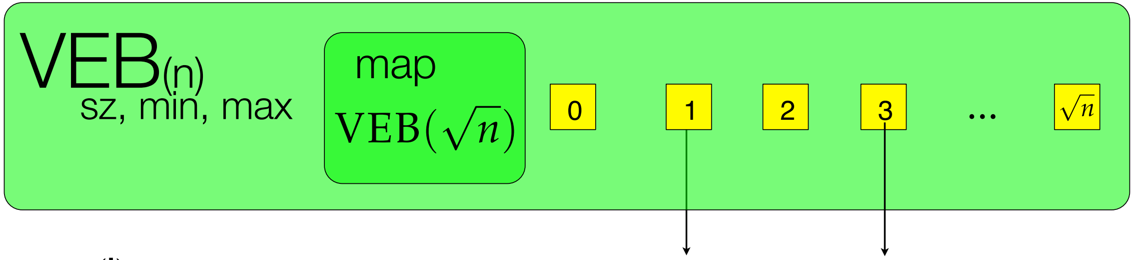
A.INSERT(B)

MAP.INSERT(A)

It is too much work to perform 2 inserts here!!

$$T(n) = 2T(\sqrt{n}) + \Theta(1)$$

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



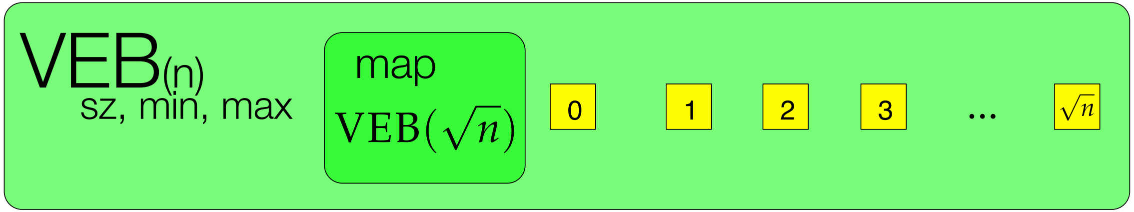
INSERT(i)

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

A.INSERT(B)

MAP.INSERT(A)

WHAT IS THE PROBLEM WITH THIS?



insert(i)

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

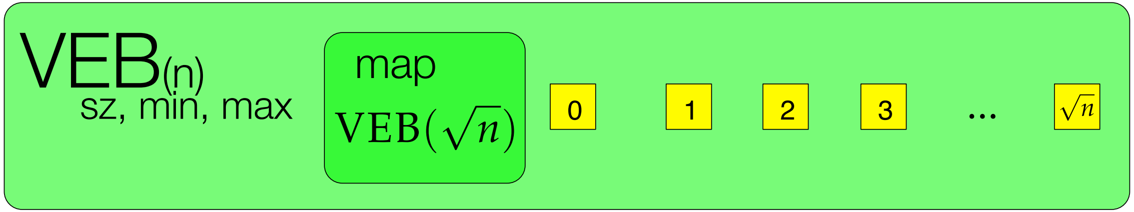
a.insert(b)

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 update sz=1 min=max=i

ELSE

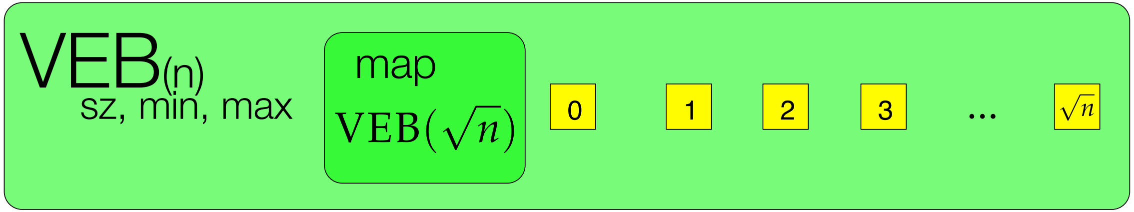
if min>i swap(i,min)

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

if a.sz==0 then map.insert(a)

a.insert(b).

update max, sz.



INSERT(i)

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

ELSE

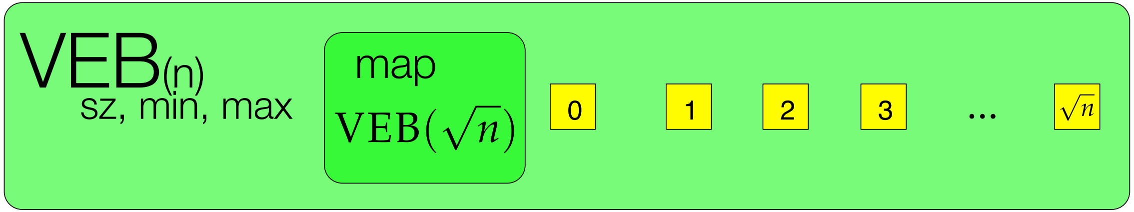
IF MIN>i SWAP(i,MIN)

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

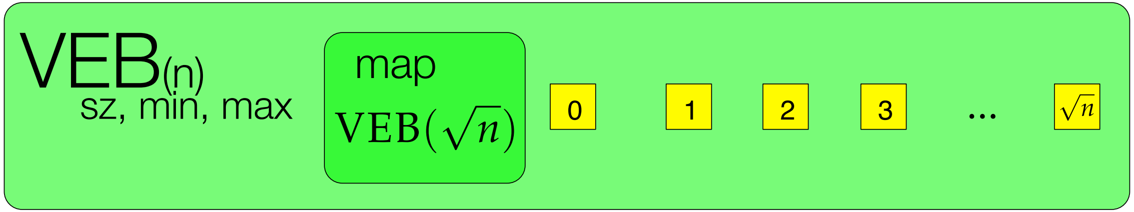
If a is empty:

then 1 full recursive call + 1 base case

If a is not empty:

we just run this case

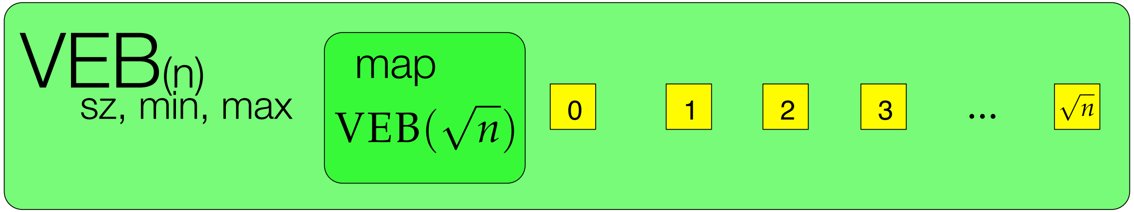
$$T(u) = T(\sqrt{u}) + \Theta(1) = \Theta(\log \log u)$$




```
INSERT(i)
  IF SZ==0 THEN UPDATE SZ=1,MIN=MAX=i
  ELSE
      IF MIN>i SWAP(i,MIN)
      WRITE  i = a√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).

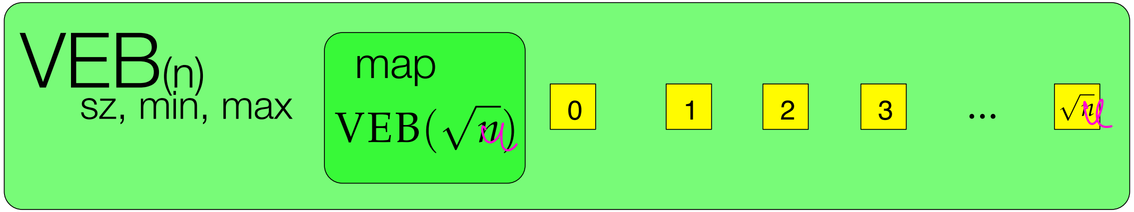
a .INSERT(b)

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)

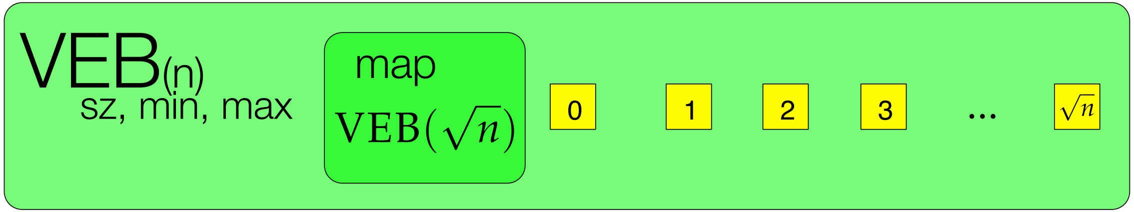
WRITE $i = a\sqrt{n} + b$

If sz == 0 return false

If i == min return true

else return a.lookup(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)

$u = \lceil 2^{b} \rceil$

$u \sim 2^{32}$

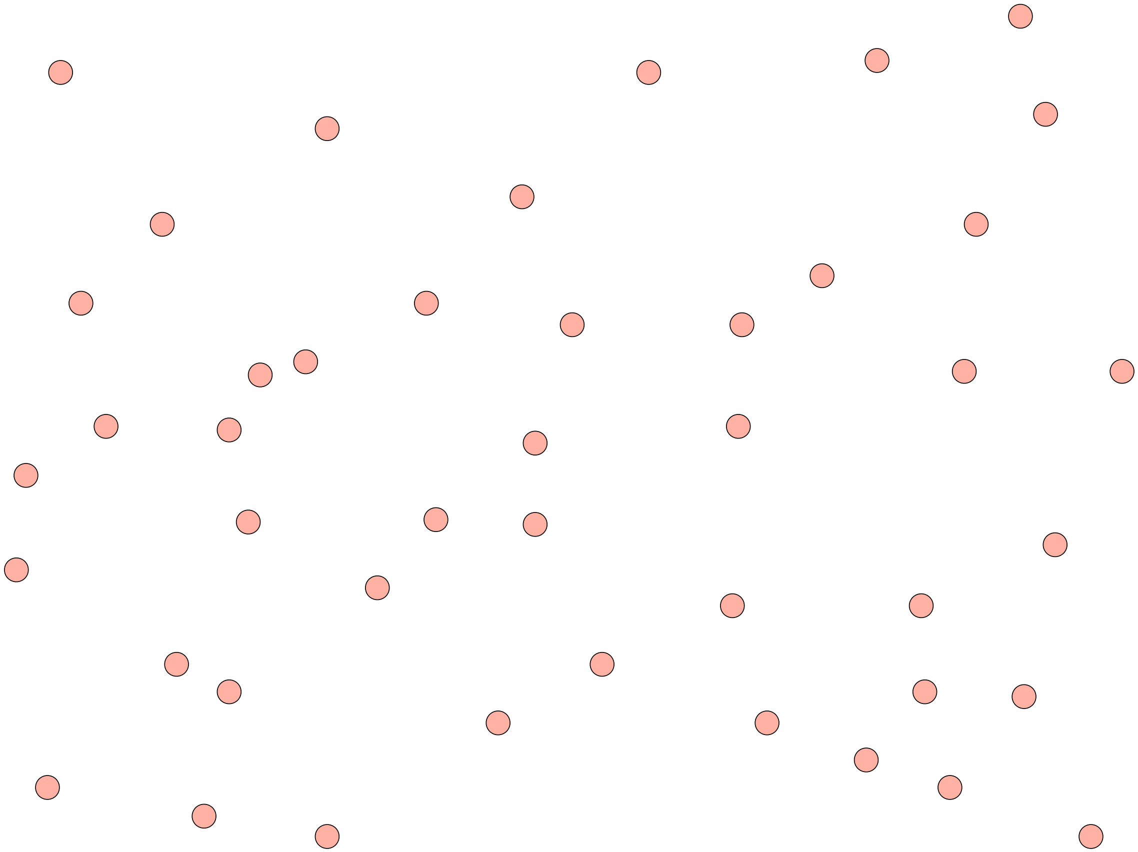
$\log\log(u) =$

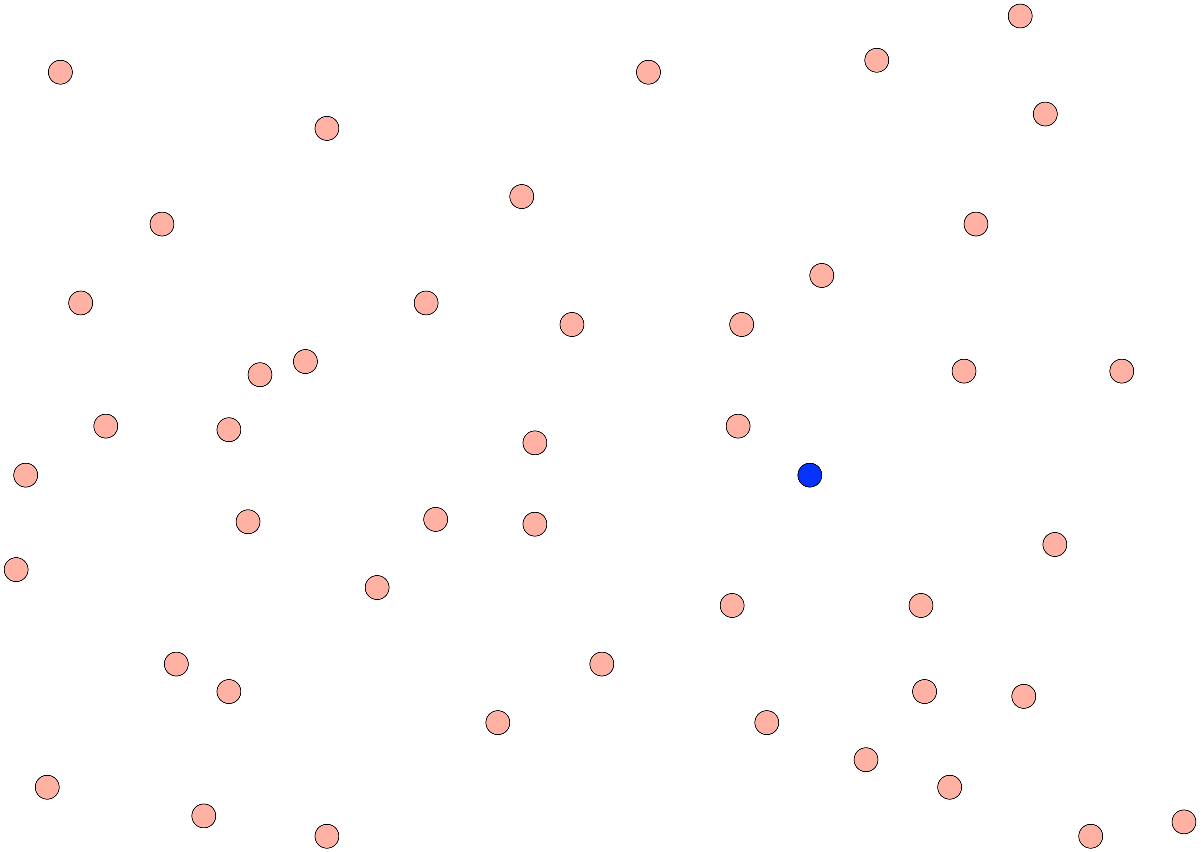
$\log(b)$

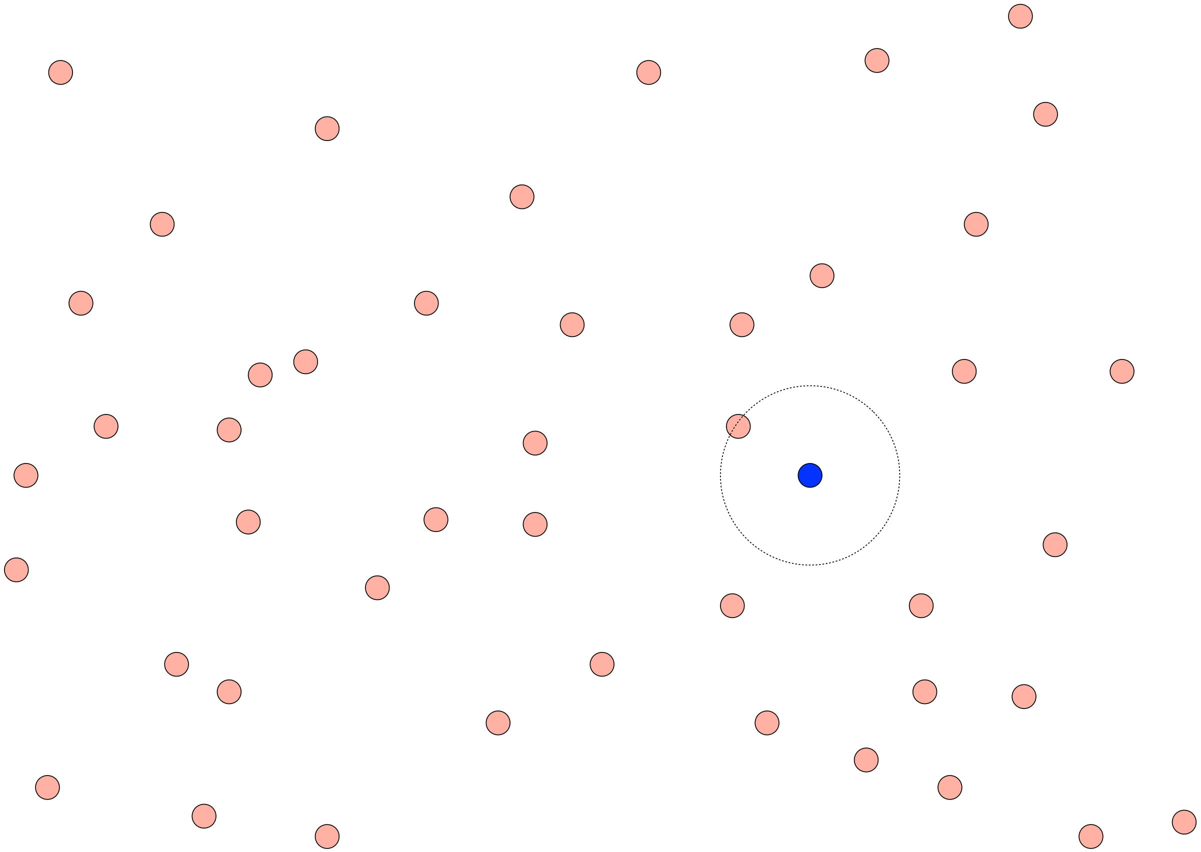
$\lceil \log\log(u) \rceil = 5$

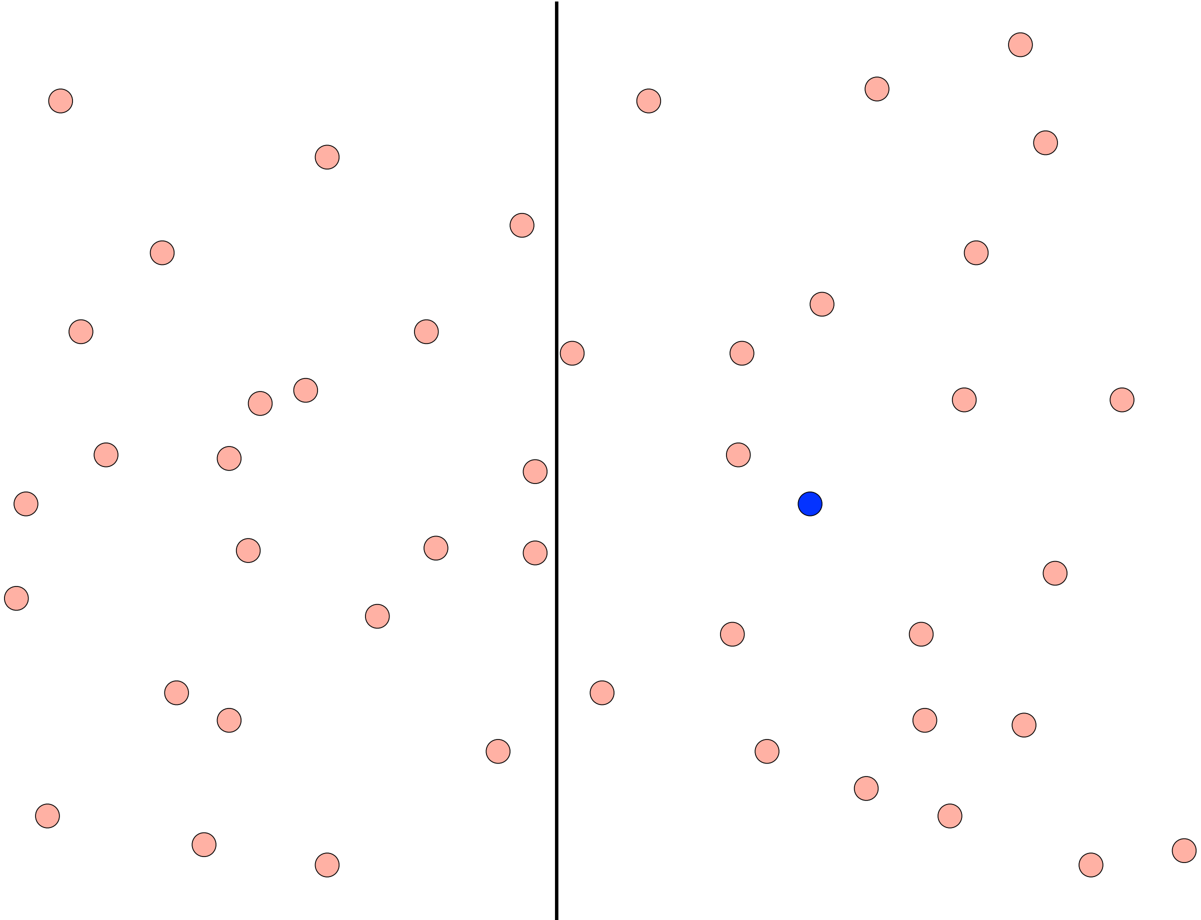
We need to fix the Lookup to work with Lazy inserts.

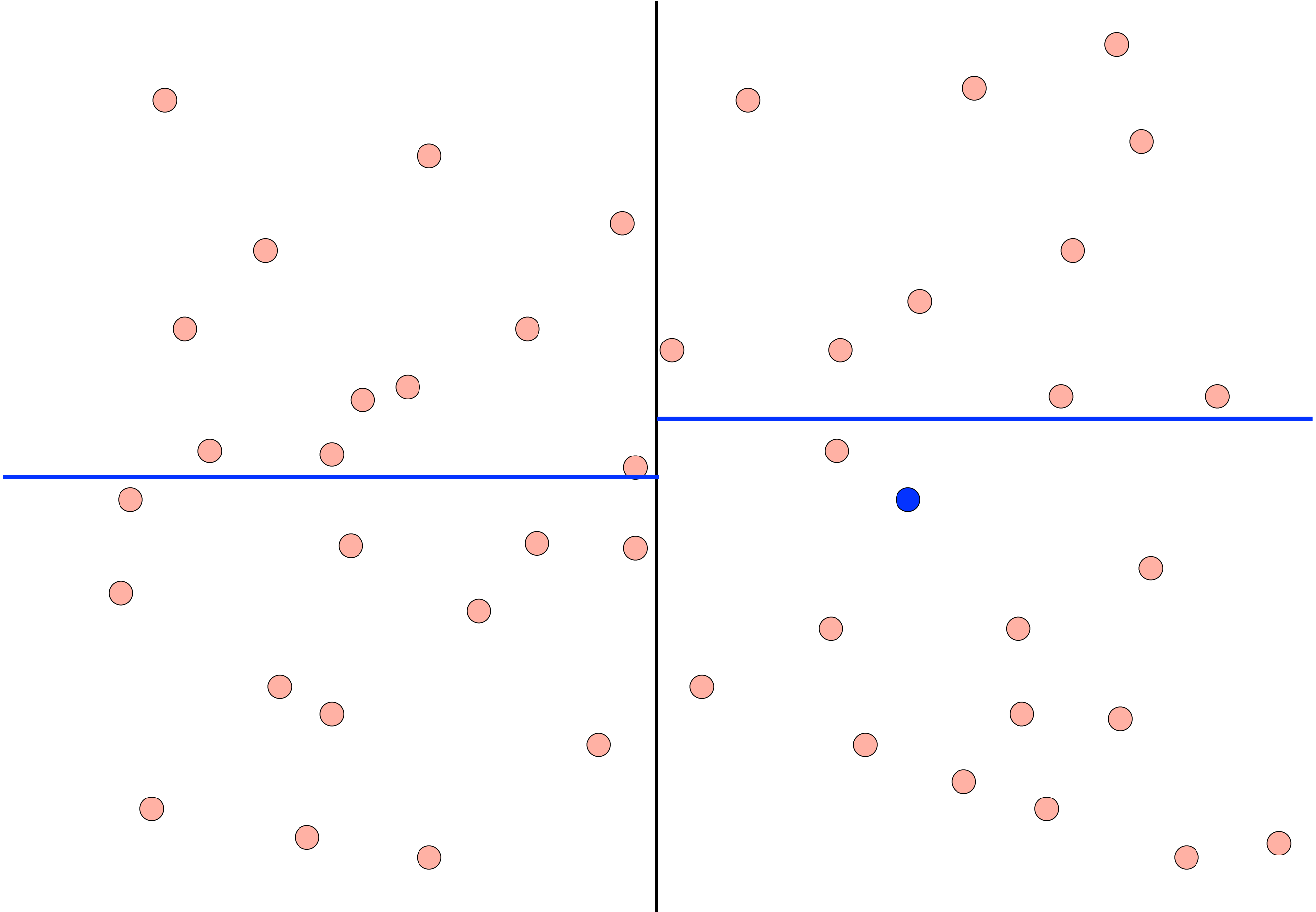
Nearest neighbor queries

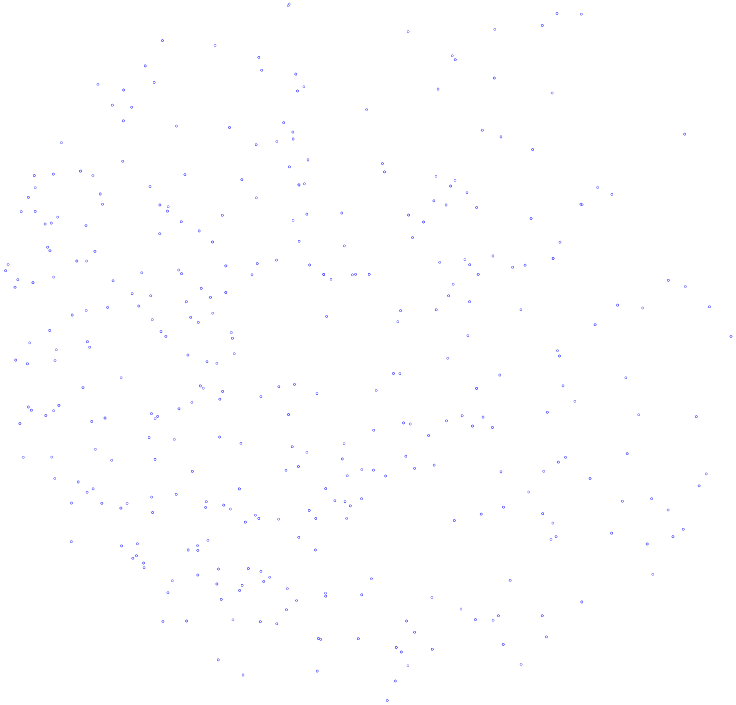


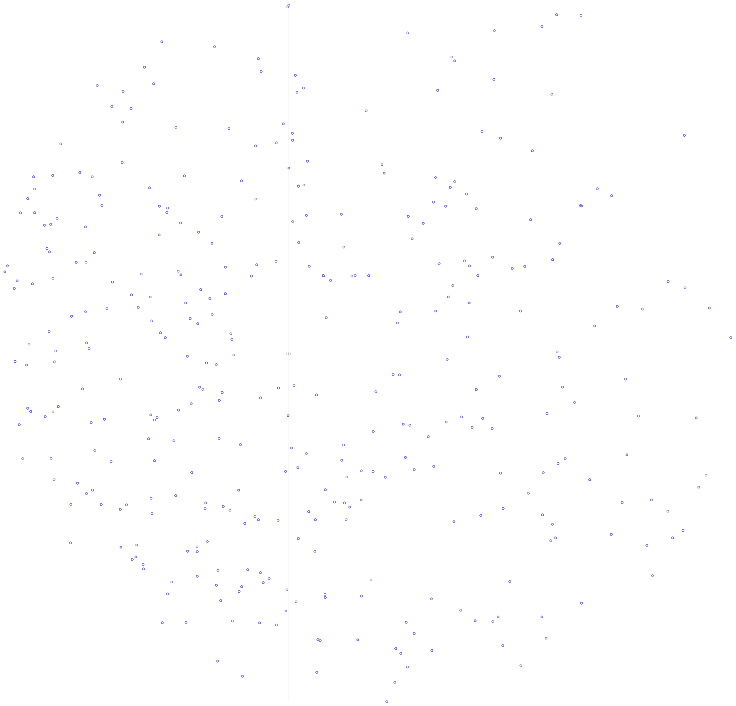


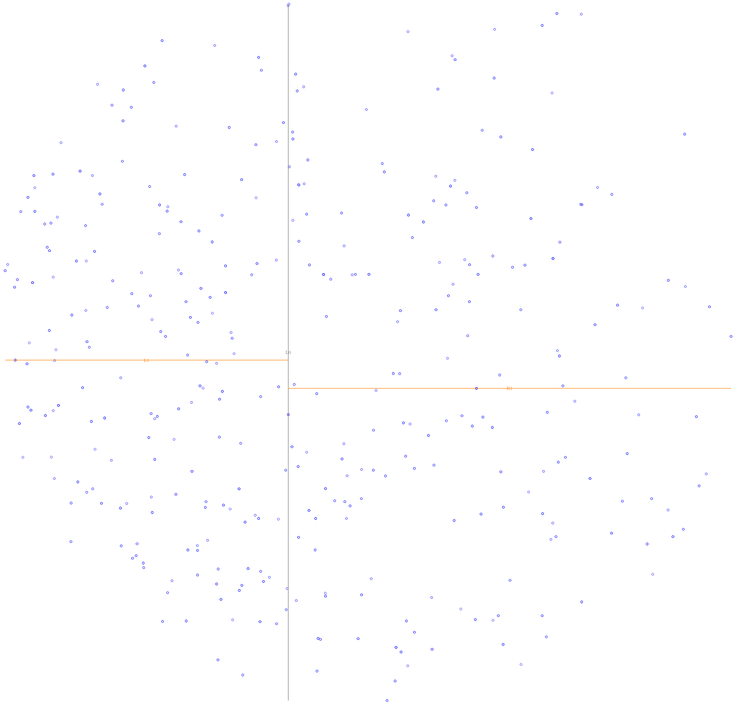


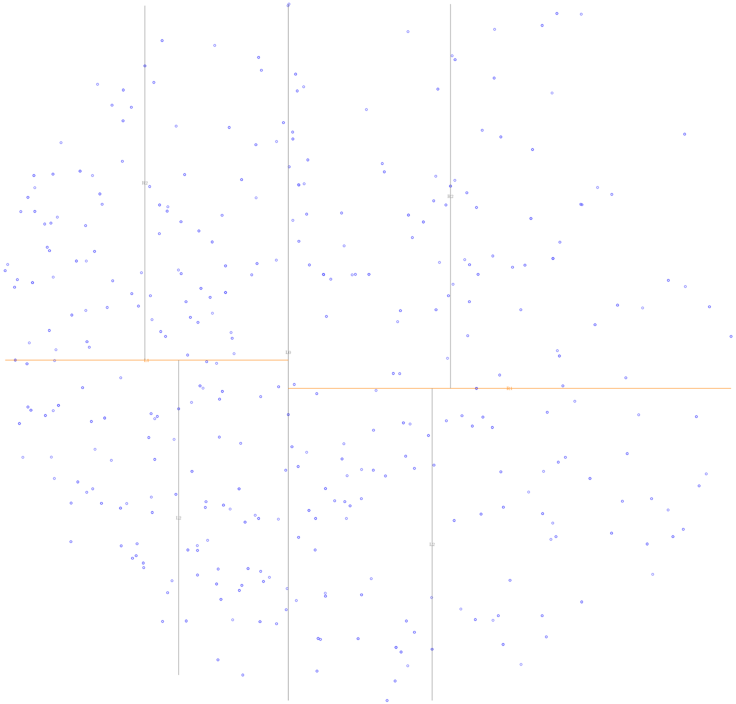


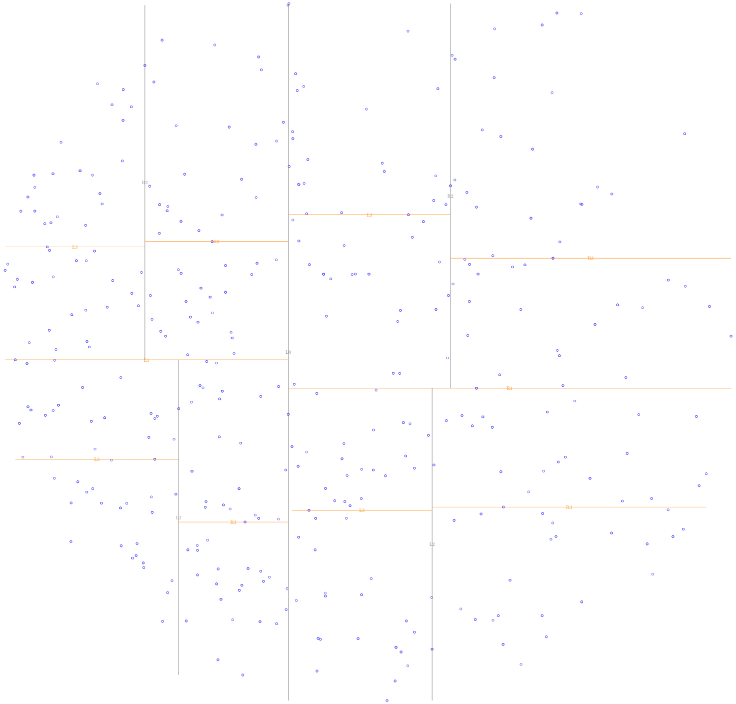


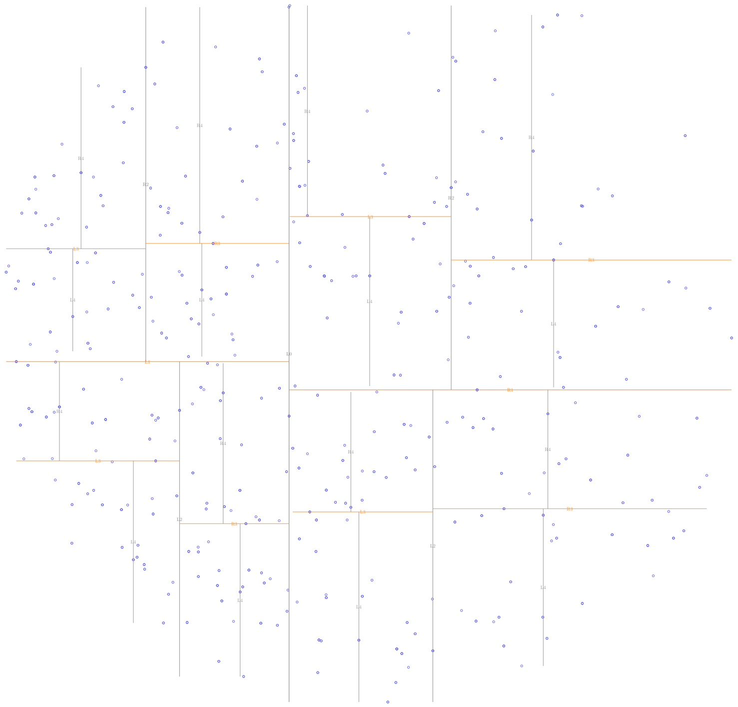





L0
L1
R1


R2
L0
L1
L2
R2
R1
L2

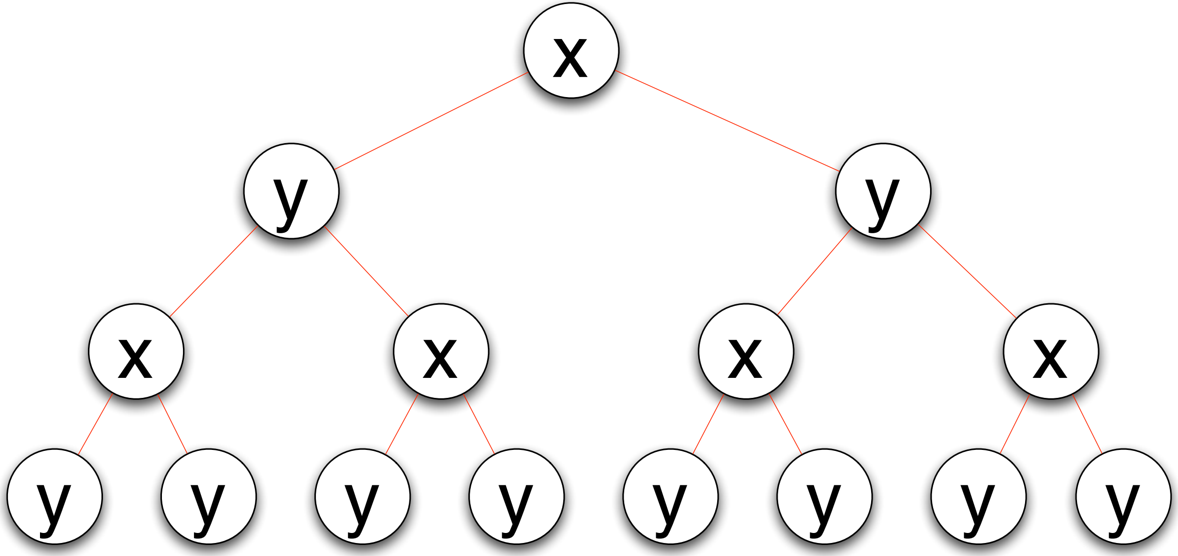

R2
R2
L3
R3
L3
R3
L0
L1
R1
L3
L2
L3
R3
R3
L2

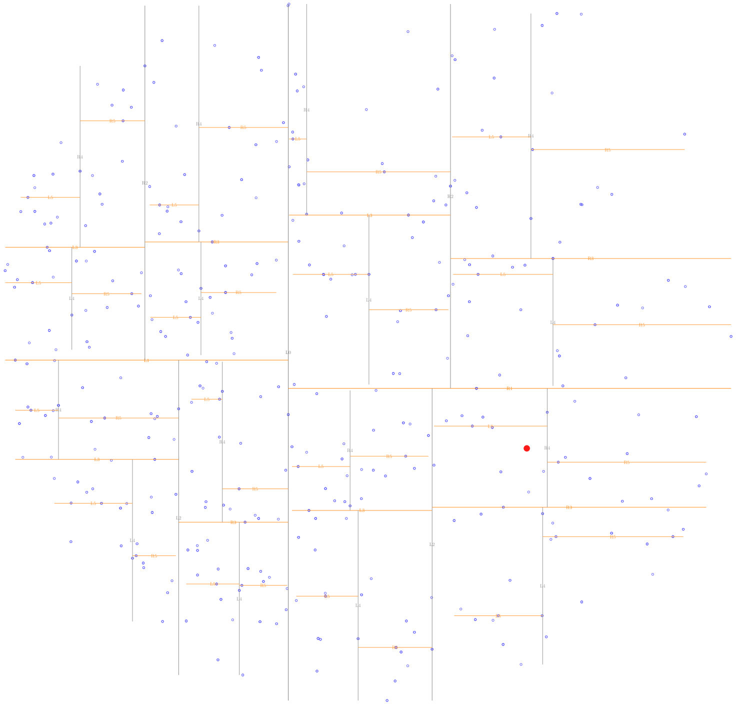


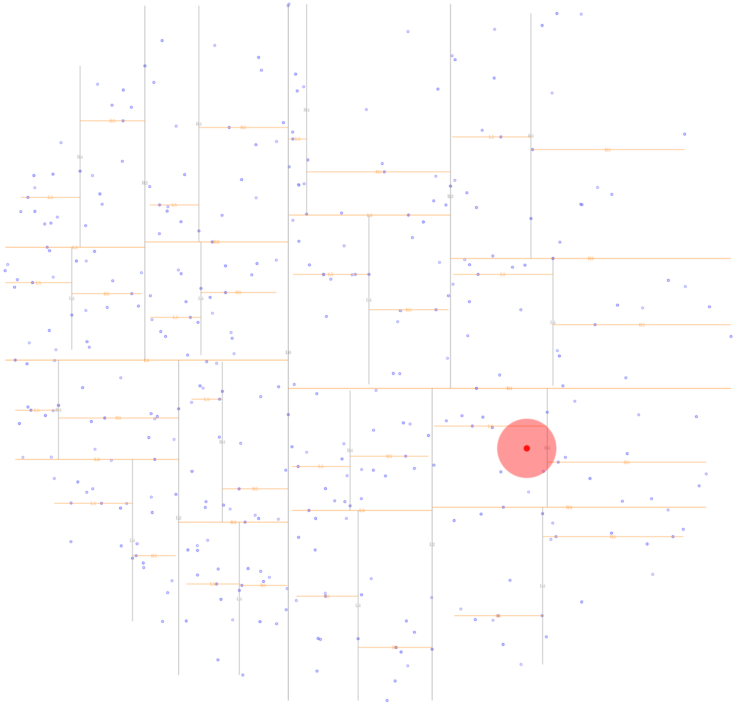
KD-Tree

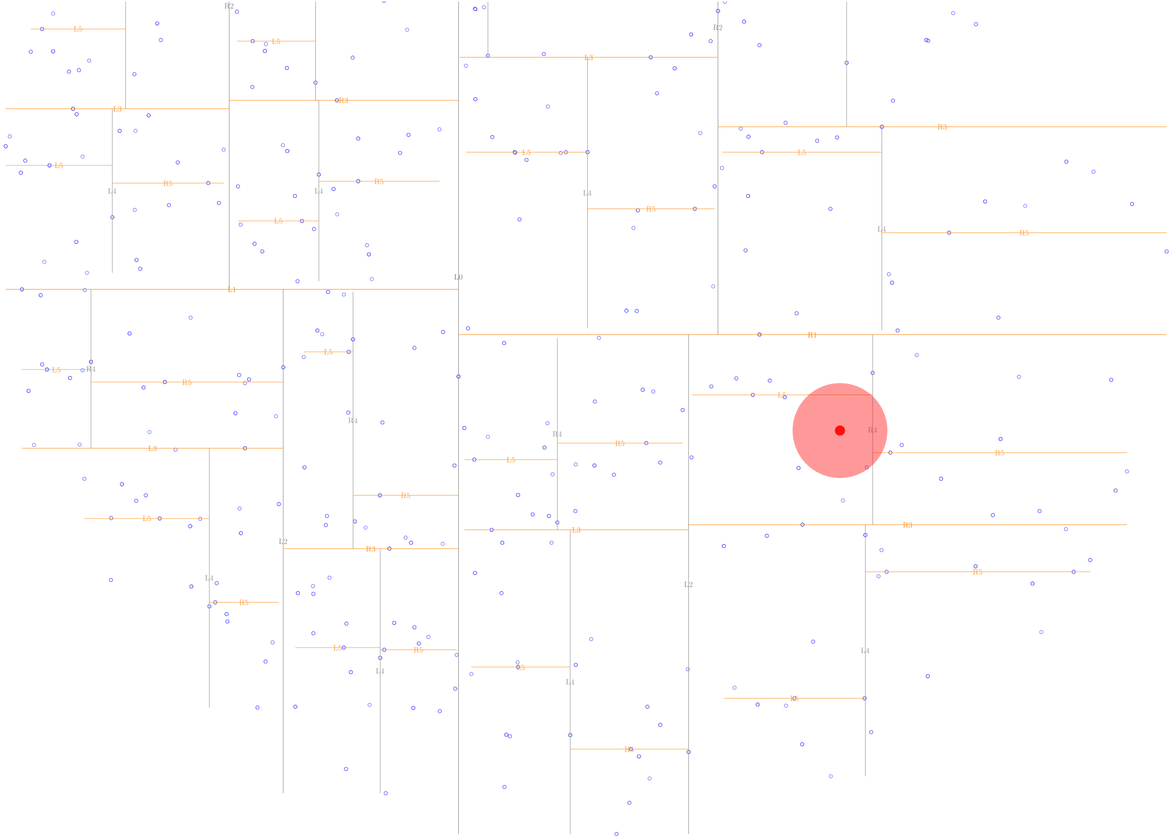
Each node in tree maintains variable “box”

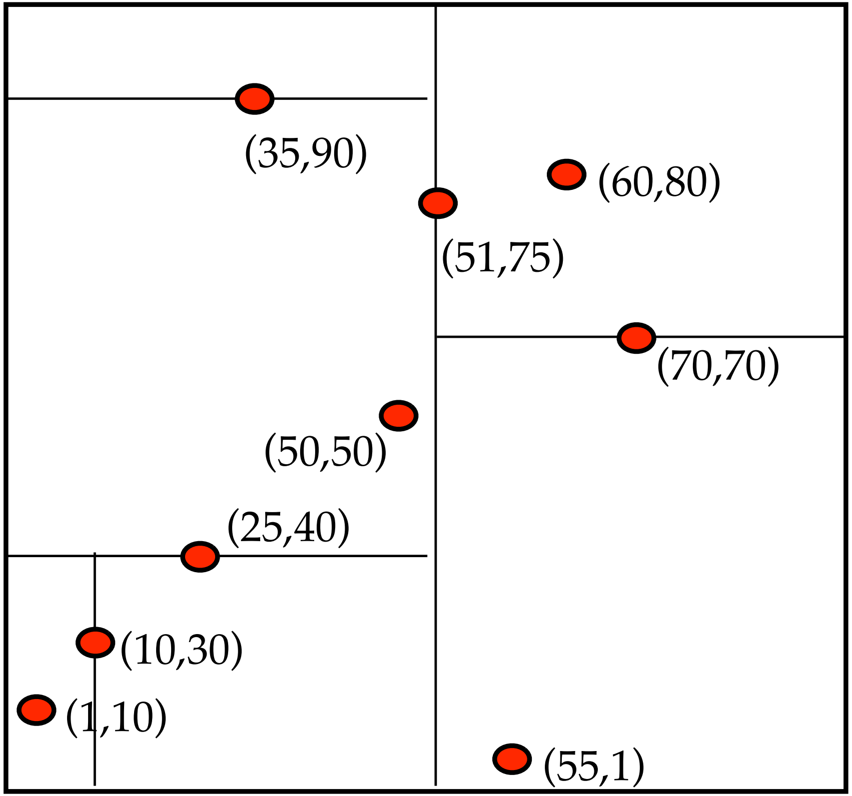
```
node {
 rect box
 point split
 node* left
 node* right
}
```


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)
    }
```