5800
data structures

apr8/apr11 2022
shelat

Dictionary

data structure

insert(key, value) : add an item to the data structure

lookup(key) : returns (key, value) if it was previously inserted into the data structure

Delete(key) :

FINDNEXT(key) : finds the (key', value) entry where key' is the smallest value inserted for which key' > key

MIN, MAX

Dictionary

insert(key, value)

delete(key)

lookup(key)

findnext(key)

Dictionary

standard solution: hashtable

insert(key, value)

delete(key)

lookup(key)

findnext(key)

We expect O(1) performance. But usually this only happens in expectation. worst case performance could be O(n).

→ might require scanning all keys. O(n)

Hashtables are tricky

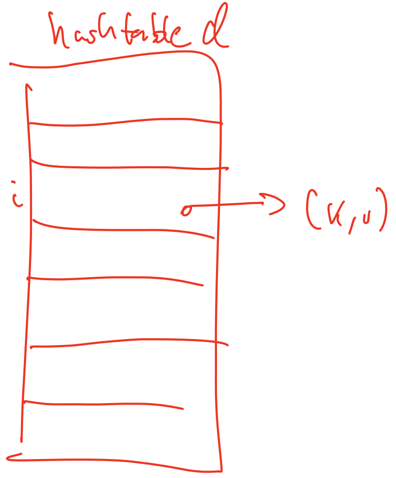
```python

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

```

d[k] = v

↳ i = hash(k).

if i is free
add (k,v)
at i

else
: keep looking for
a new spot
"open addressing"



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:

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

$U \sim 2^{32}$

$\log \log U = 5 !!$

insert(key, value)

delete(key)

lookup(key)

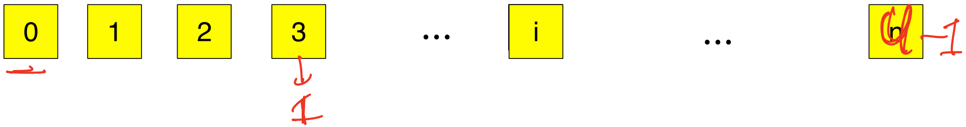
findnext(key)

$\Theta(\log \log U)$ performance.

Note: $O(\log n)$ or $O(\log U)$ performance can be achieved with balanced binary trees.

A simple solution: bit vector

Maintain an array of bits



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

$\Theta(1)$

$\Theta(n)$ in worst case

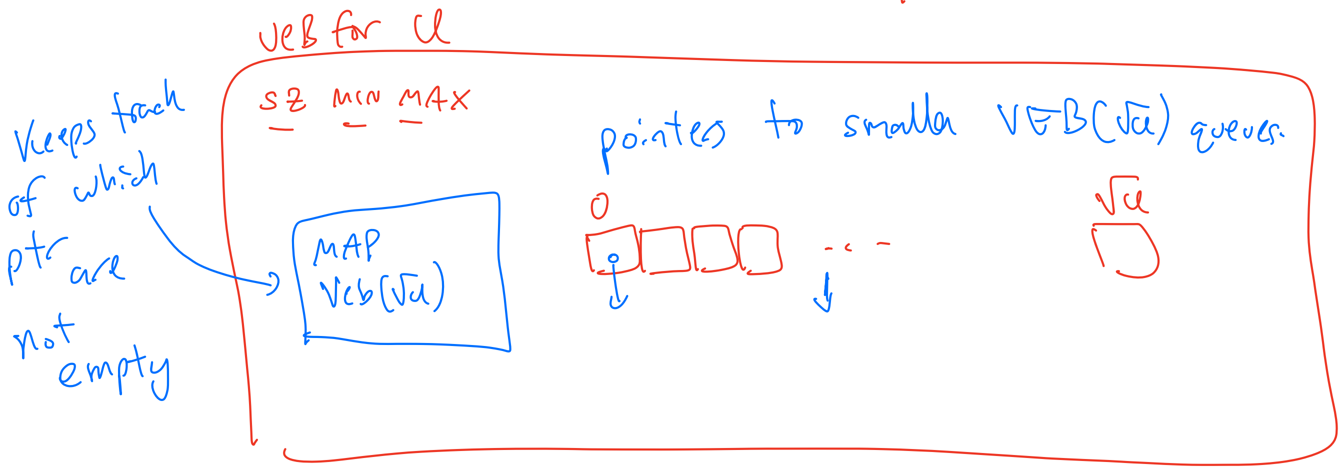
we can still use this DS as a base case for small U.

Can we do better than O(n) findnext?

van emde Boas Q

vEB Q

THE BIG IDEA: a datastructure for universe 1...U that is comprised of small versions of the same datastructure for universe 1...√U.



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

$\mathrm{VEB}_{(N)}$

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: 4 BIT VECTOR. for $u \le 4$, bit vector.

NORMAL CASE:

MAP

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

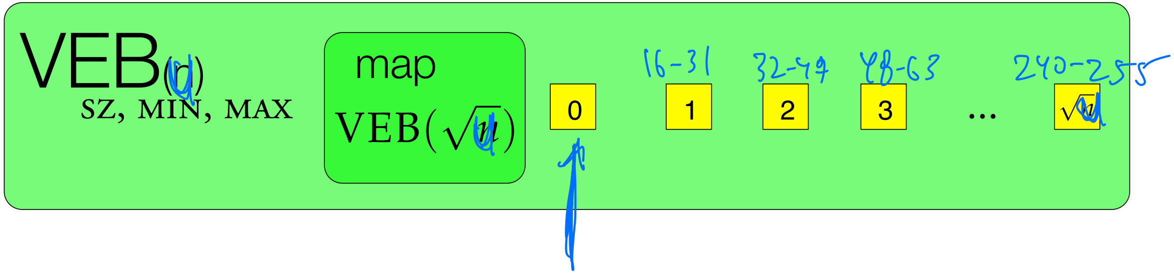
Pointers to recursive, smaller instances of VEB.

0 1 2 3 … $\sqrt{n}$

Keeps track of which ptrs are not empty.

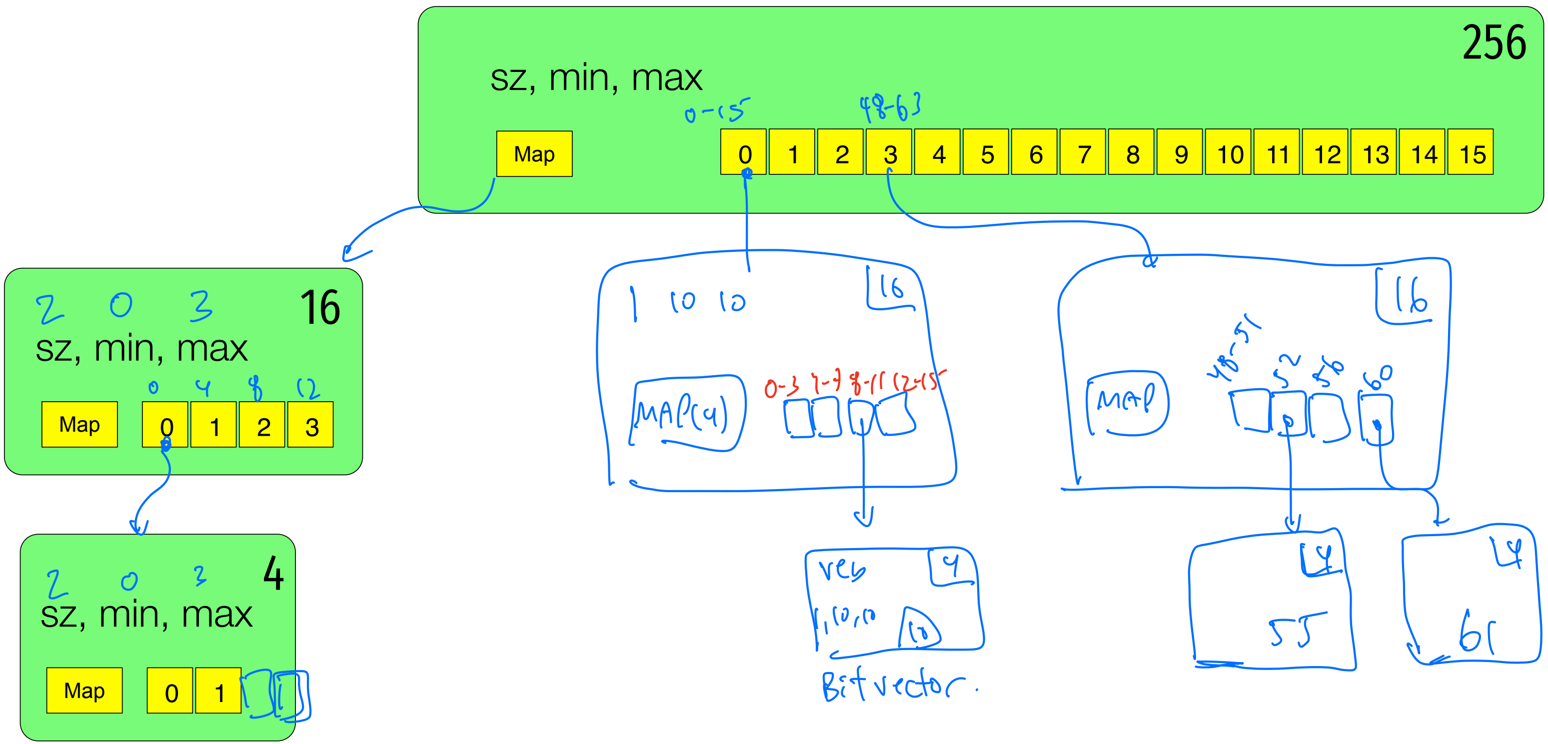
EXAMPLE:

$u = 256$

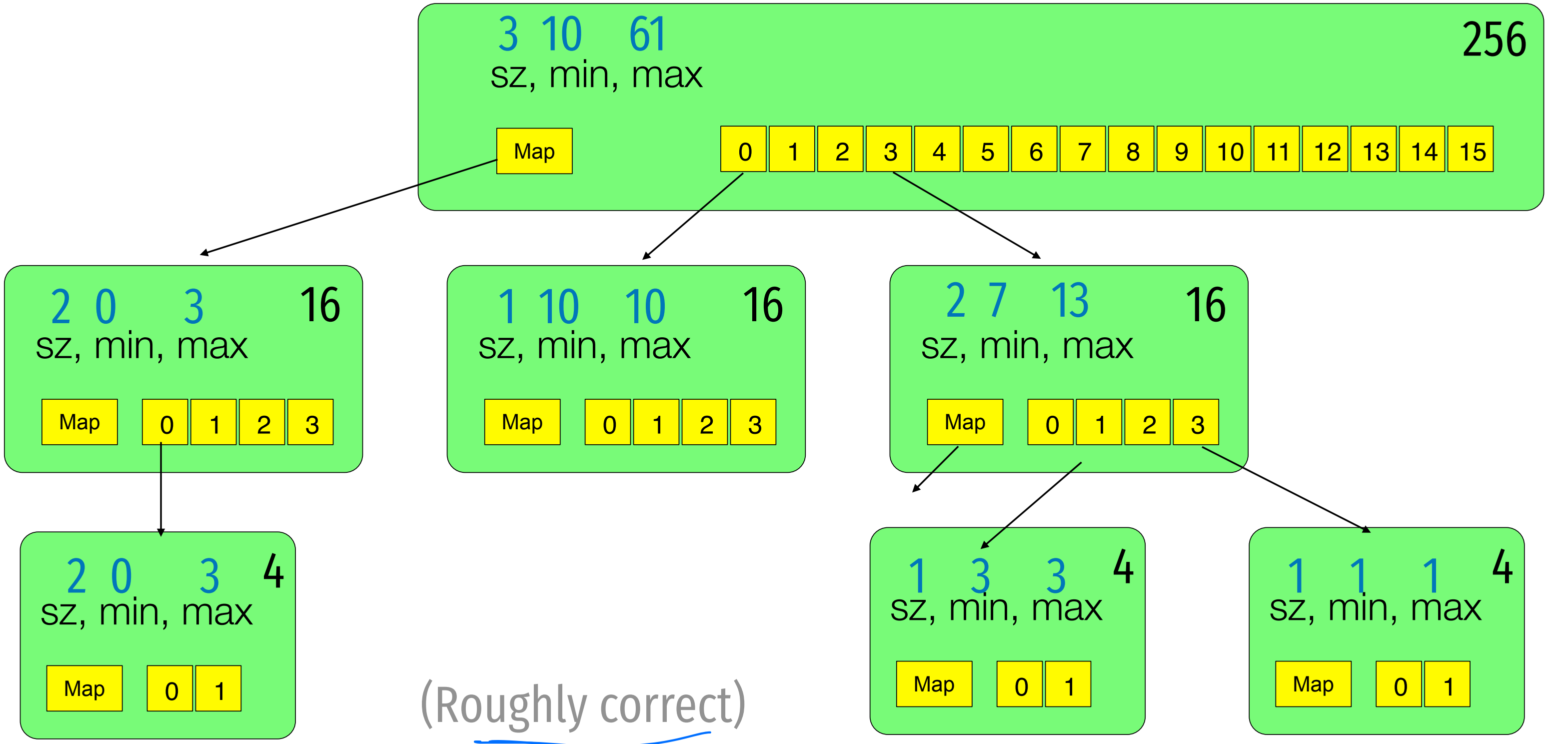


this pointer would point to a VEB queue that would store any entries between 0...15

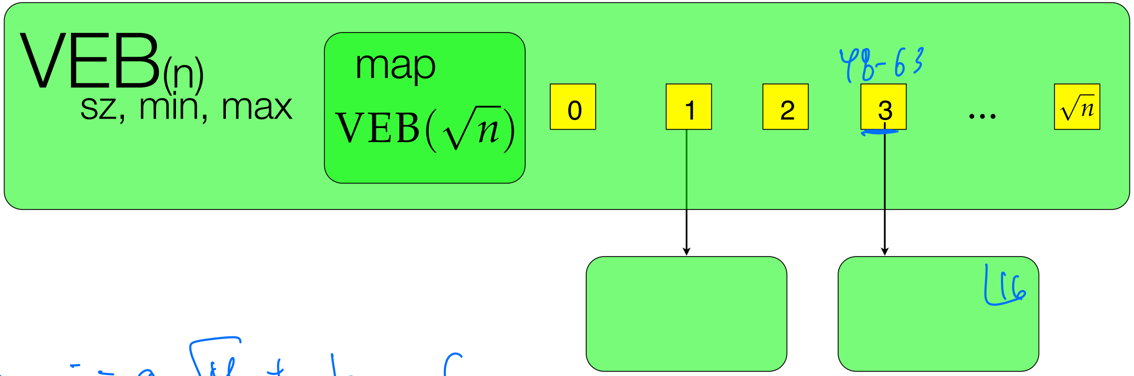
Example n=256, keys={10,55,61}



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



(Roughly correct)



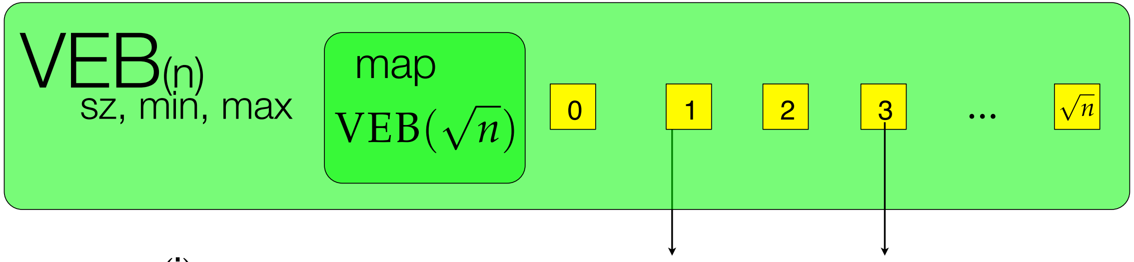
LOOKUP(i)

O(1) {
- write $i = a \cdot \sqrt{u} + b$ for $a, b \in [0 \ldots \sqrt{u}]$
- <base case> → for $u \leq 4$, bit vector
- If a == null return false

eg $55 = 3 \cdot \sqrt{256} + 7$

else return a.lookup(b)

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



LOOKUP(i)

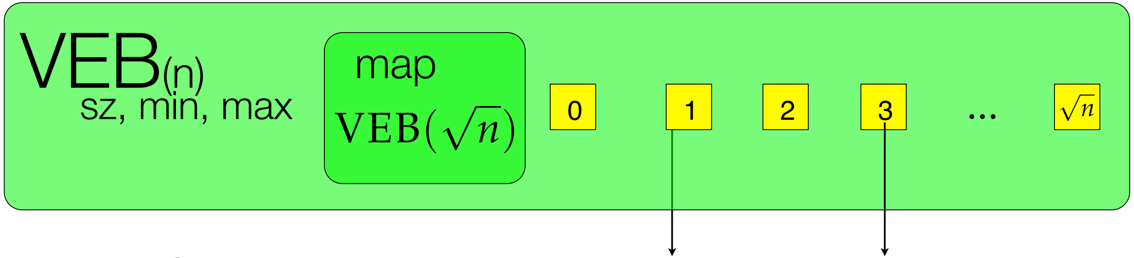
WRITE $i = a\sqrt{n} + b$

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

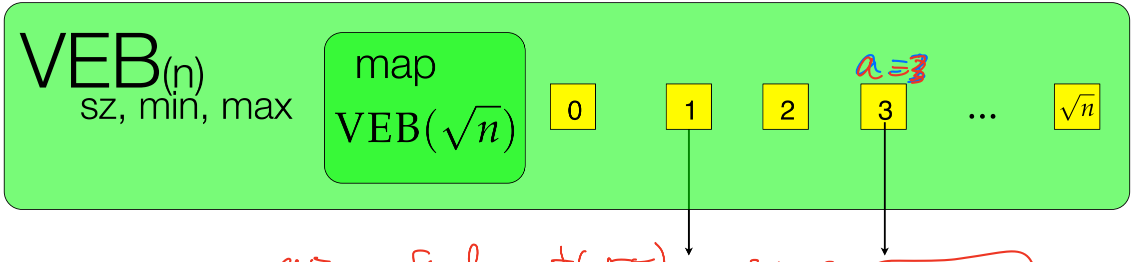
IF <BASE CASE>: CHECK BIT VECTOR

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)

example findnext(55) $= 3 \cdot 16 + 7$

max 55

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

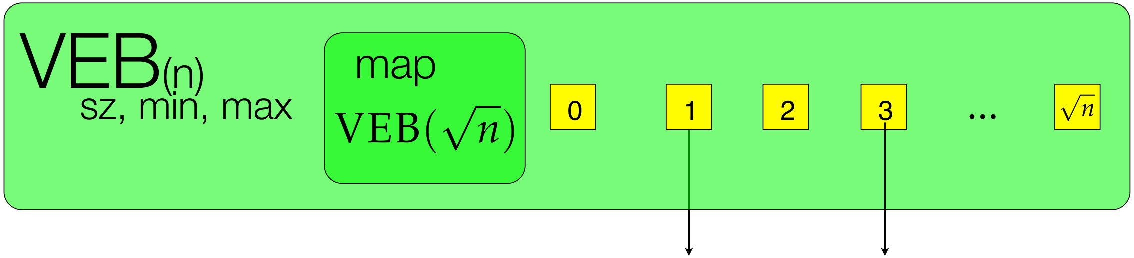
Case 1: if the next element occurs in bucket a

easy: we can use a.findnext(b)

use a.max to pick

Case 2: the next element occurs in the next non-empty bucket

use map.findnext(a).min



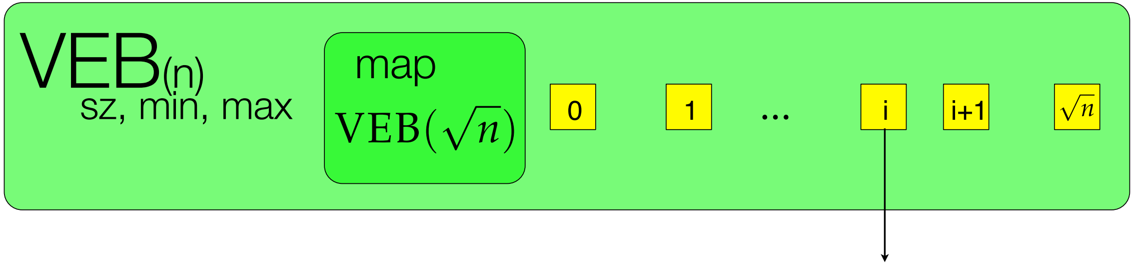
FINDNEXT(i)

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 $i = a \cdot \sqrt{u} + b$

[base case, use the bitvector if $u \le 4$]

If $a.max > b$ then
}

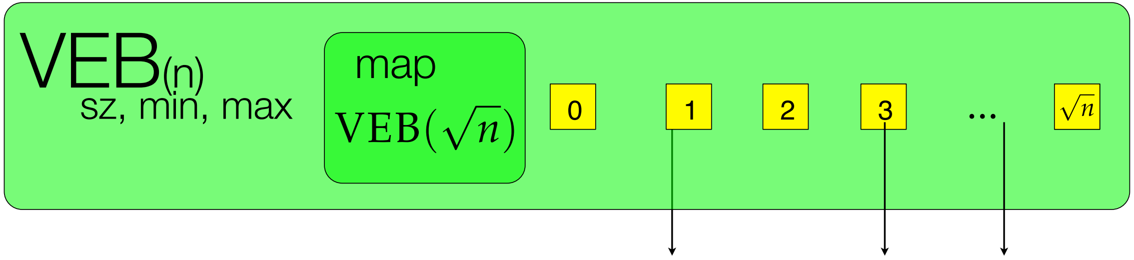
return a.findnext(b) ← only do one

else

return map.findnext(a).min ←

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

$$= \Theta(\log \log u)$$



FINDNEXT(i)

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

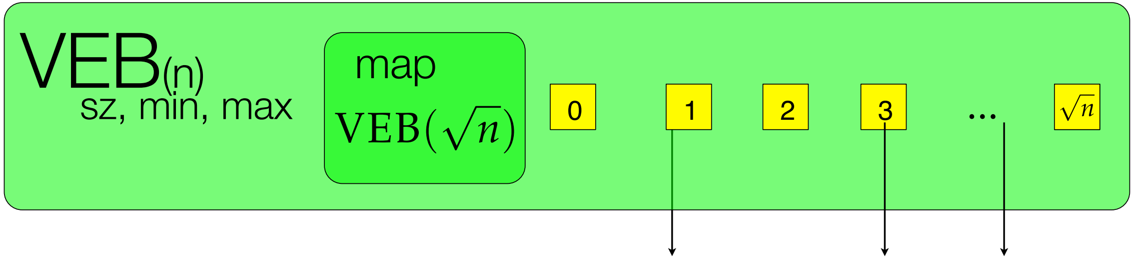
<BASE CASE IF SIZE IS ZERO>

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

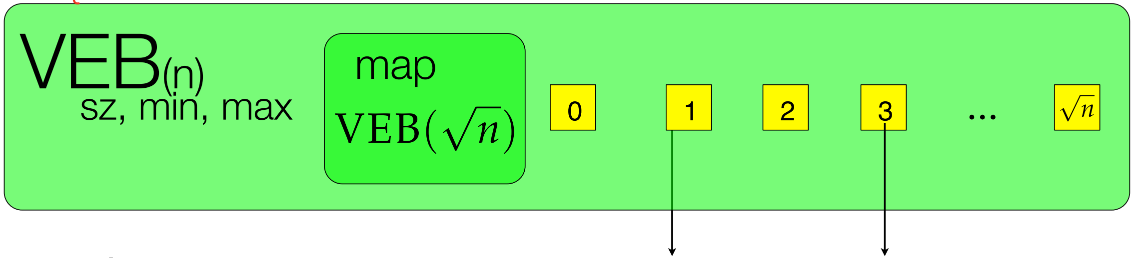
RETURN MAP .FINDNEXT(a) .MIN

Running time:

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

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

First attempt



INSERT(i)

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

a.insert(b)

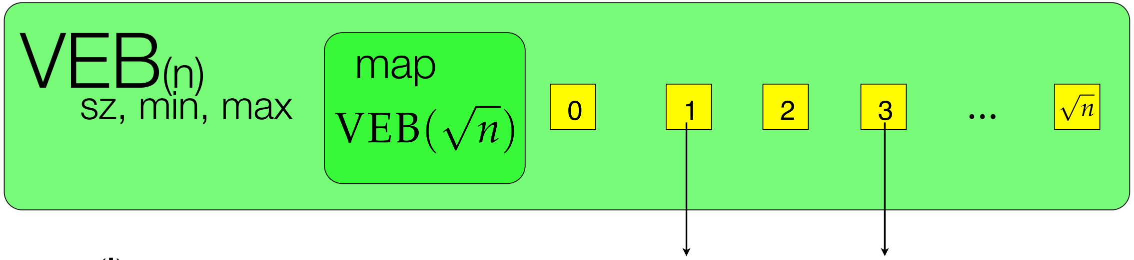
map.insert(a)

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

$$= O(\log u)$$

$$S(n) = 2S(\tfrac{n}{2}) + \Theta(1)$$

$$O(n)$$

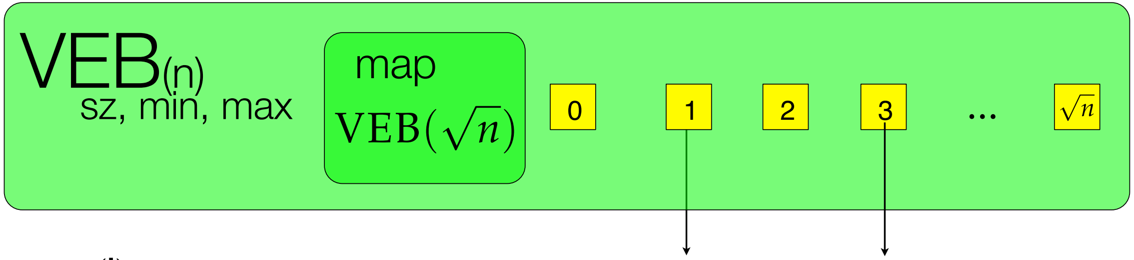


INSERT(i)

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

A.INSERT(B)

MAP.INSERT(A)



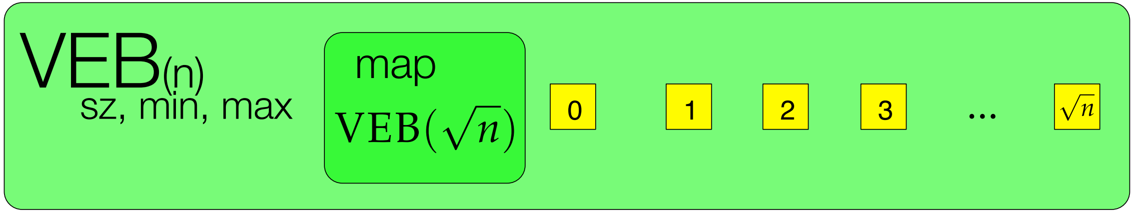
INSERT(i)

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

A.INSERT(B)

MAP.INSERT(A)

WHAT IS THE PROBLEM WITH THIS?



insert(i)

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

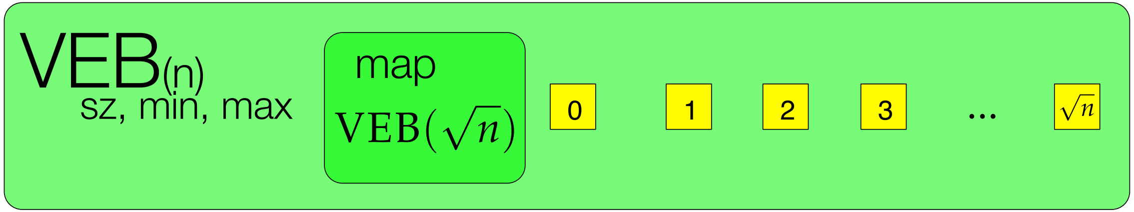
a.insert(b)

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

se case w/ BIT vector)

IF SZ==O THEN → SZ=1 MIN = MAX = i

$\Theta(1)$ for insert into empty datastructure.

ELSE

IF MIN > i SWAP(i, MIN)

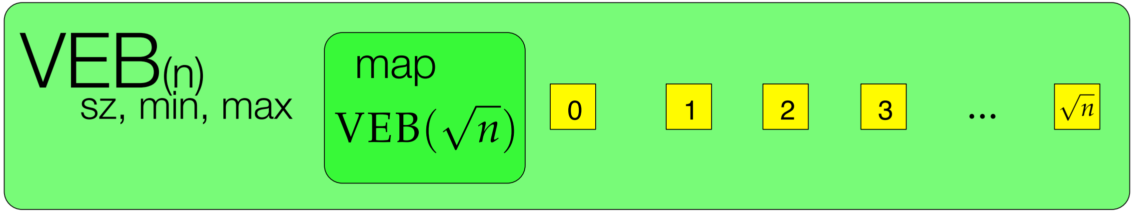
write $i = a \cdot \sqrt{u} + b$

IF a.size == 0 {map.insert(a)}

a.insert(b)

update SZ AND MAX

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



INSERT(i)

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

ELSE

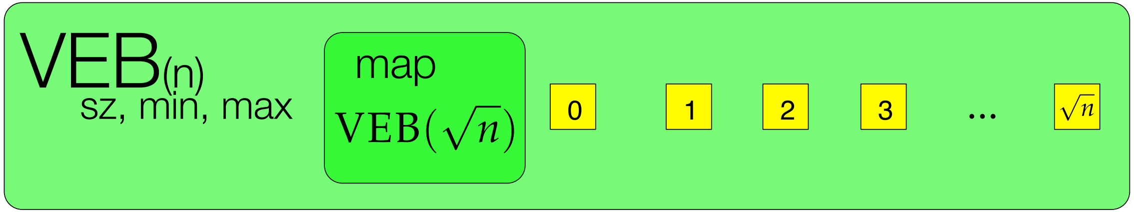
IF MIN>i SWAP(i,MIN)

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

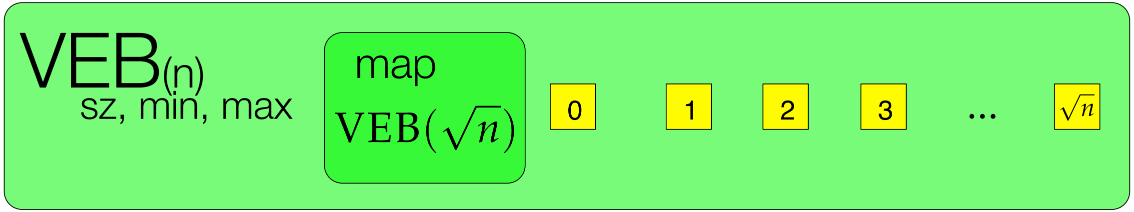
IF a .SZ==0 THEN MAP .INSERT(a).

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

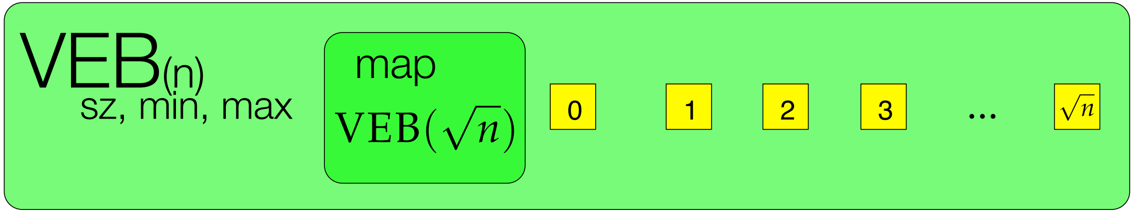
IF a .SZ==0 THEN MAP .INSERT(a).

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).

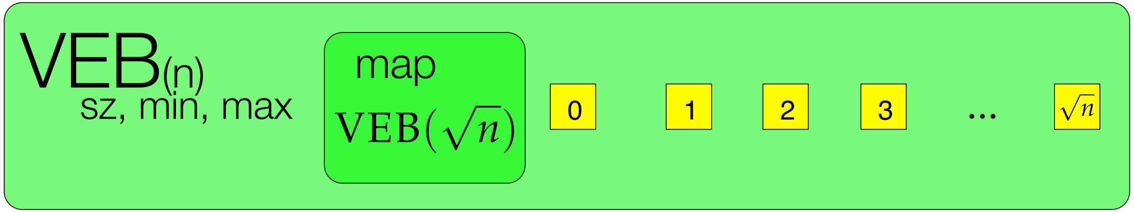
a .INSERT(b)

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

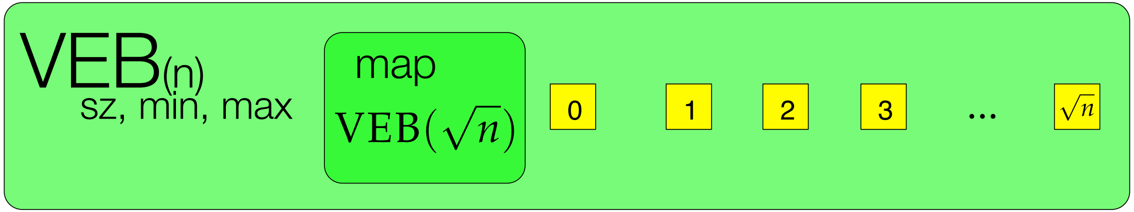
$$T(n) = T(\sqrt{n}) + \Theta(1)$$



LOOKUP(i)

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

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



LOOKUP(i)

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

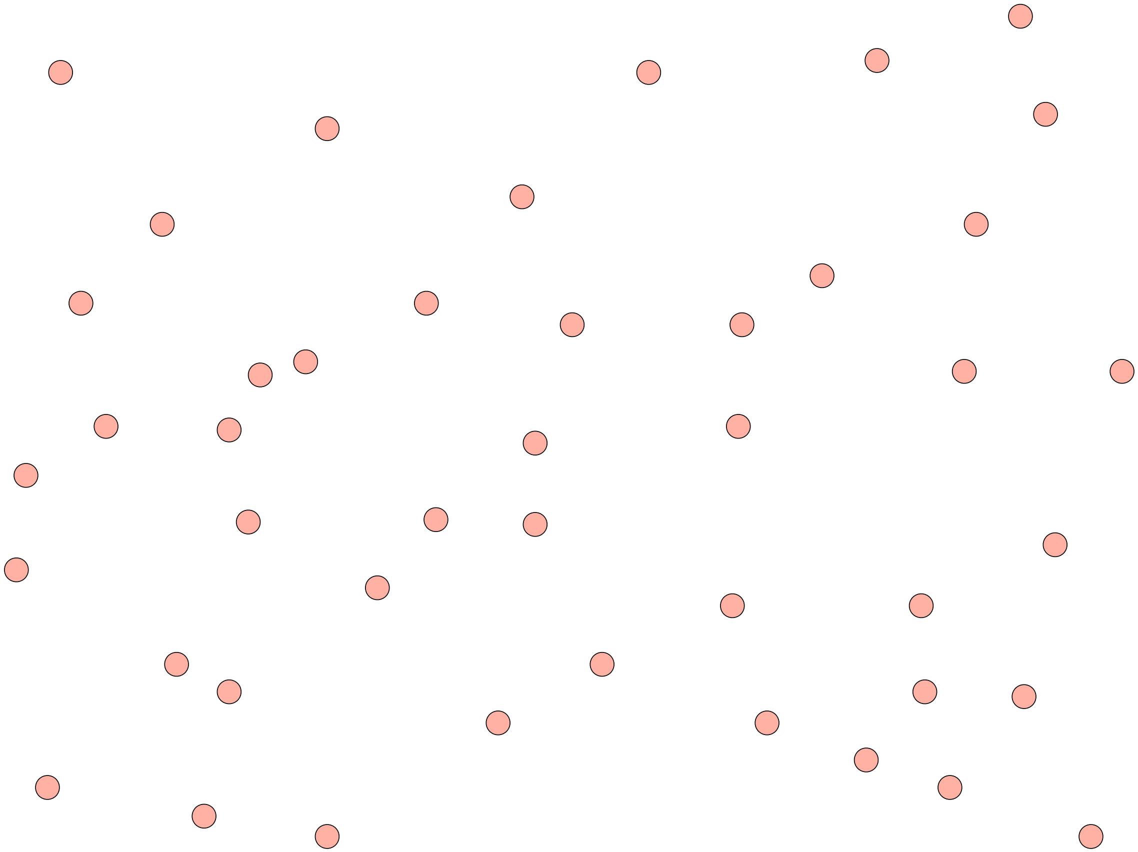
IF SIZE==0 RETURN FALSE

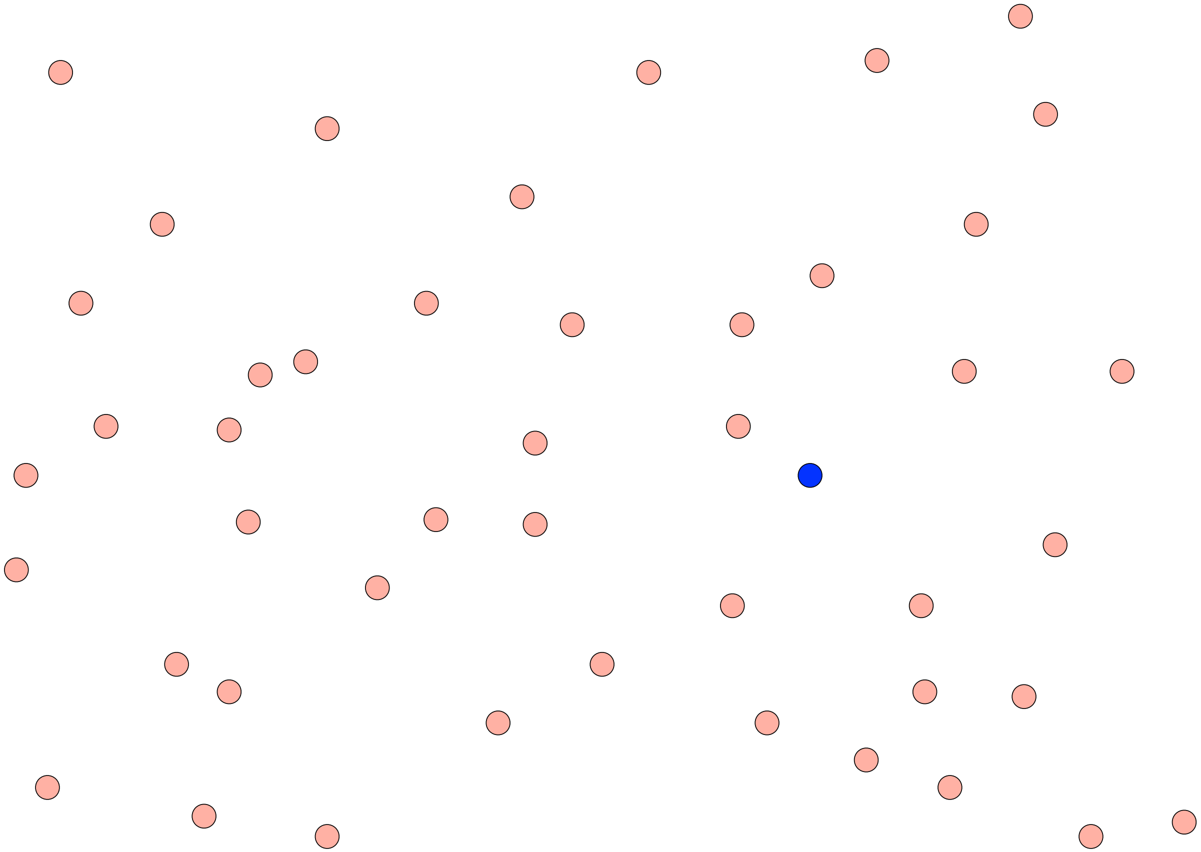
IF i==MIN RETURN TRUE

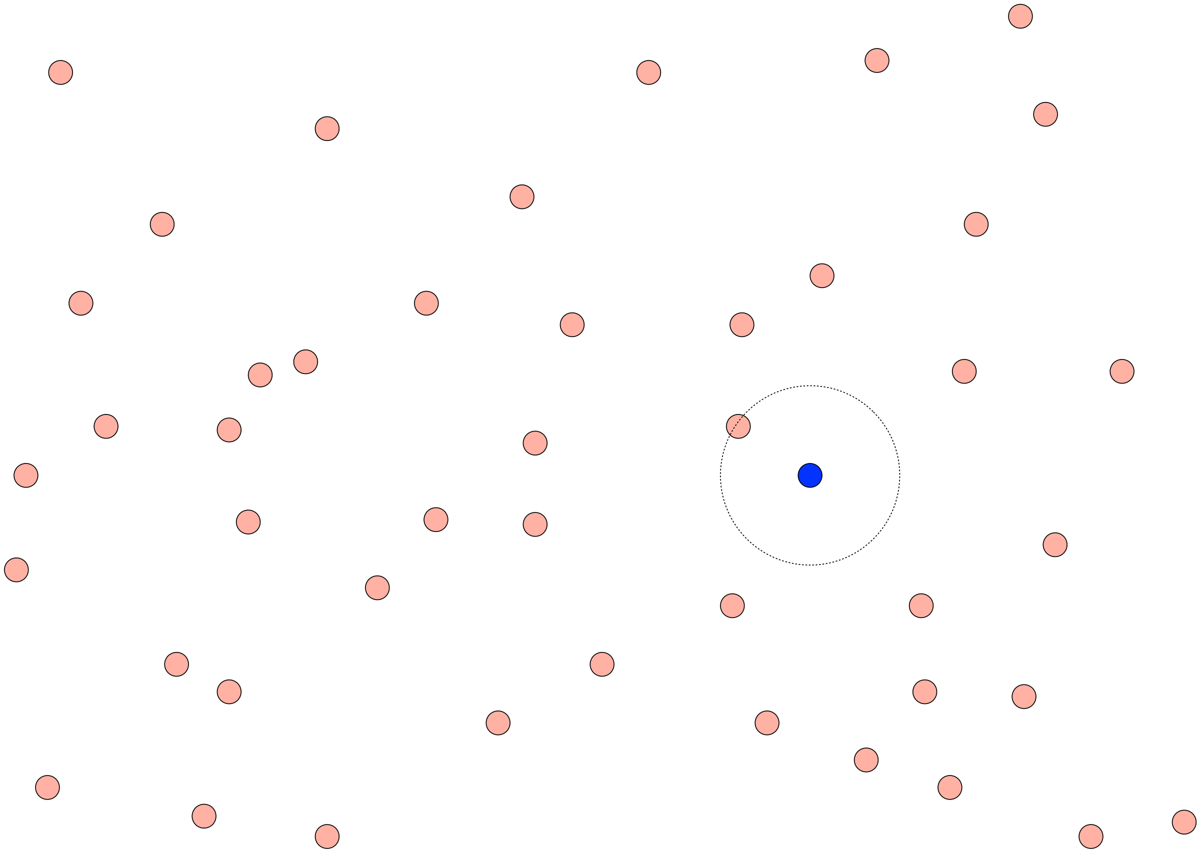
ELSE RETURN a.LOOKUP(b)

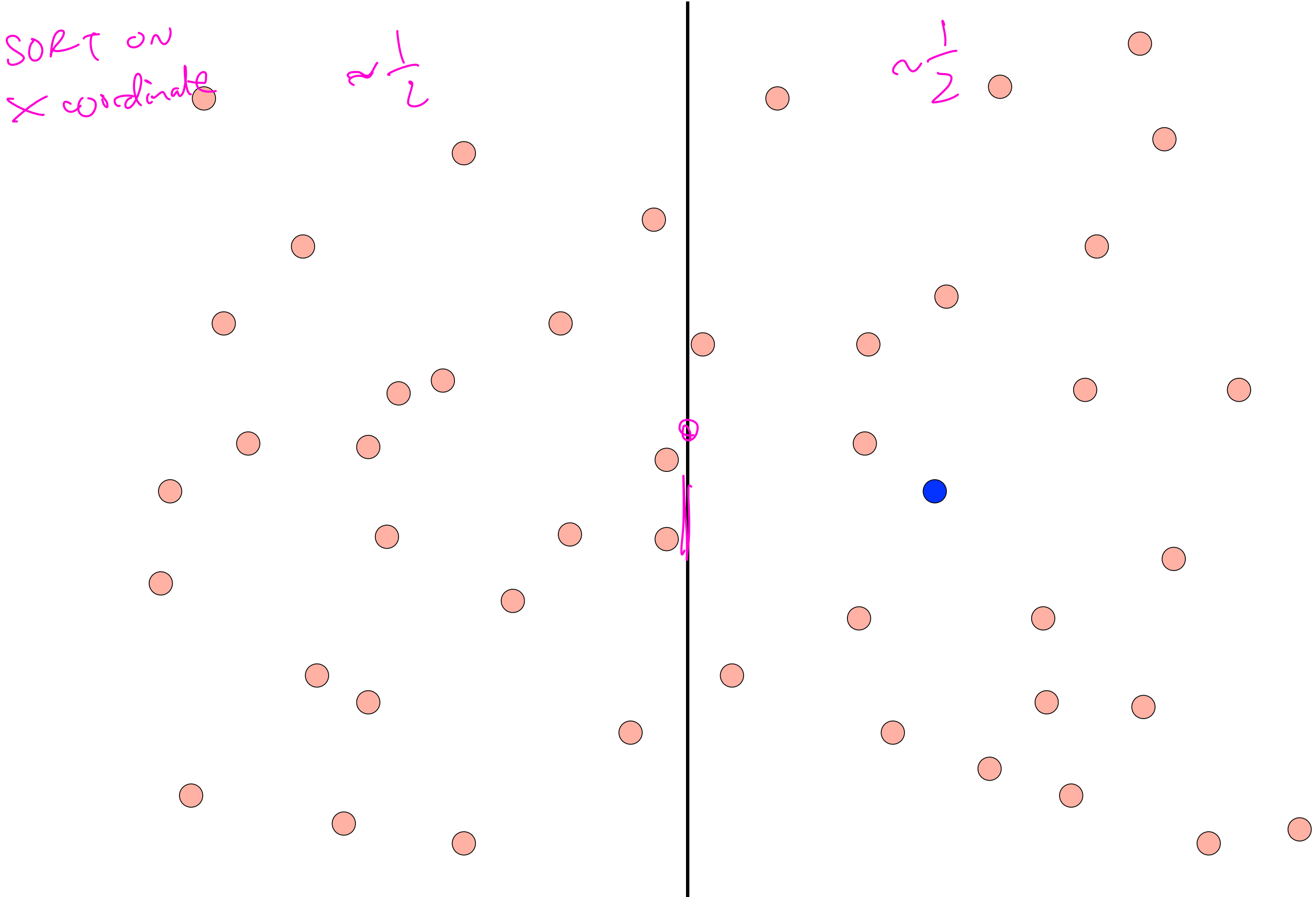
We need to fix the Lookup to work with Lazy inserts.

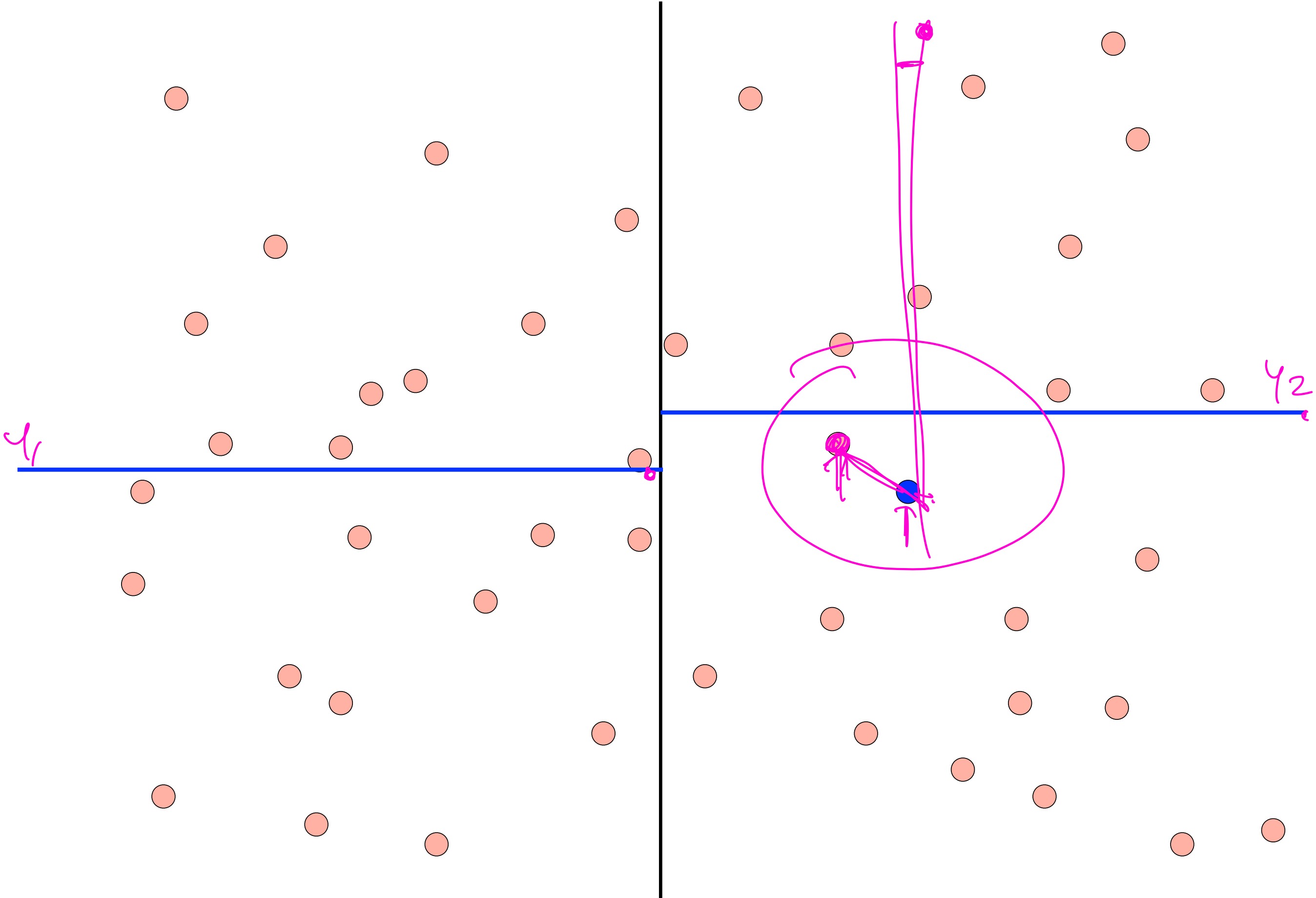
Nearest neighbor queries



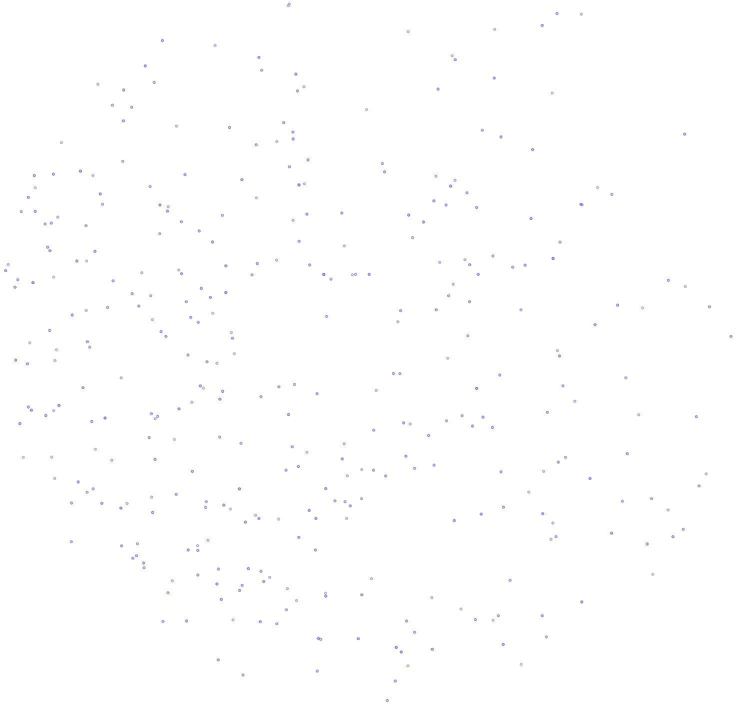


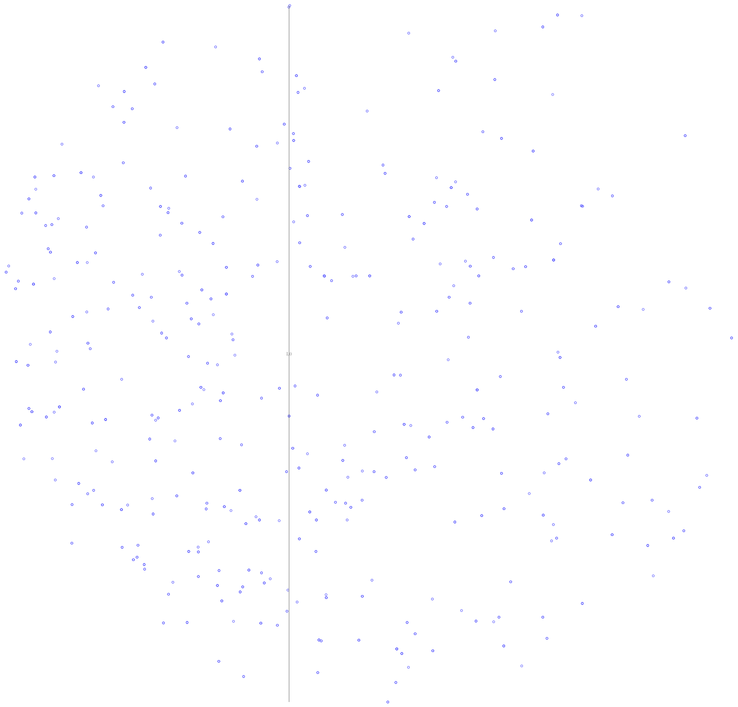



SORT ON
x coordinate
~1/2
~1/2

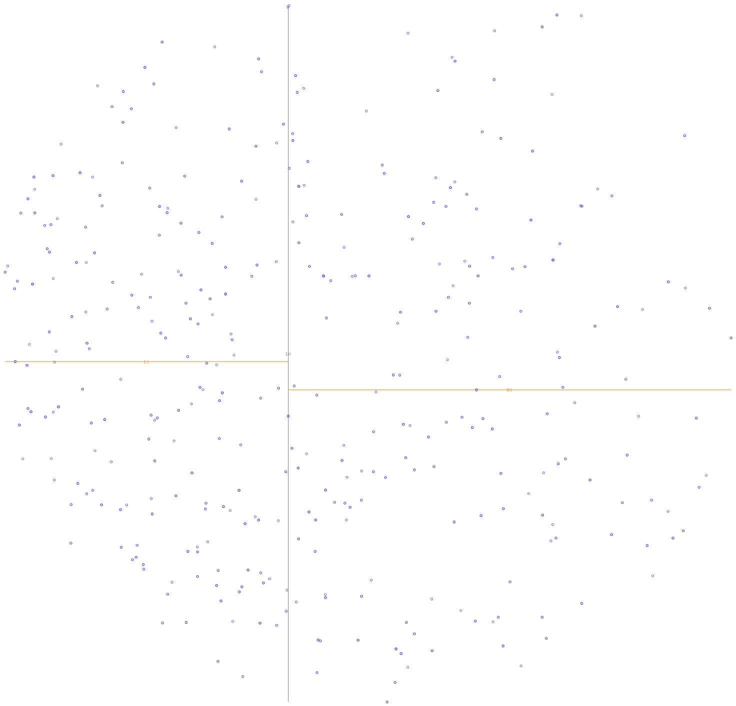


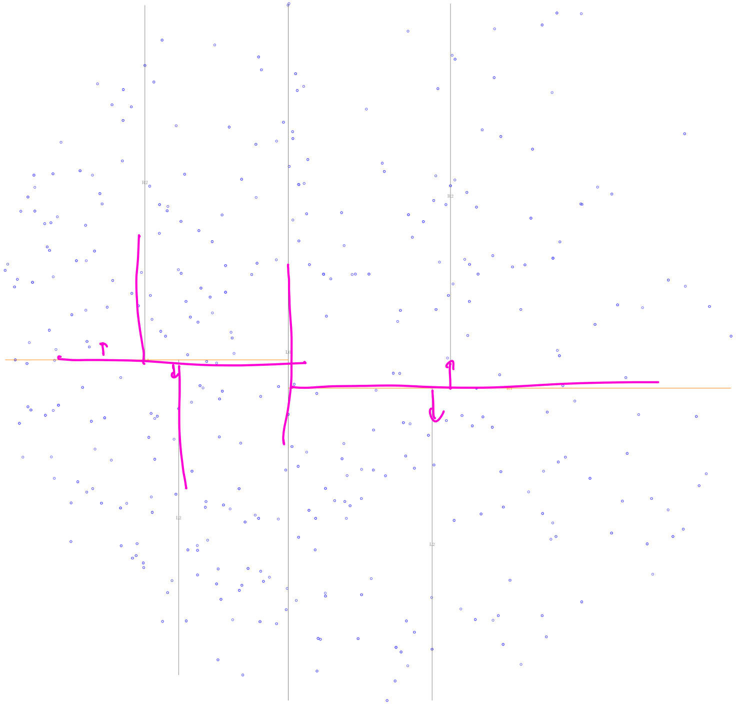
y1
y2

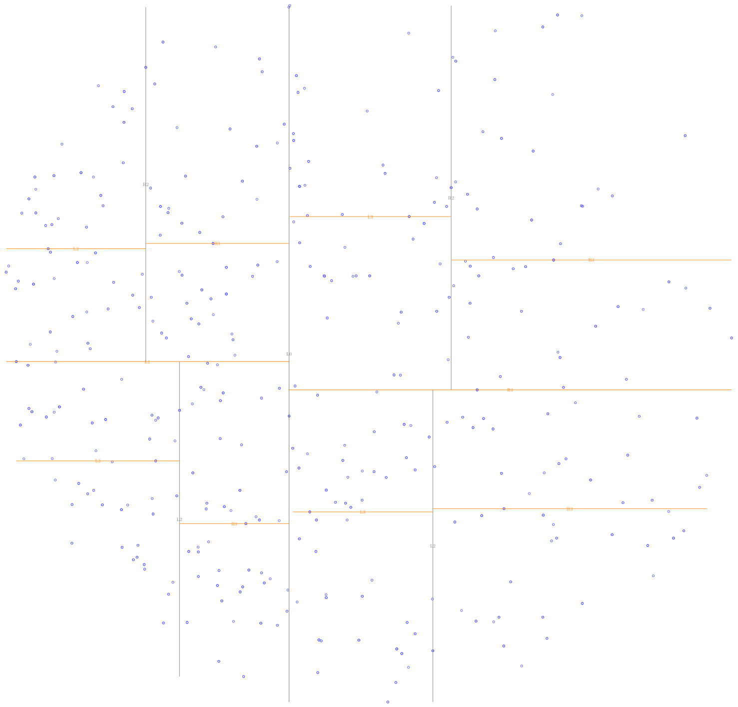


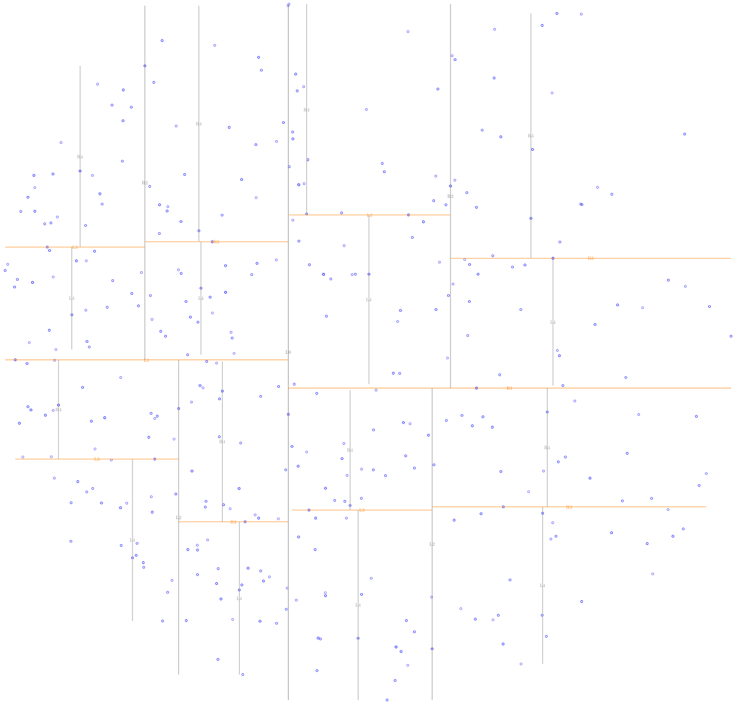


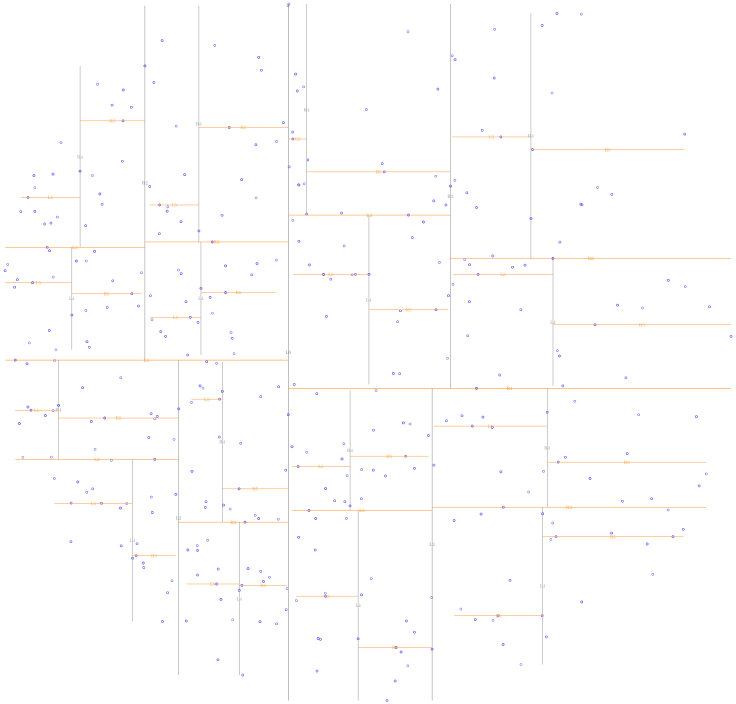
1.0


L0
L1
R1




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

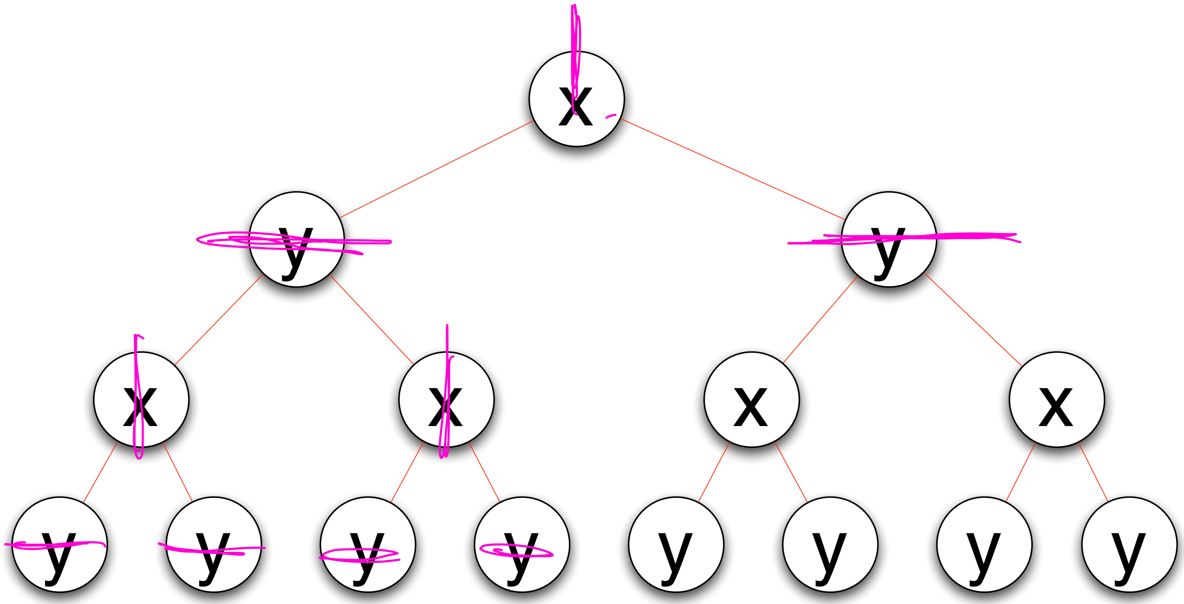


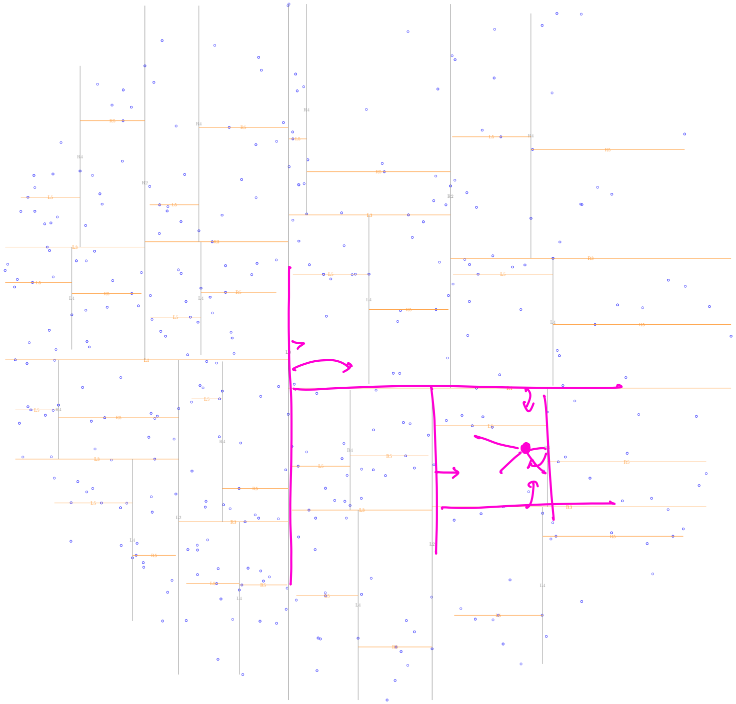


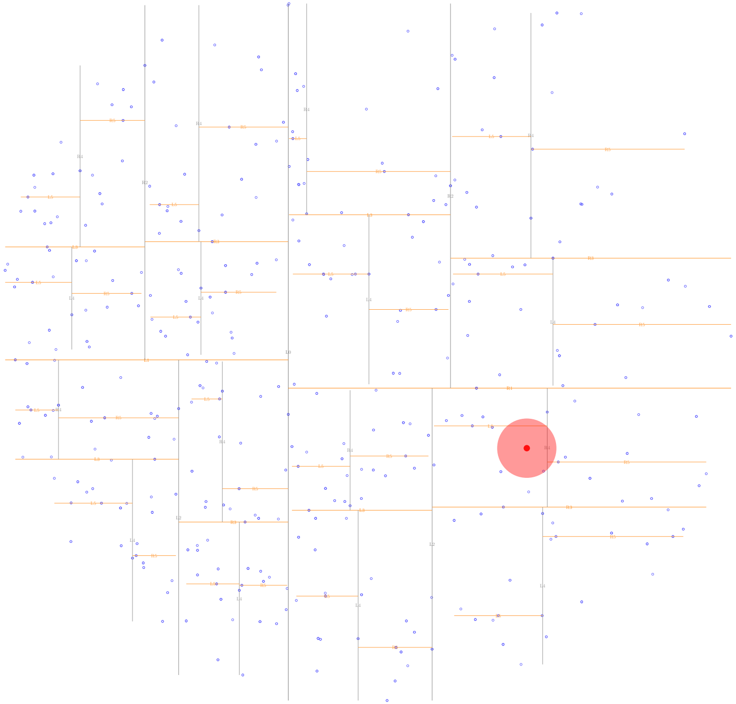
KD-Tree

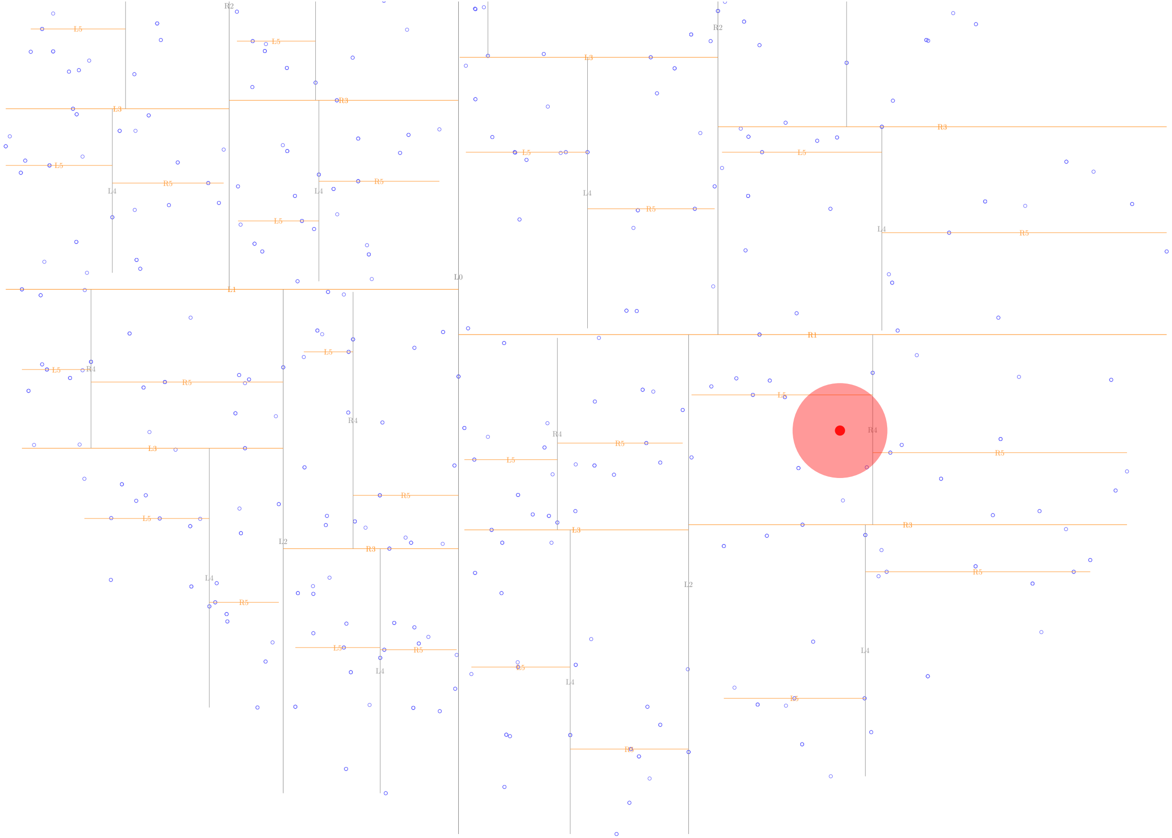
Each node in tree maintains variable "box"

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

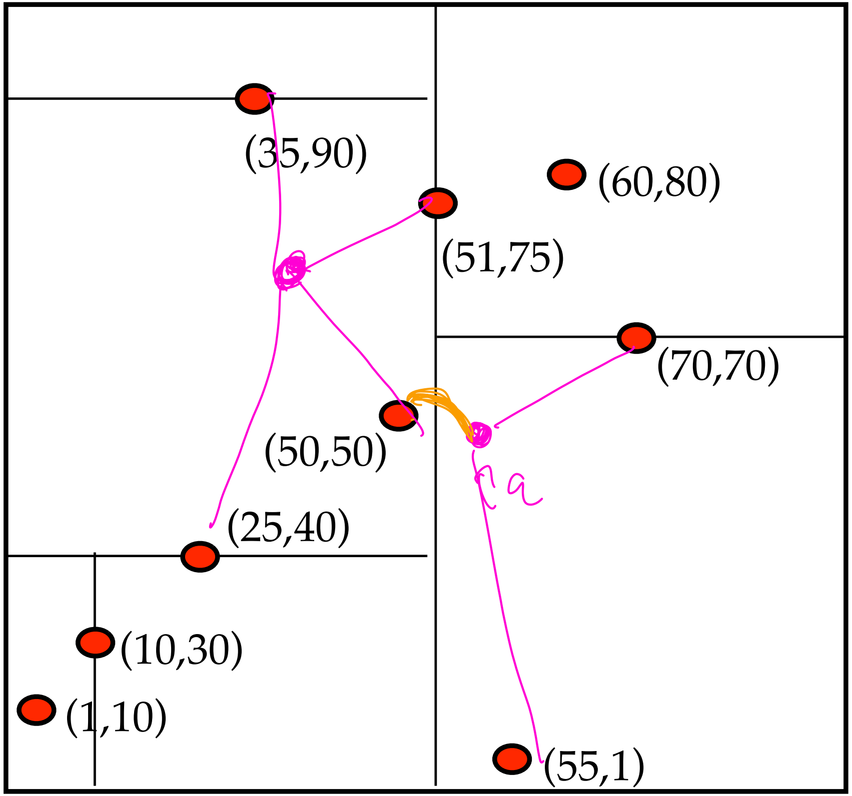

x
y
y
x
x
x
x
y
y
y
y
y
y
y
y







L0
L1
R1
L2
R2
L3
R3
L4
R4
L5
R5


(35,90)
(60,80)
(51,75)
(70,70)
(50,50)
na
(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)
    }
```

heuristically

$T(n) = 2T(\frac{n}{2}) + \theta(1)$

$\log(n)$