07-A1

BST Application

Range Query: 1D

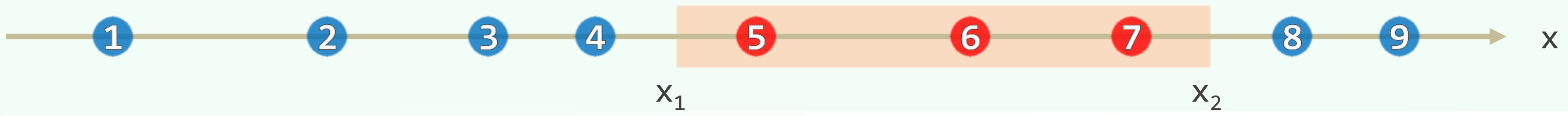
邓俊辉
deng@tsinghua.edu.cn

你这个人太敏感了。这个社会什么都需要，唯独不需要敏感。

1D Range Query

- Let $P = \{ p_1, p_2, p_3, \ldots, p_n \}$ be a set of n points on the x-axis
- For any given interval $I = (x_1, x_2]$
  - COUNTING: how many points of P lies in the interval?
  - REPORTING: enumerate all points in $I \cap P$ (if not empty)
- [Online] P is fixed while I is randomly and repeatedly given
- How to PREPROCESS P into a certain data structure s.t.

  the queries can be answered efficiently?

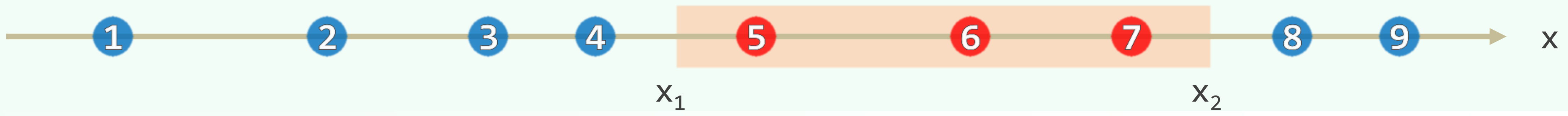


Brute-Force

- For each point p of P, test if $p \in (x_1, x_2]$
- Thus each query can be answered in LINEAR time
- Can we do it faster? It seems we can't, for ...
- In the worst case,

  the interval contains up to $\mathcal{O}(n)$ points, which need $\mathcal{O}(n)$ time to enumerate
- However, how if we

  ignore the time for enumerating and count only the searching time?



Binary Search

❖ Sort all points into a sorted vector and add an extra sentinel p[0] = -∞

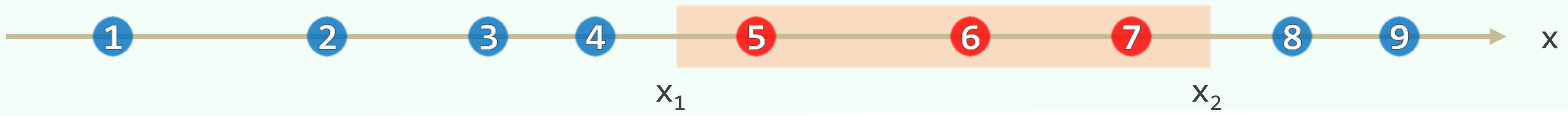
❖ For any interval $I = (x_1, x_2]$

- Find t = search(x_2) = max{ i | p[i] ≤ x_2 } //$\mathcal{O}$(logn)

- Traverse the vector BACKWARD from p[t] and report each point //$\mathcal{O}$(r)

  until escaping from I at point p[s]

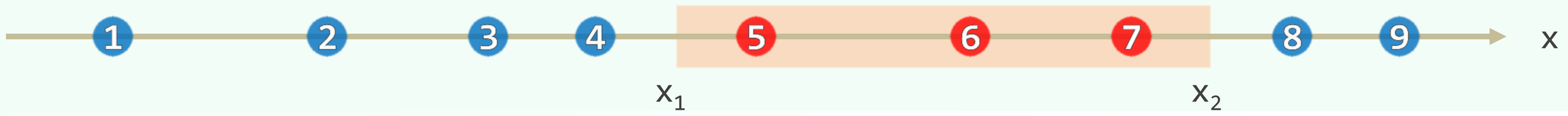
- return r = t - s //output size



Output-Sensitivity

- An enumerating query can be answered in $\mathcal{O}(r + \log n)$ time
- p[s] can also be found by binary search in $\mathcal{O}(\log n)$ time
- Hence for COUNTING query, $\mathcal{O}(\log n)$ time is enough //independent to r
- Can this simple strategy be extended to PLANAR range query?

  TTBOMK, unfortunately, no!



07-A2

BST Application

Range Query: 2D

昔者明王必尽知天下良士之名；既知其名，又知其数；
既知其数，又知其所在。

邓俊辉
deng@tsinghua.edu.cn

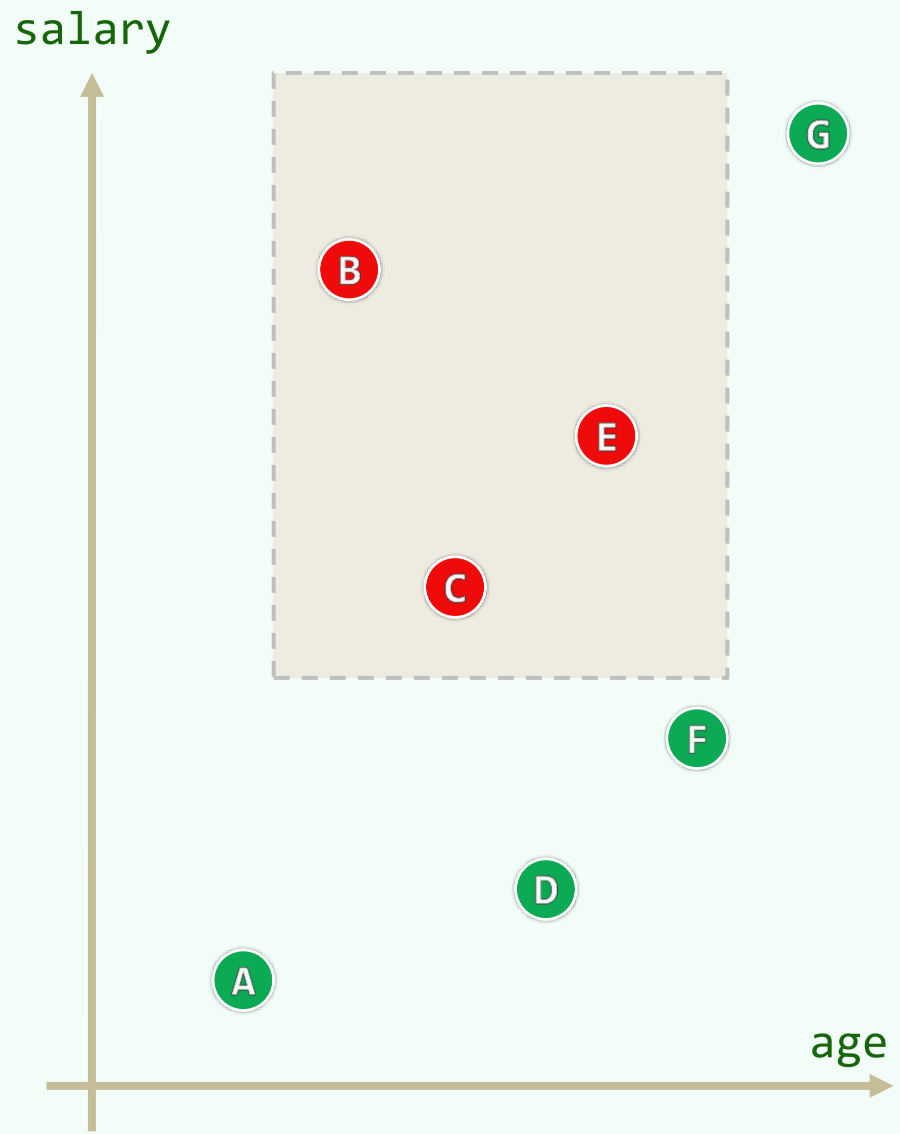
Planar Range Query

- Let $P = \{ p_1, p_2, p_3, \ldots, p_n \}$ be a planar set
- Given $R = (x_1, x_2] \times (y_1, y_2]$
  - COUNTING: $|R \cap P| = ?$
  - REPORTING: $R \cap P = ?$
- Binary search

  doesn't help this kind of query
- You might consider to

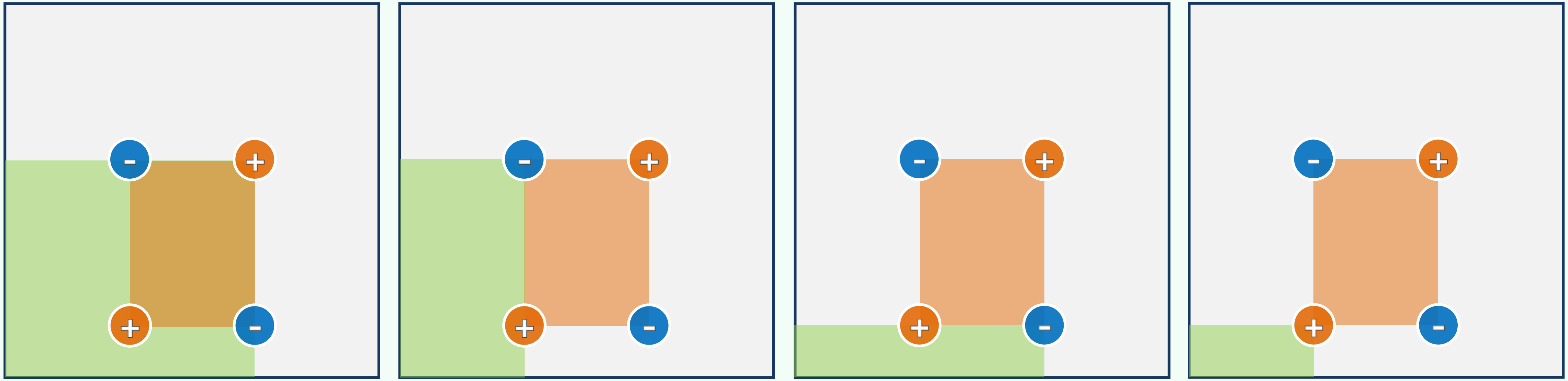
  expand the counting method using

  the Inclusion-Exclusion Principle



Preprocessing

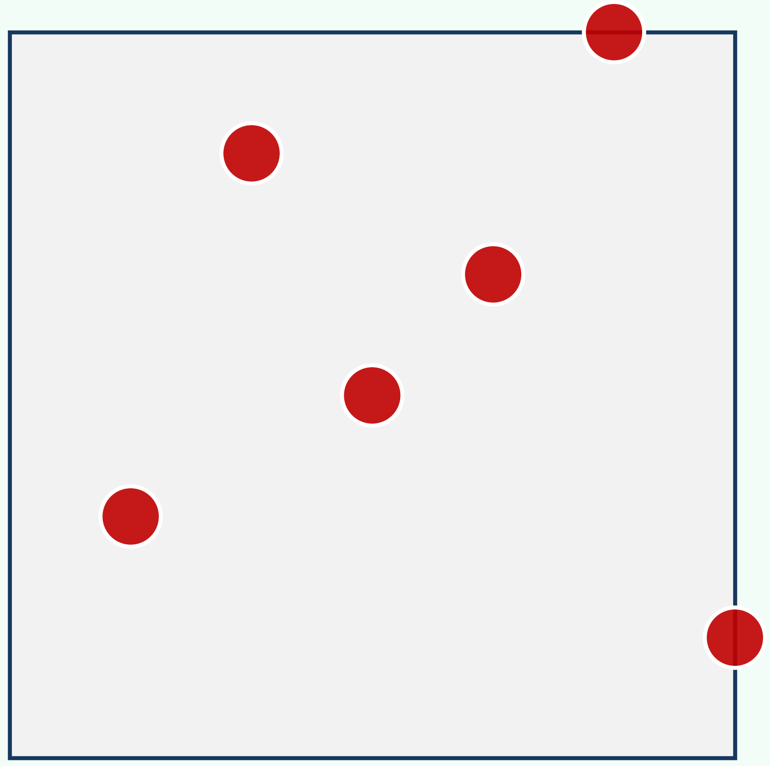
- $\forall$ point (x, y), let $n(x, y) = \left|((0, x] \times (0, y]) \cap P\right|$
- **This requires $\mathcal{O}(n^2)$ time/space**

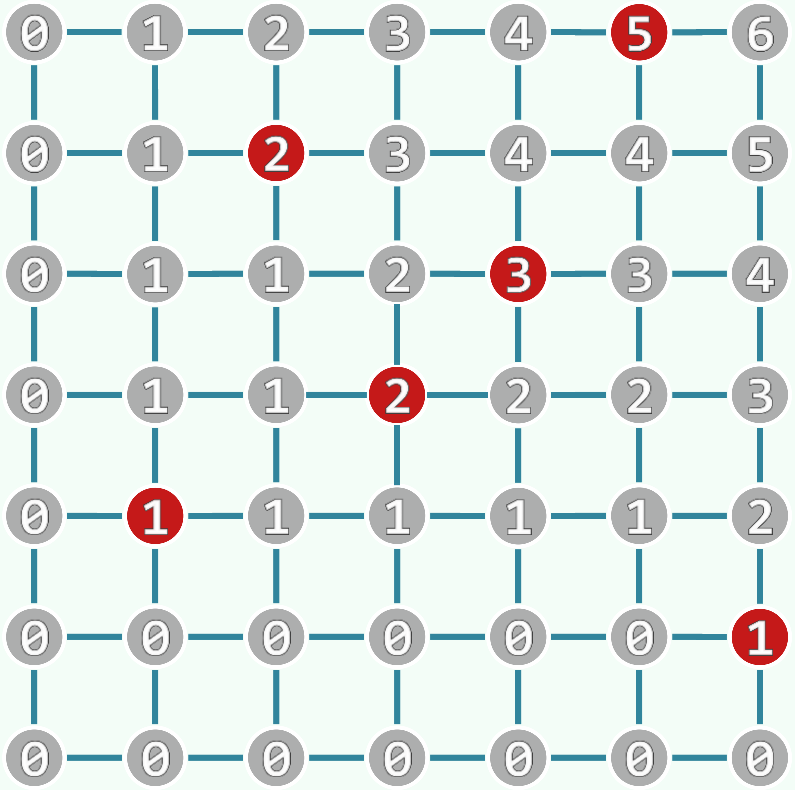


Domination

❖ A point (u, v) is called to be **DOMINATED** by point (x, y) if

$u \leq x$ and $v \leq y$

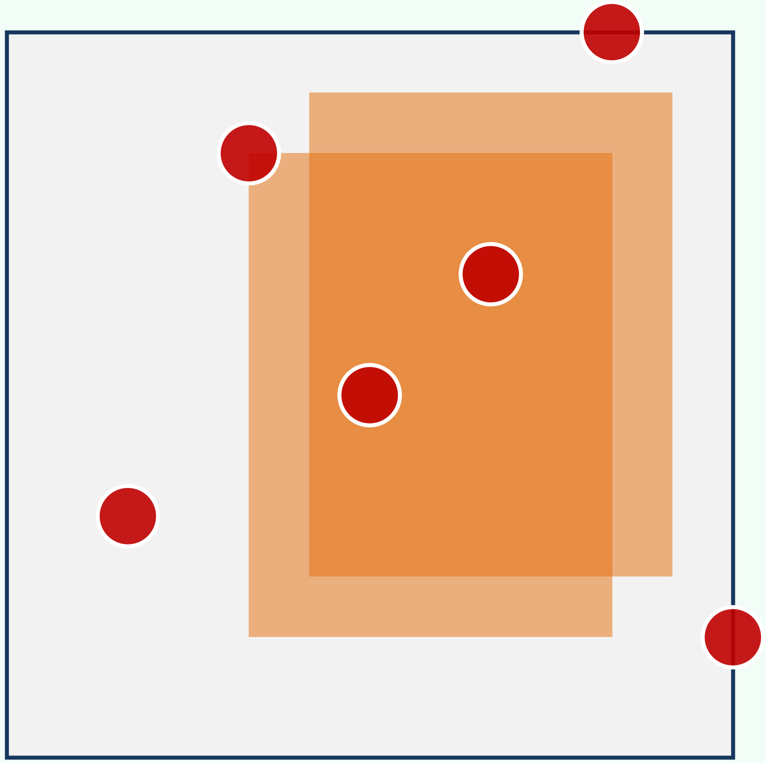


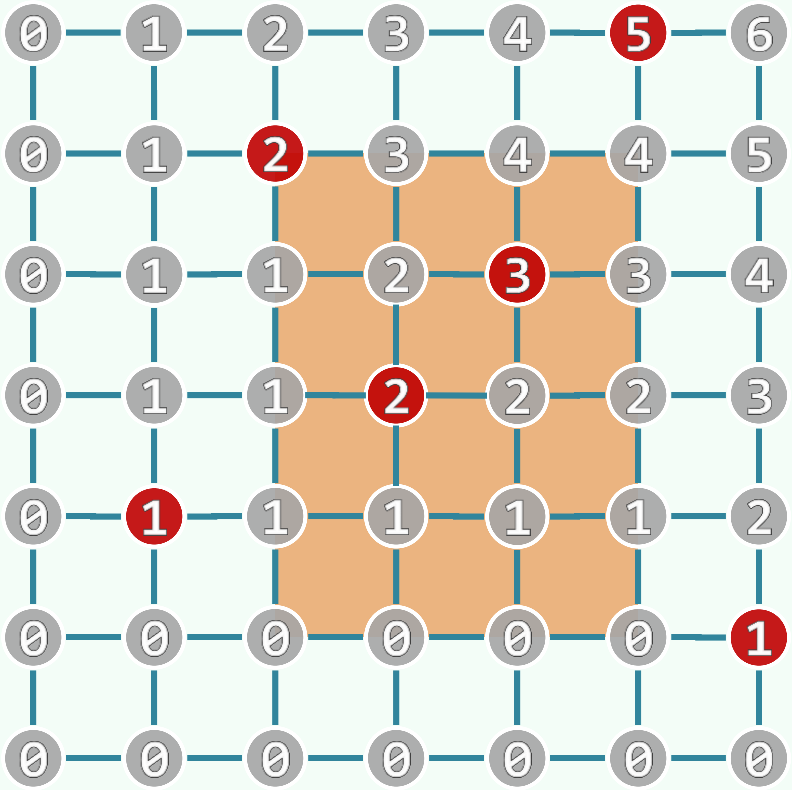


Inclusion-Exclusion Principle

❖ **Then for any rectangular range $\mathcal{R} = (x_1, x_2] \times (y_1, y_2]$, we have**

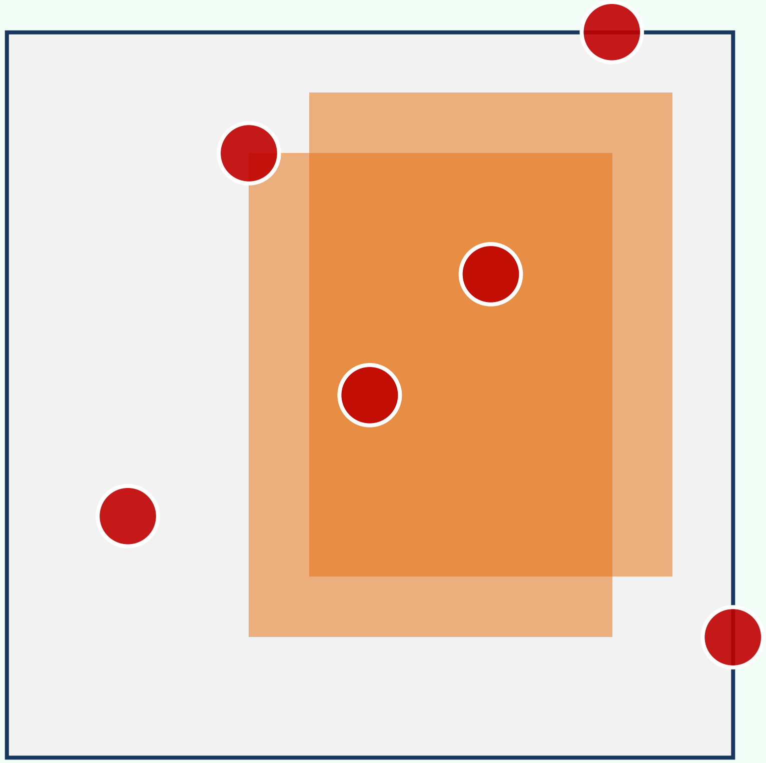
$$\left|\mathcal{R} \cap \mathcal{P}\right| \;=\; n(x_1, y_1) + n(x_2, y_2) - n(x_1, y_2) - n(x_2, y_1)$$

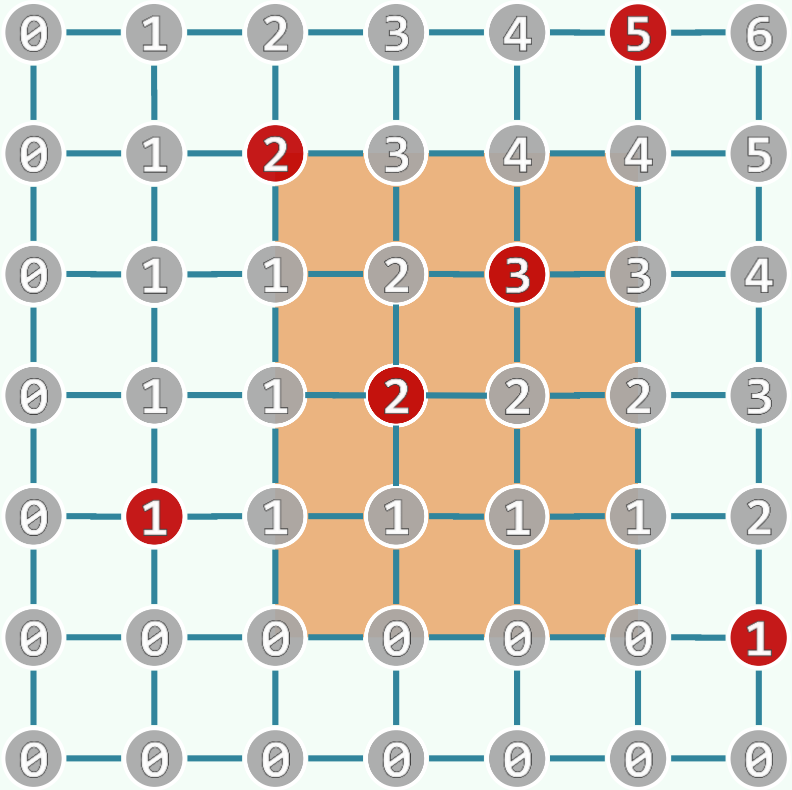




Performance

- Each query needs only $\mathcal{O}(\log n)$ time
- Uses $\Theta(n^2)$ storage and even more for higher dimensions
- To find a better solution, let's go back to the **1D** case ...





07-B1

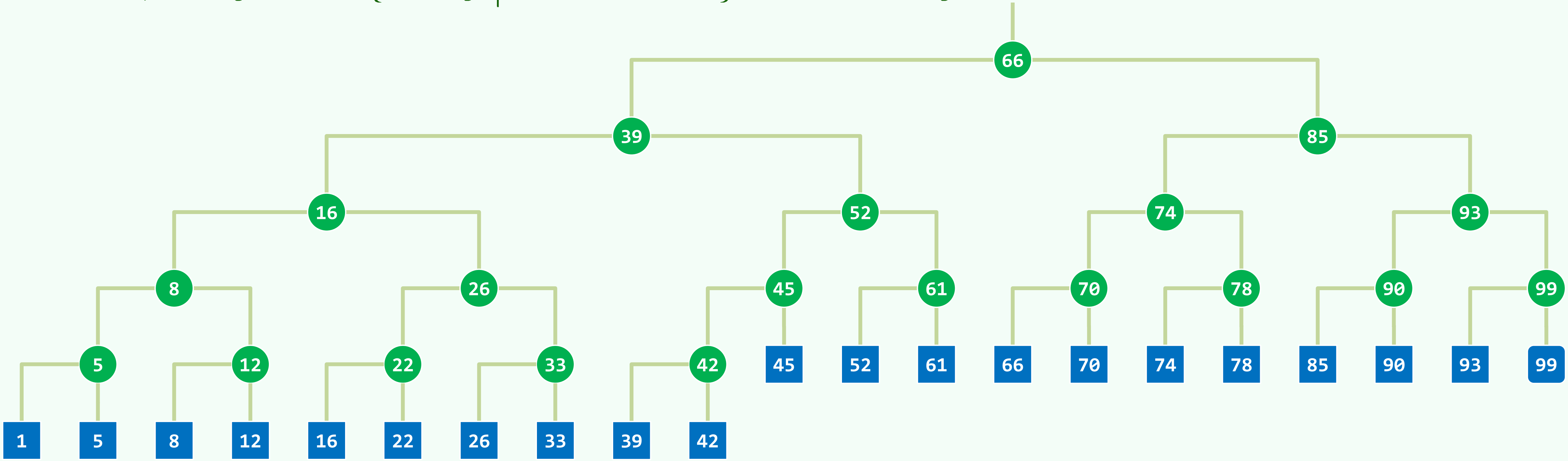
BST Application

kd-Tree: 1D

邓俊辉
deng@tsinghua.edu.cn

Structure: A Complete (Balanced) BST

❖ $\forall v,\ v.key = \min\{\, u.key \mid u \in v.rTree \,\} = v.succ.key$

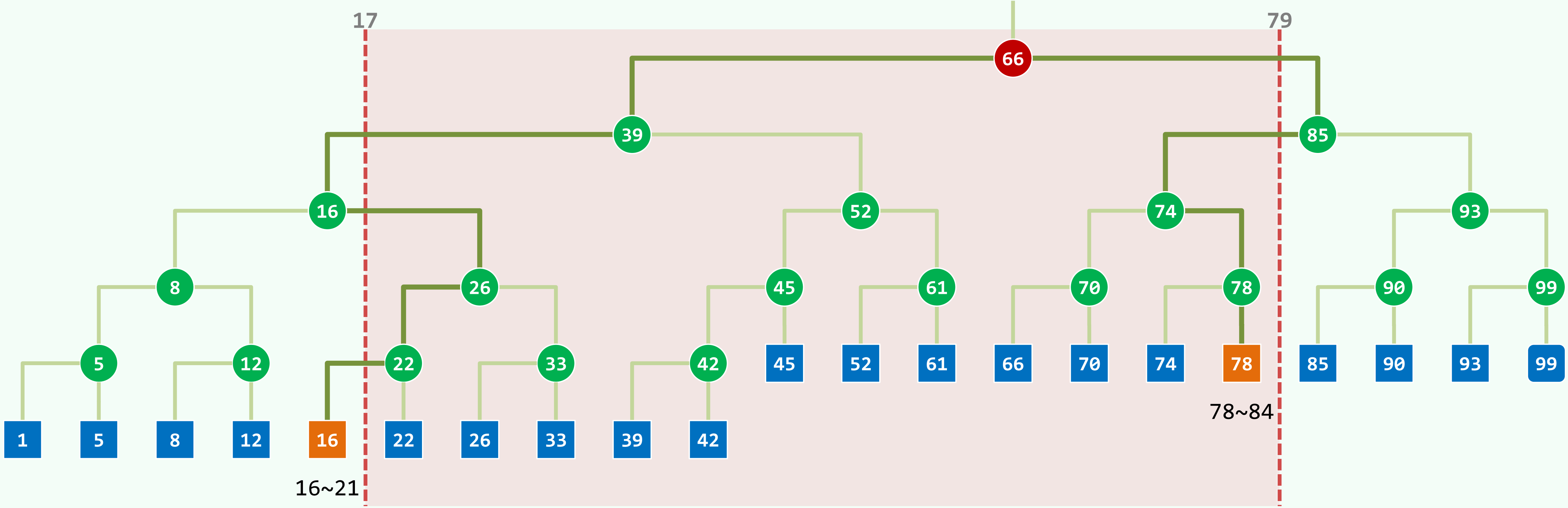


❖ $\forall u \in v.lTree,\ u.key < v.key$ $\qquad \forall u \in v.rTree,\ u.key \geq v.key$

❖ search(x) : returns the **MAXIMUM** key not greater than x

Lowest Common Ancestor

❖ Consider, as an example, the query for [17, 79] ...

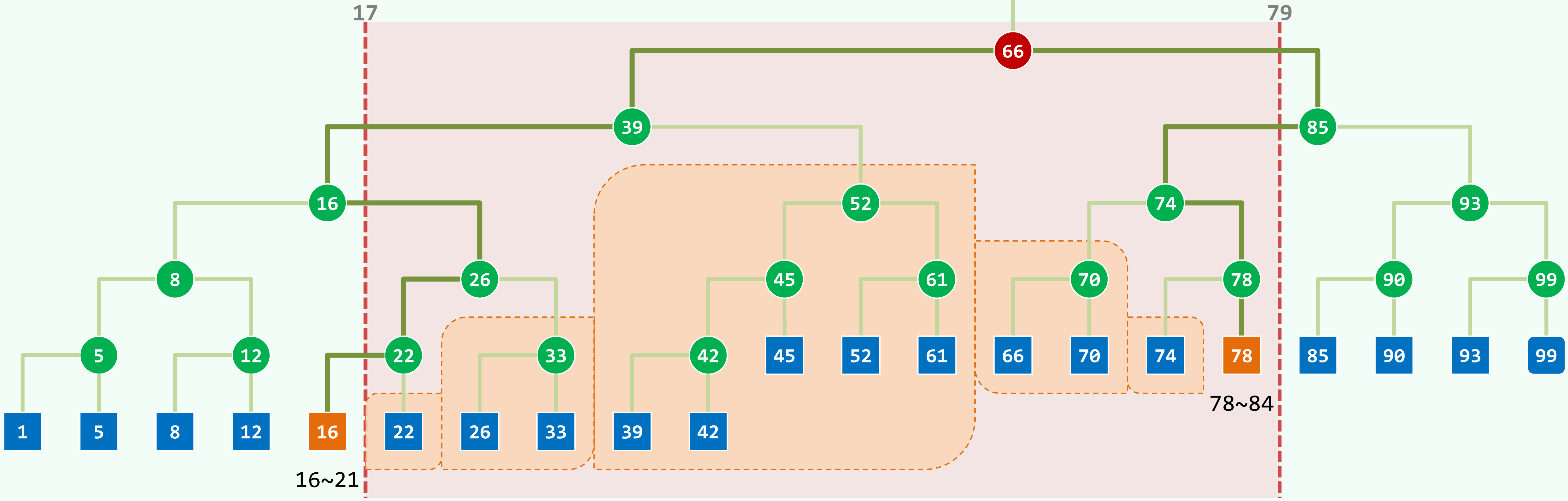


❖ Do search(17) = 16 (might rejected) and search(79) = 78 (must accepted)

❖ Consider LCA(16, 78) = 66 ...

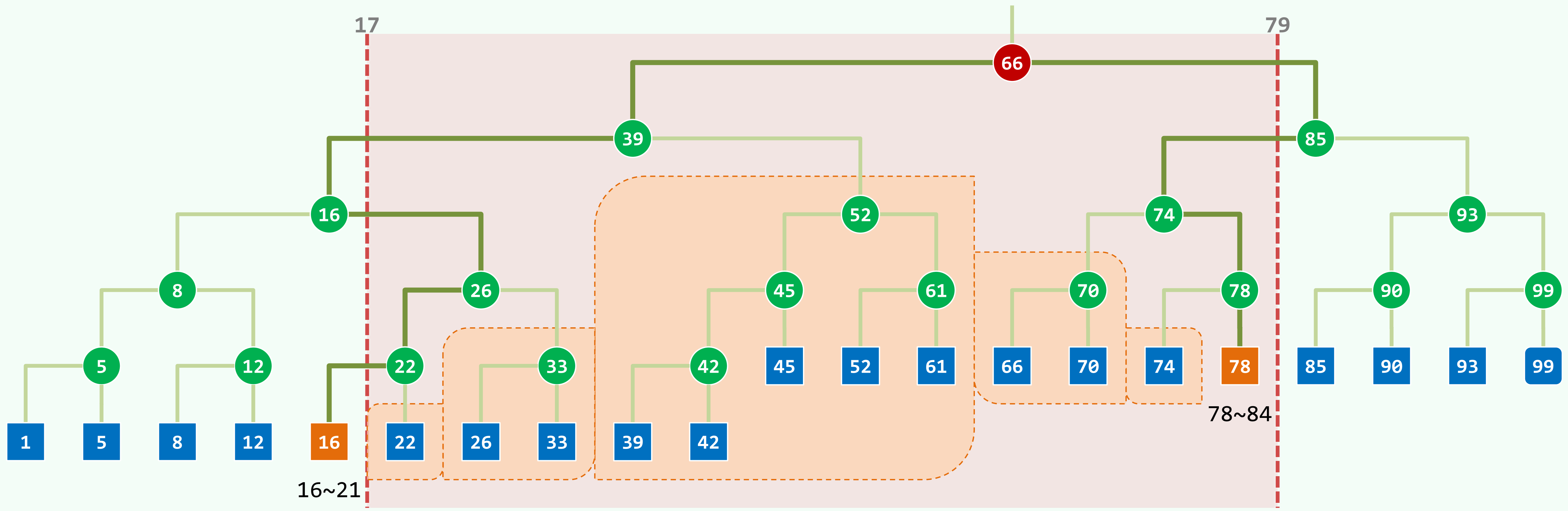
Union of $\mathcal{O}(\log n)$ Disjoint Subtrees

❖ Starting from the LCA, traverse path(16) and path(78) once more resp.



- All R/L-turns along path(16)/path(78) are ignored and
- the R/L subtree at each L/R-turn is reported

Complexity

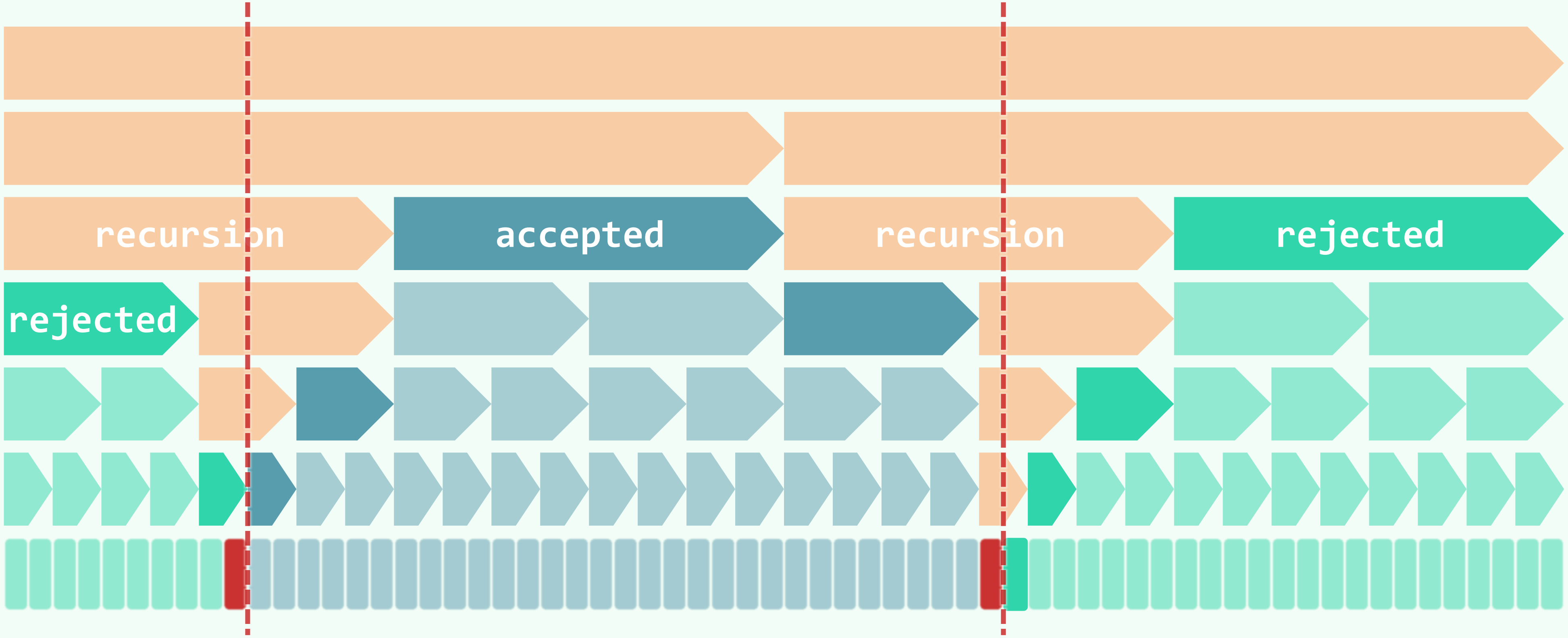


Query: $\mathcal{O}(\log n)$ **Preprocessing:** $\mathcal{O}(n \log n)$ **Storage:** $\mathcal{O}(n)$

Hot Knives Through A Chocolate Cake of Height $\mathcal{O}(\log n)$

❖ Region(u) is enclosed by Region(v) iff u is a descendant of v in the 1d-tree

❖ Region(u) and Region(v) are disjoint iff neither is the ancestor of the other



❖ All nodes are partitioned into 3 types: accepted + rejected + recursion

07-B2

BST Application

kd-Tree: 2D

凡见字数，如停匀，即平分一半为上卦，一半为下卦。如字数不均，即少一字为上卦，取天轻清之义，以多一字为下卦，取地重浊之义

邓俊辉

deng@tsinghua.edu.cn

Divide-And-Conquer

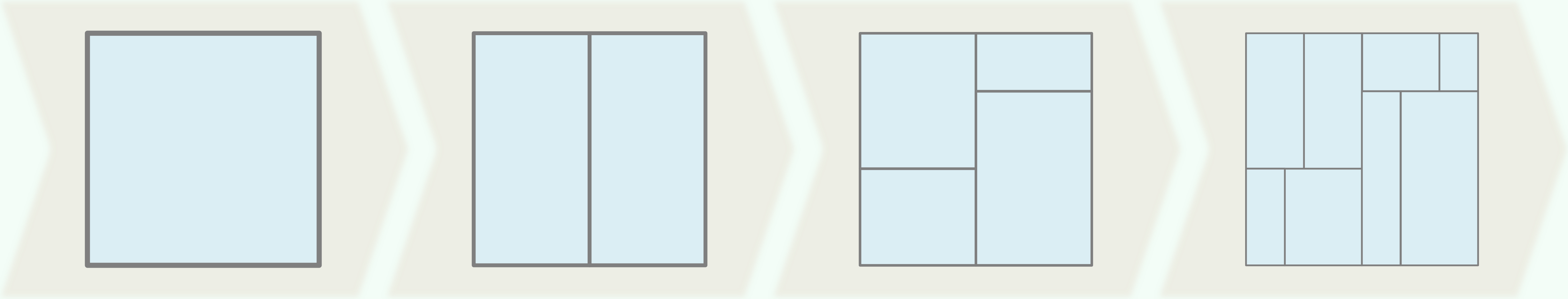
❖ To extend the BBST method to planar GRS, we

- **divide** the plane recursively and
- **arrange** the regions into a kd-tree

❖ Start with a single region (the entire plane)

Partition the region vertically/horizontally on each even/odd level

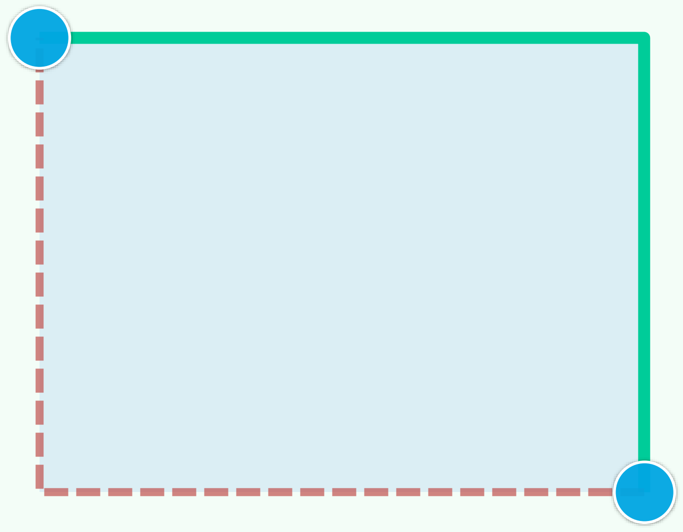
Partition the sub-regions recursively

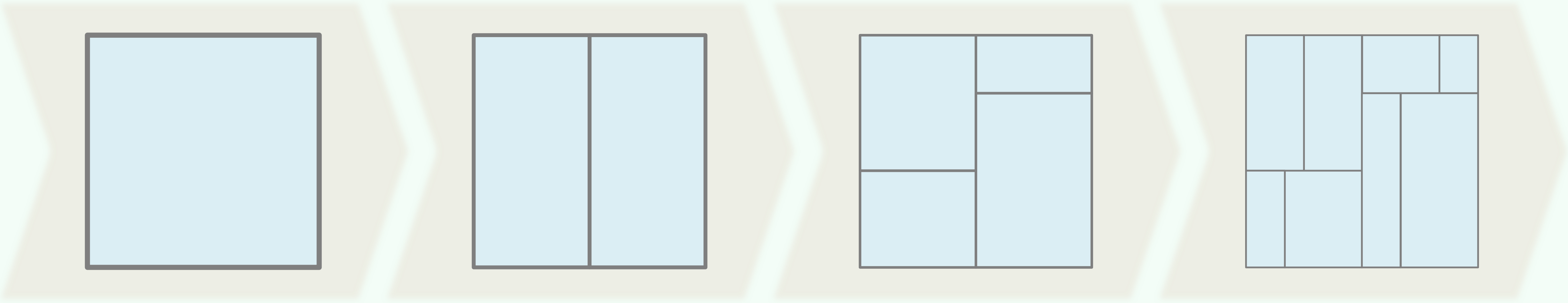


More Details

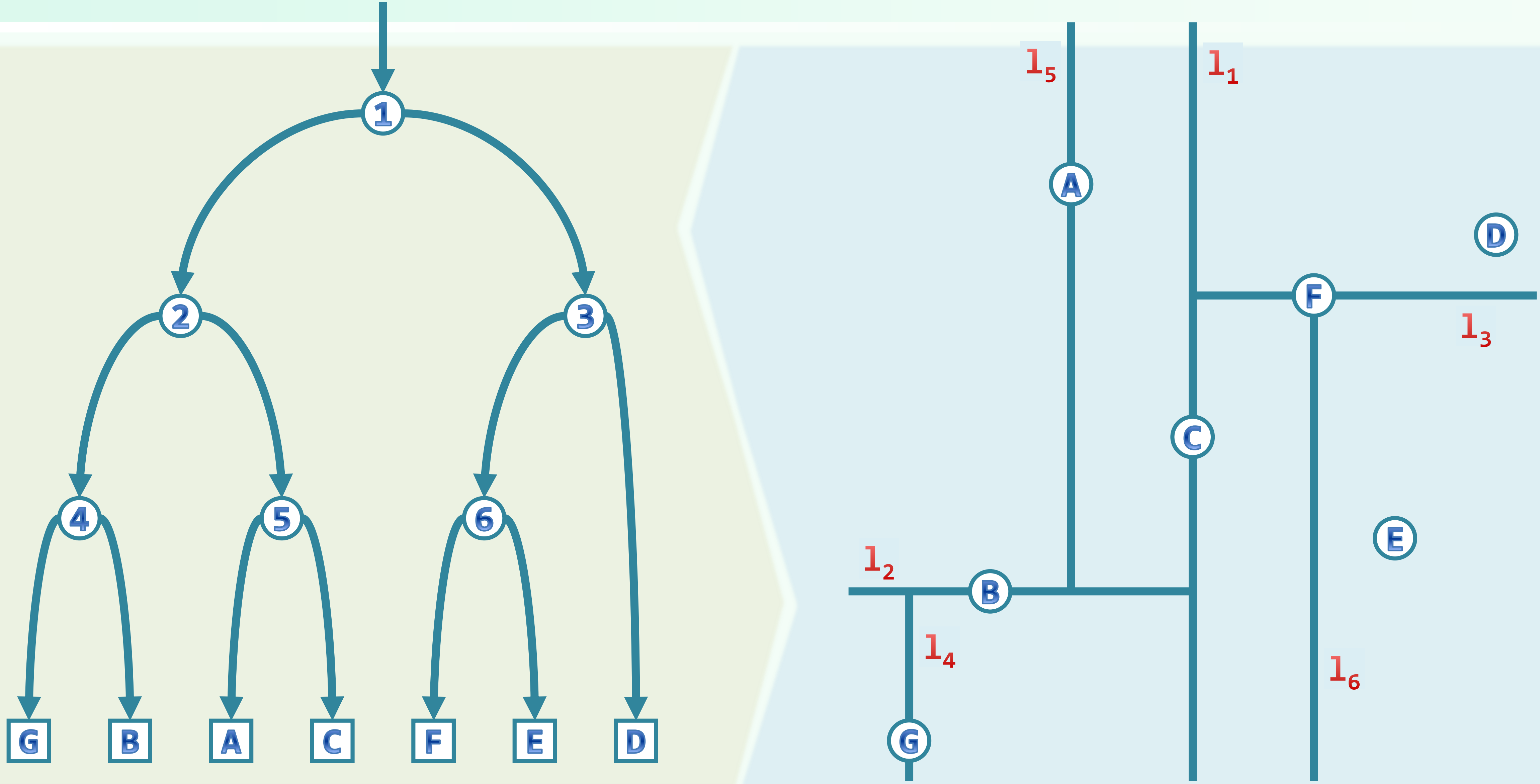
❖ To make it work,

- each partition should be done as evenly as possible (at median)
- each region is defined to be open/closed on the left-lower/right-upper sides

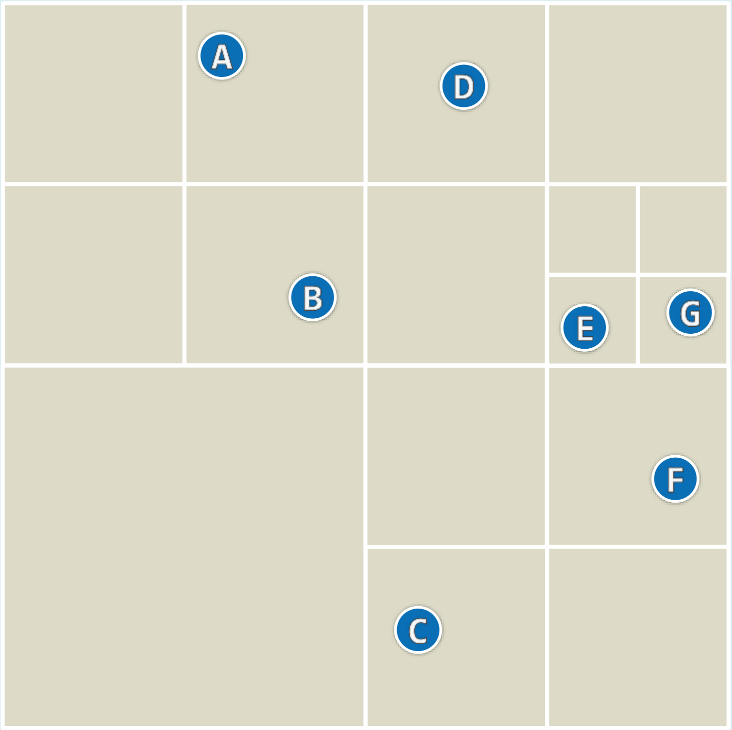


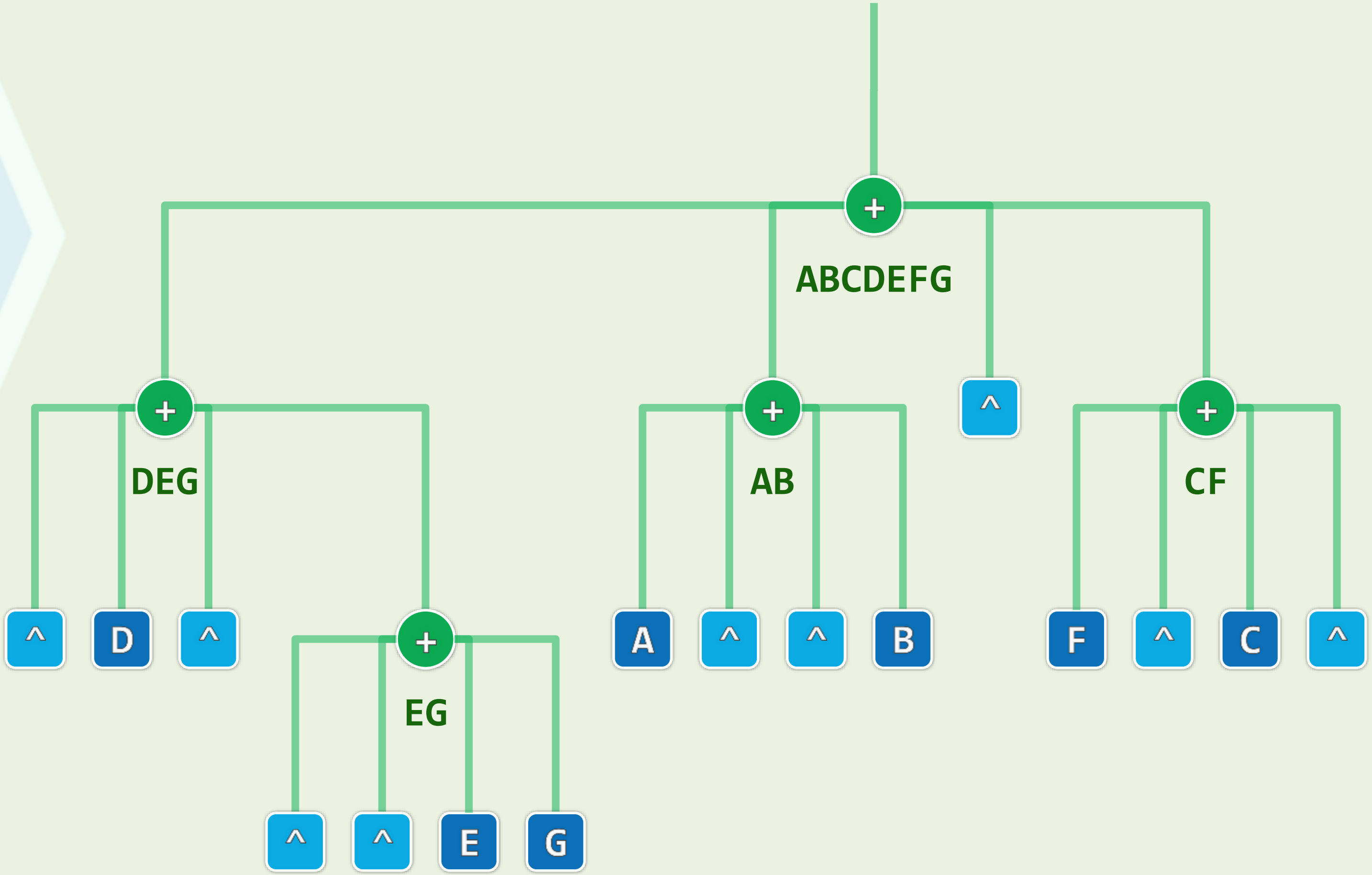


Example



Quadtree





07-B3

BST Application

kd-Tree: Construction

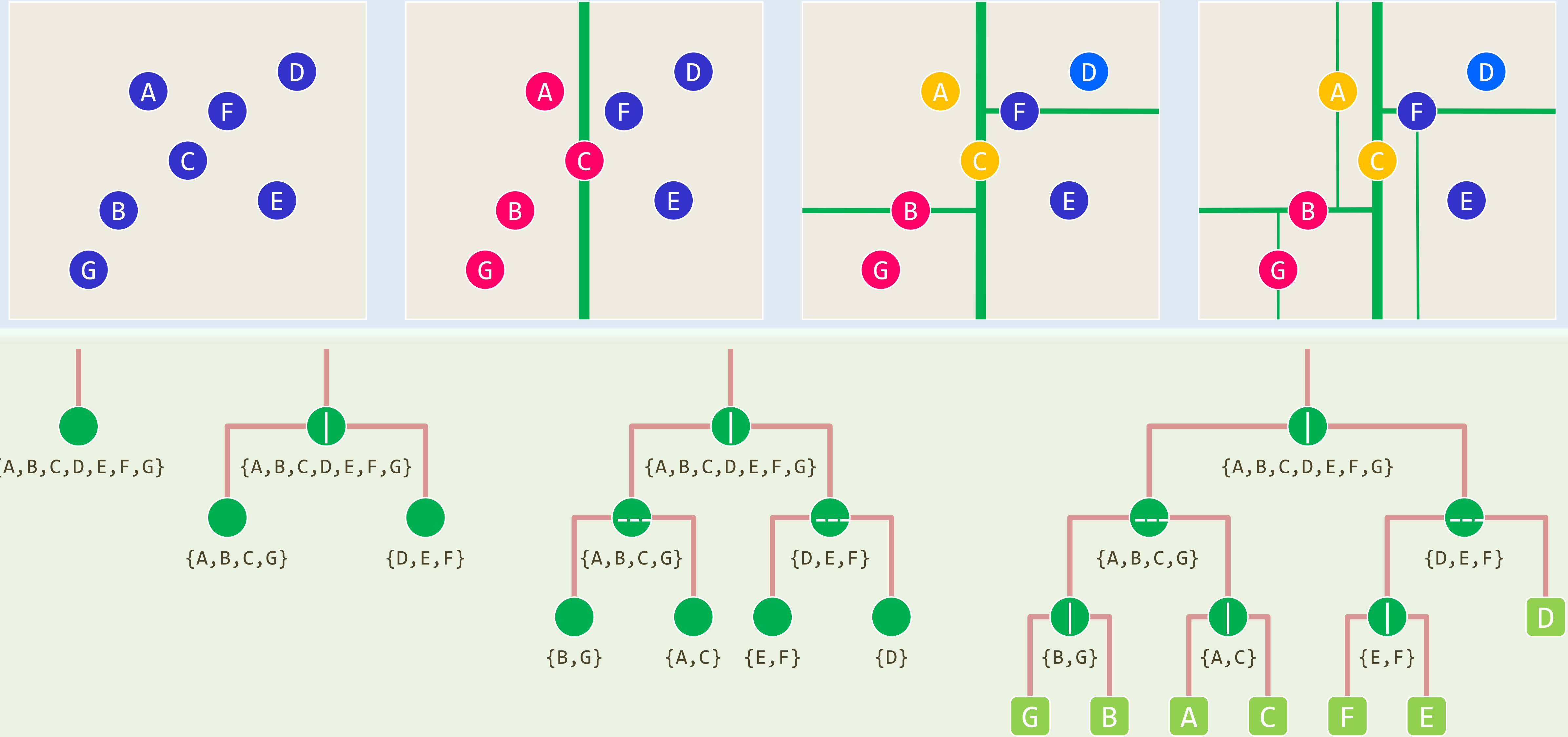
邓俊辉
deng@tsinghua.edu.cn

God found himself by creating.

buildKdTree(P,d)

```
// construct a 2d-(sub)tree for point (sub)set P at depth d
   if ( P == {p} ) return CreateLeaf( p )  //base
   root = CreateKdNode()
   root->splitDirection = Even(d) ? VERTICAL : HORIZONTAL
   root->splitLine = FindMedian( root->splitDirection, P )  //𝒪(n)!
   ( P1, P2 ) = Divide( P, root->splitDirection, root->splitLine )  //DAC
   root->lc = buildKdTree( P1, d + 1 )  //recurse
   root->rc = buildKdTree( P2, d + 1 )  //recurse
   return( root )
```

Example



07-B4

BST Application

kd-Tree: Canonical Subsets

韦小宝跟著她走到桌边，只见桌上大白布上钉满了几千枚绣花针，几千块碎片已拼成一幅完整无缺的大地图，难得的是几千片碎皮拼在一起，既没多出一片，也没少了一片。

邓俊辉

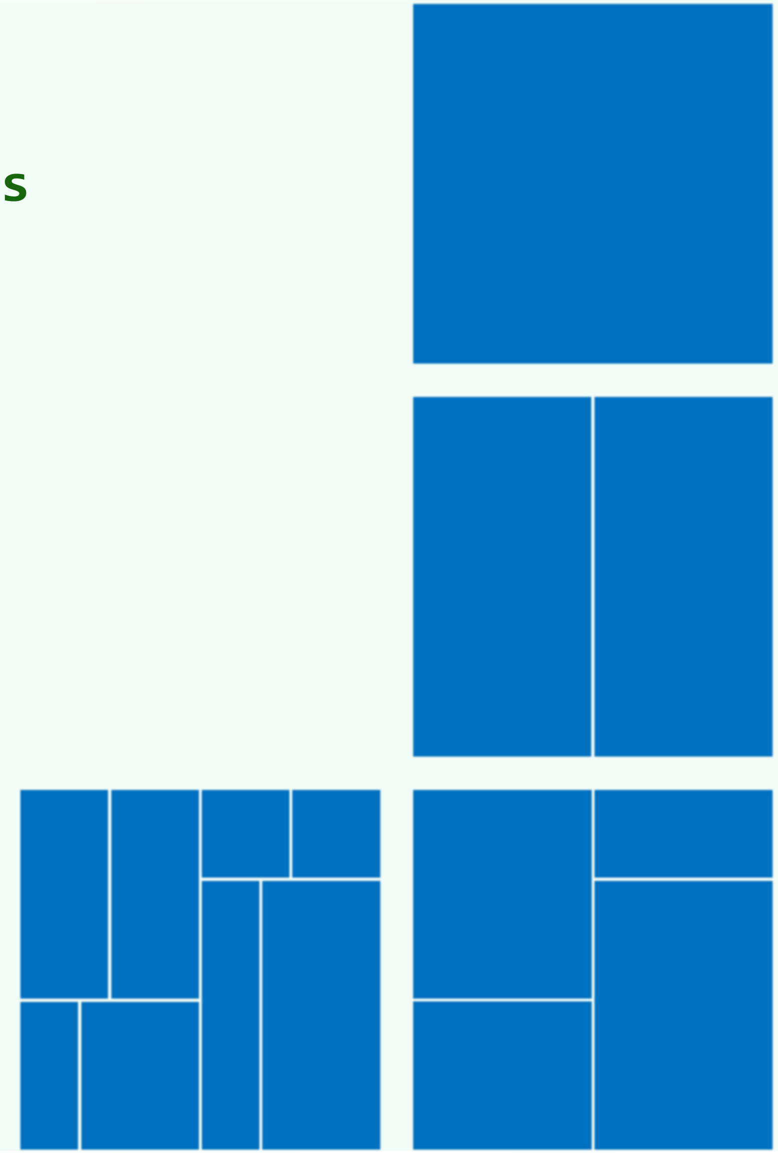
deng@tsinghua.edu.cn

Canonical Subset

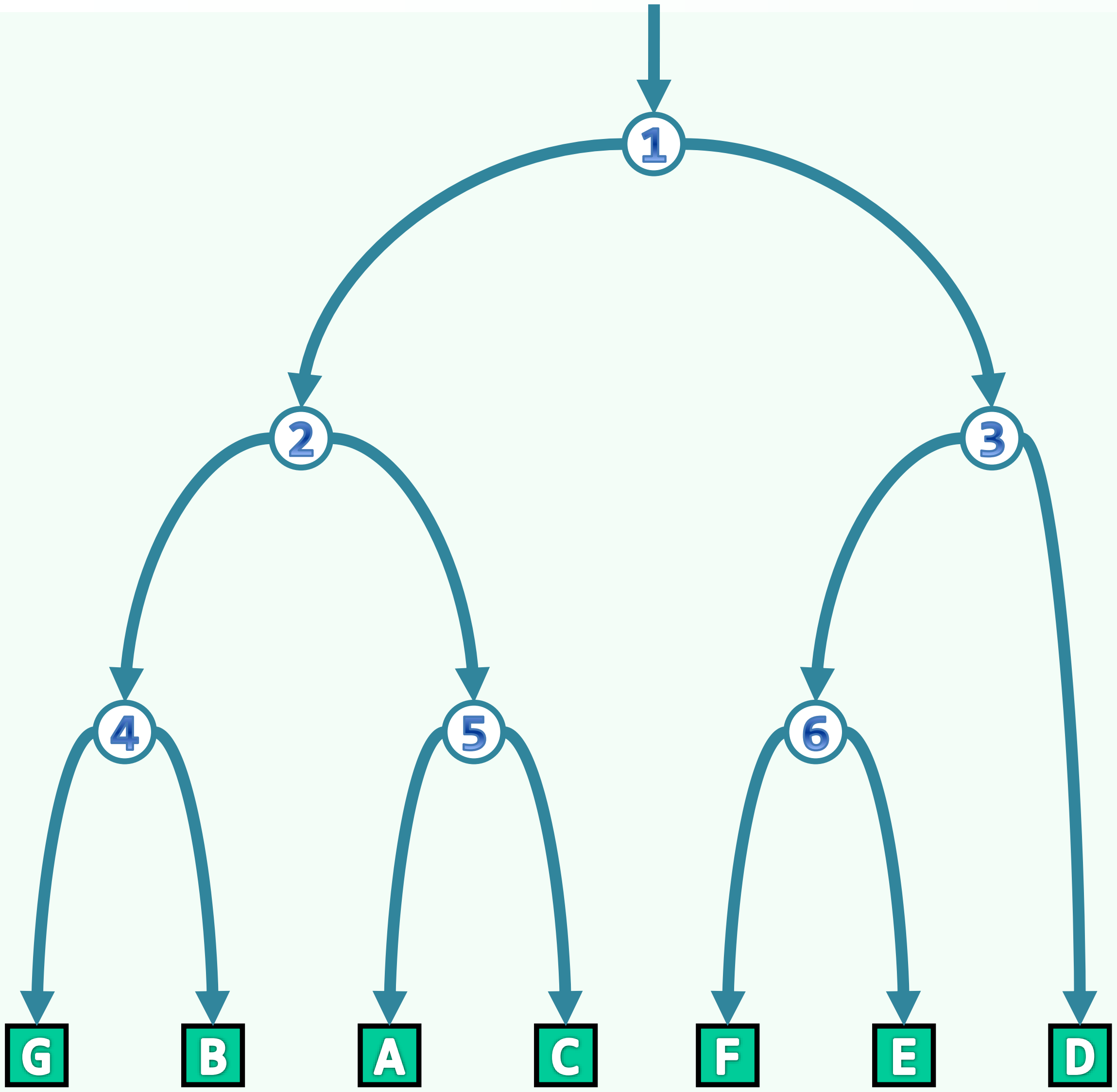
- Each node corresponds to
  - a rectangular sub-region of the plane, as well as
  - the subset of points contained in the sub-region
- Each of these subsets is called a canonical subset
- For each internal node X with children L and R,

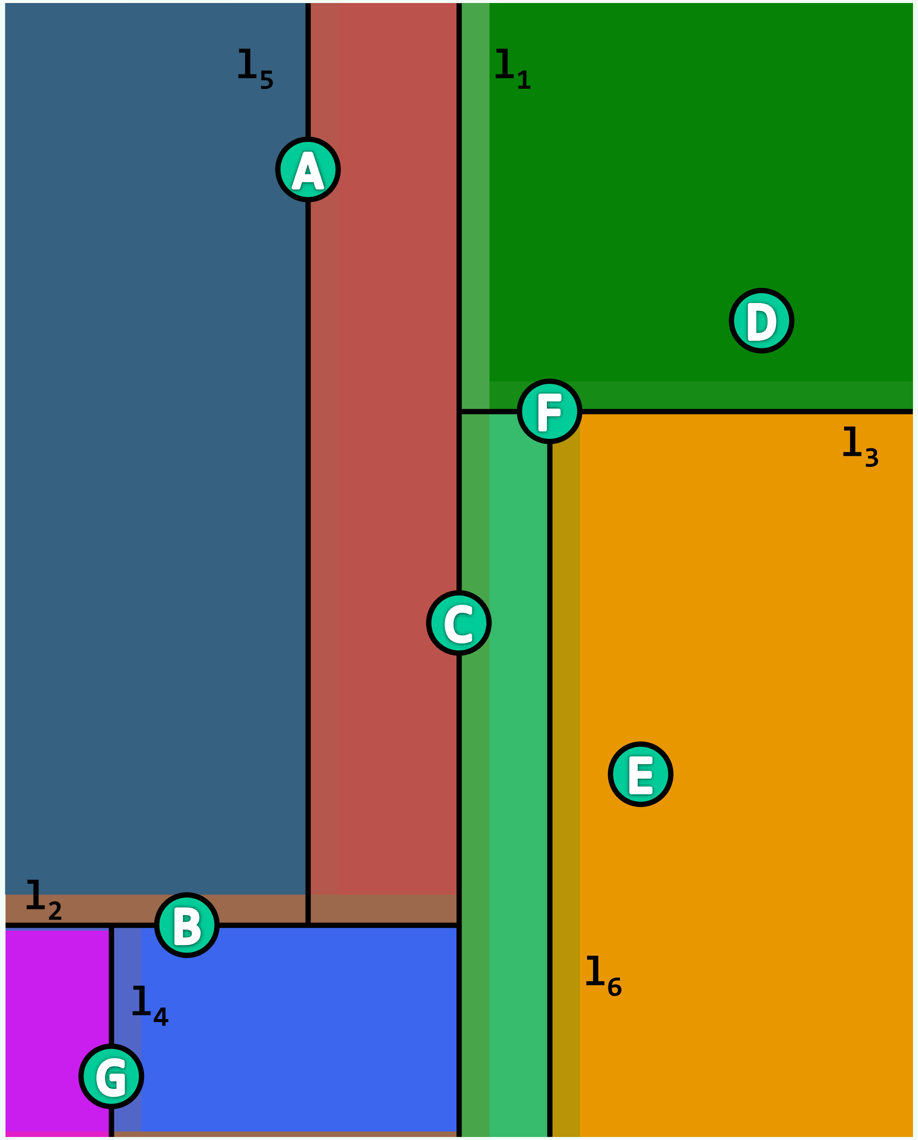
  region(X) = region(L) ∪ region(R)

- Sub-regions of nodes at a same depth
  - never intersect with each other, and
  - their union covers the entire plane
- We will see soon that each 2D GRS can be answered by the union of a number of CS's



Example





07-B5

BST Application

kd-Tree: Query

我们竟为这无用的找寻浪费了这么多天！

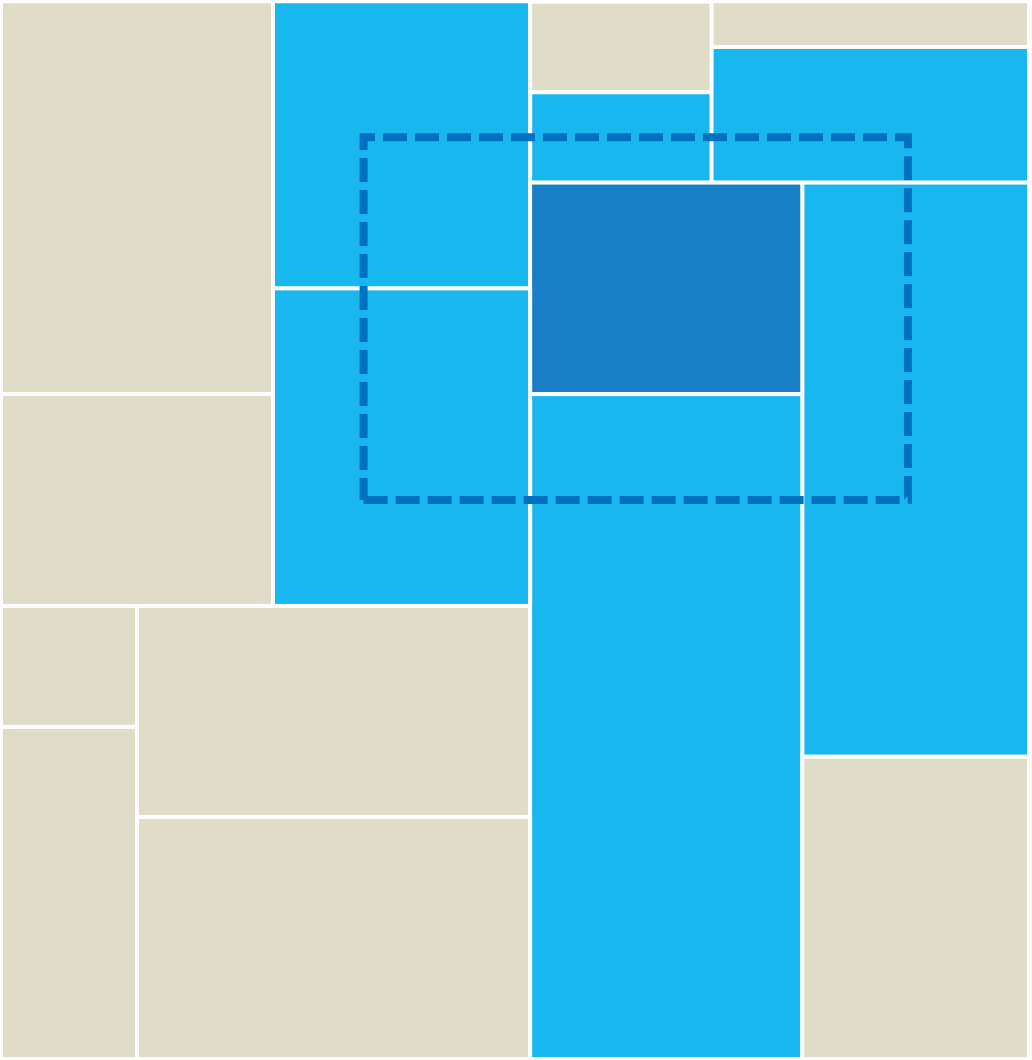
我想找寻的心上人绝对不会在这里出现。

邓俊辉

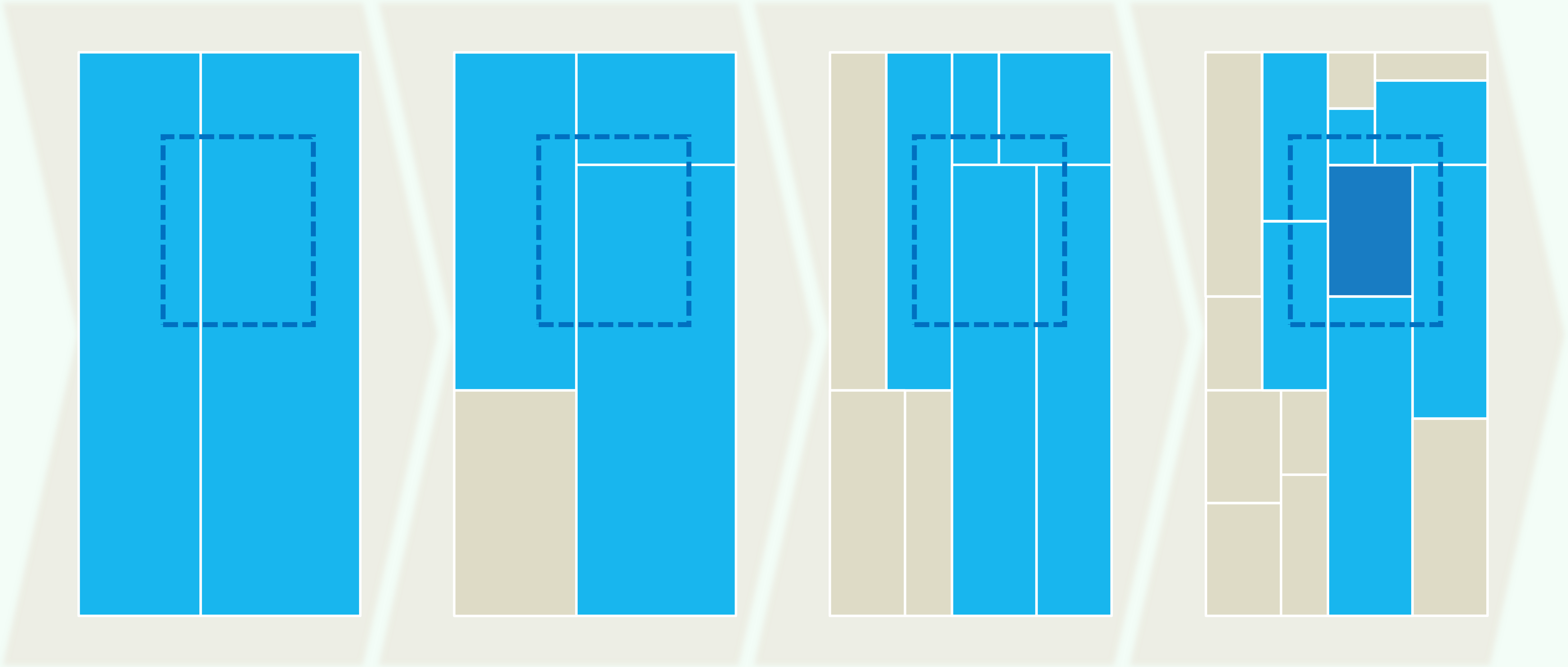
deng@tsinghua.edu.cn

kdSearch(v,R): 热刀来切~~千~~(logn)层巧克力

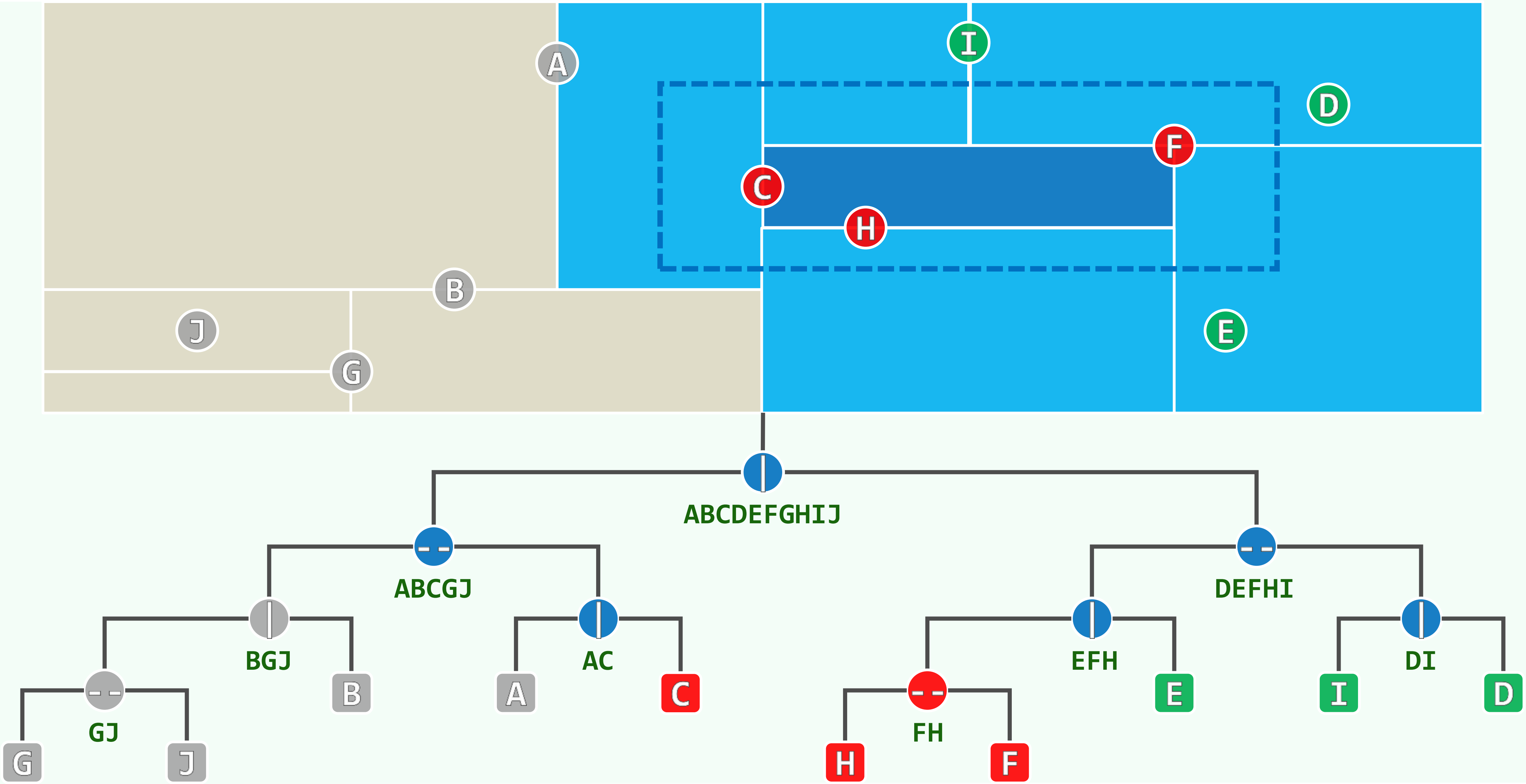
```
❖ if ( isLeaf( v ) )
      if ( inside( v, R ) ) report(v)
      return
❖ if ( region( v->lc ) ⊆ R )
      reportSubtree( v->lc )
  else if ( region( v->lc ) ∩ R ≠ ∅ )
      kdSearch( v->lc, R )
❖ if ( region( v->rc ) ⊆ R )
      reportSubtree( v->rc )
  else if ( region( v->rc ) ∩ R ≠ ∅ )
      kdSearch( v->rc, R )
```



Example

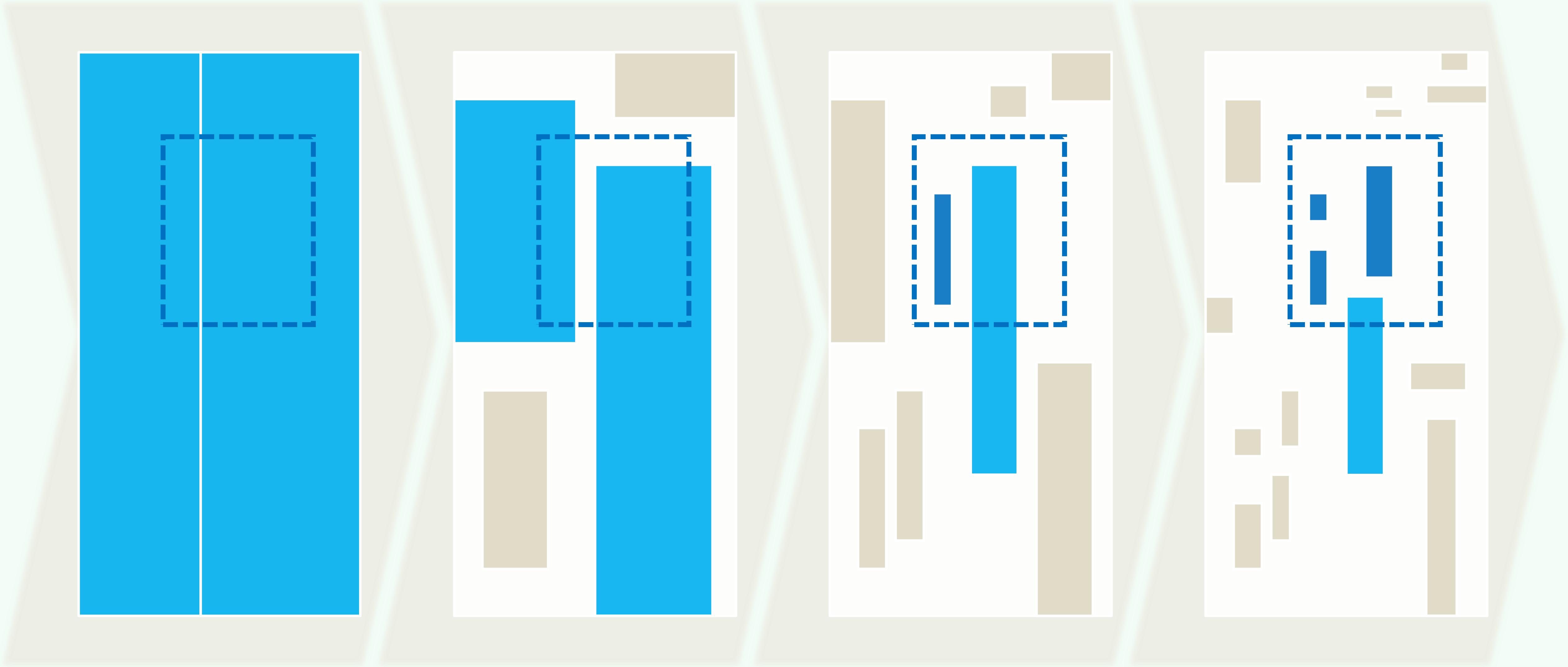


Example



A
I
D
F
C
H
B
J
G
E
ABCDEFGHIJ
ABCGJ
DEFHI
BGJ
AC
EFH
DI
GJ
B
A
C
FH
E
I
D
G
J
H
F

Bounding Box



07-B6

BST Application

kd-Tree: Complexity

肉眼看不清细节，但他们都知道那是木星所在的位置，这颗太阳系最大的行星已经坠落到二维平面上了。

有人嘲笑这种体系说：为了能发现这个比例中项并组成政府共同体，按照我的办法，只消求出人口数字的平方根就行了。

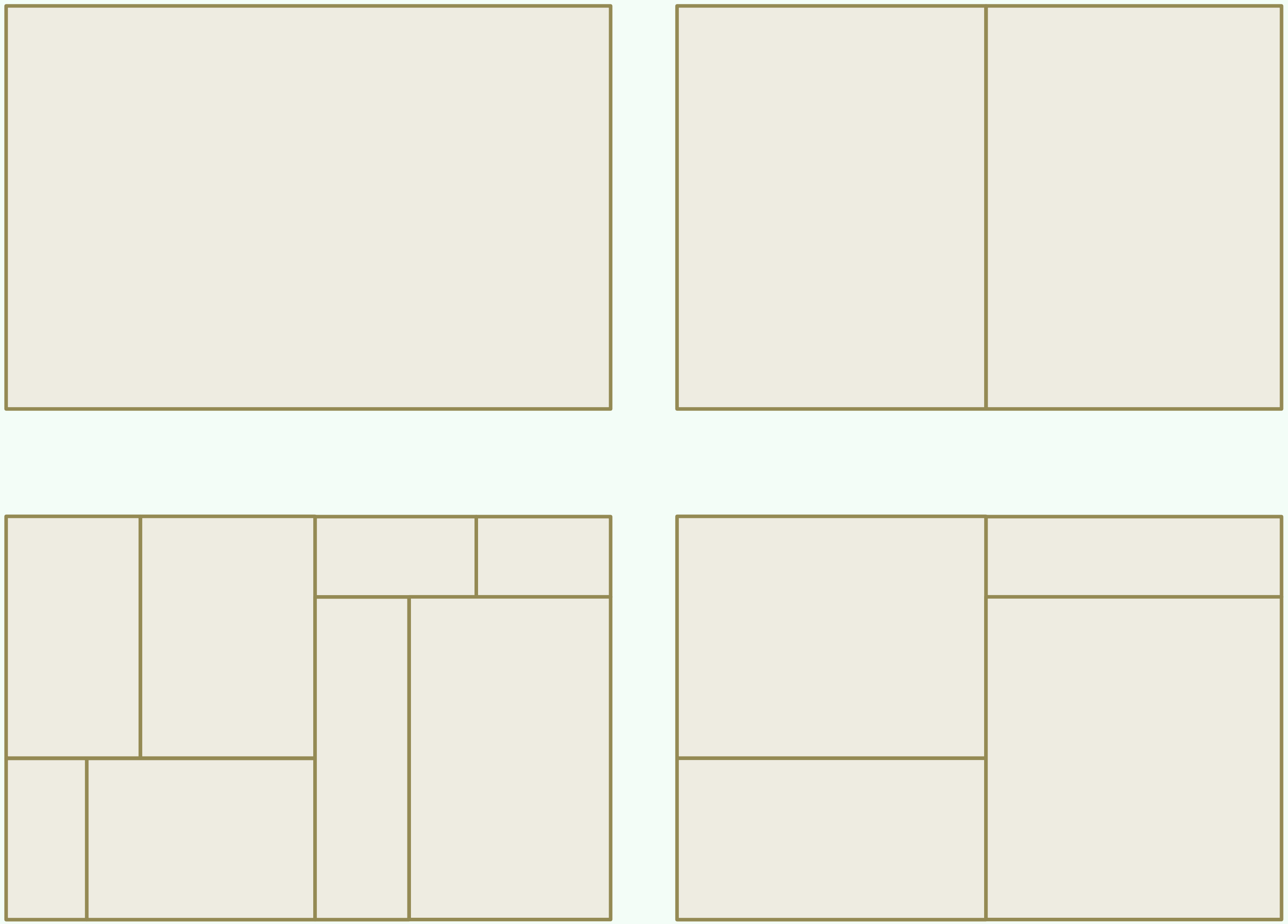
邓俊辉
deng@tsinghua.edu.cn

Preprocessing

❖ T(n)

= 2*T(n/2) + $\mathcal{O}$(n)

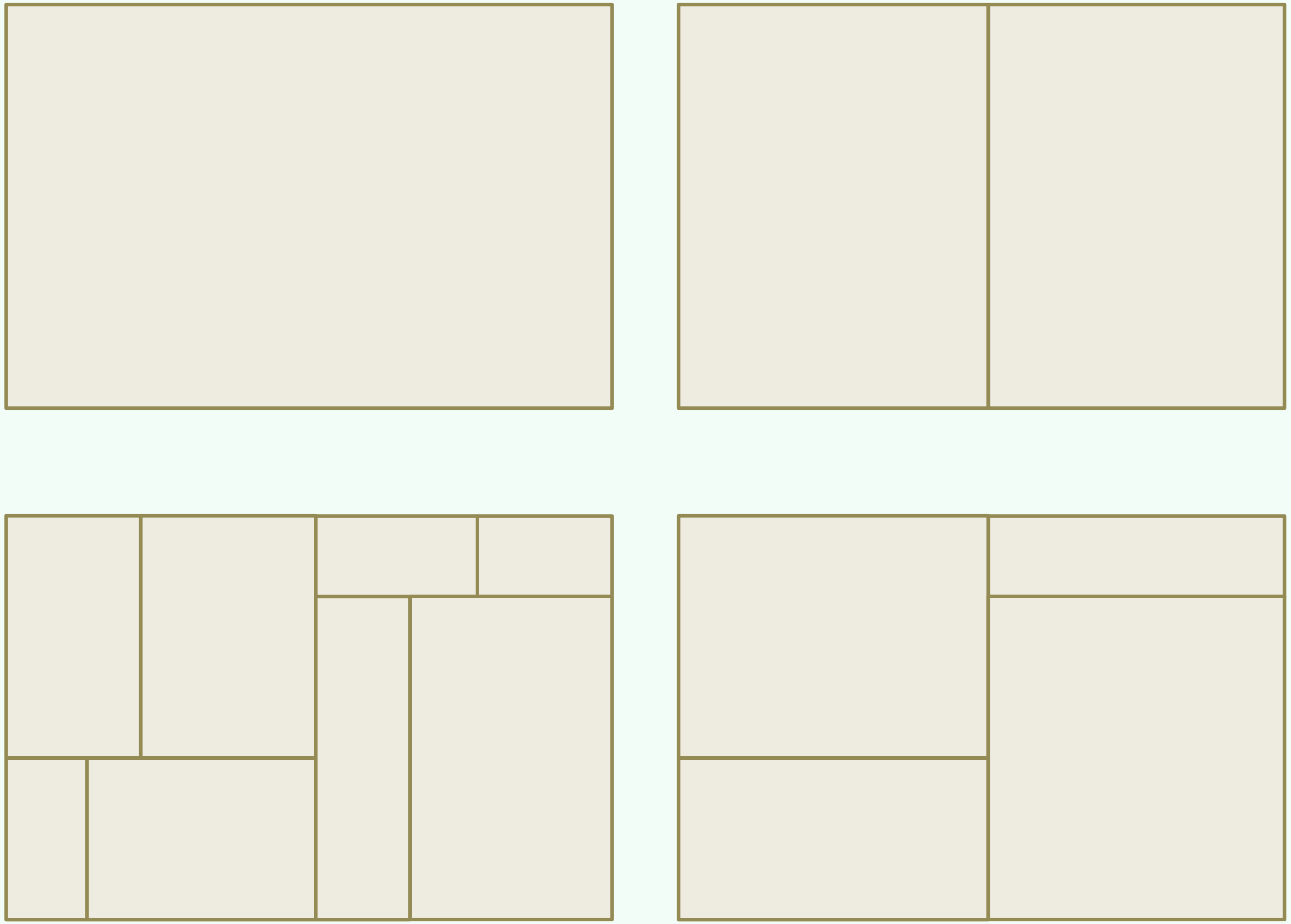
= $\mathcal{O}$(nlogn)



Storage

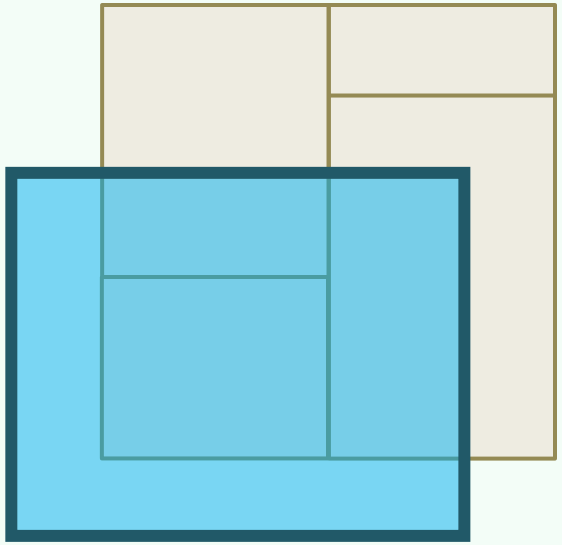
- The tree has a height of $\mathcal{O}(\log n)$

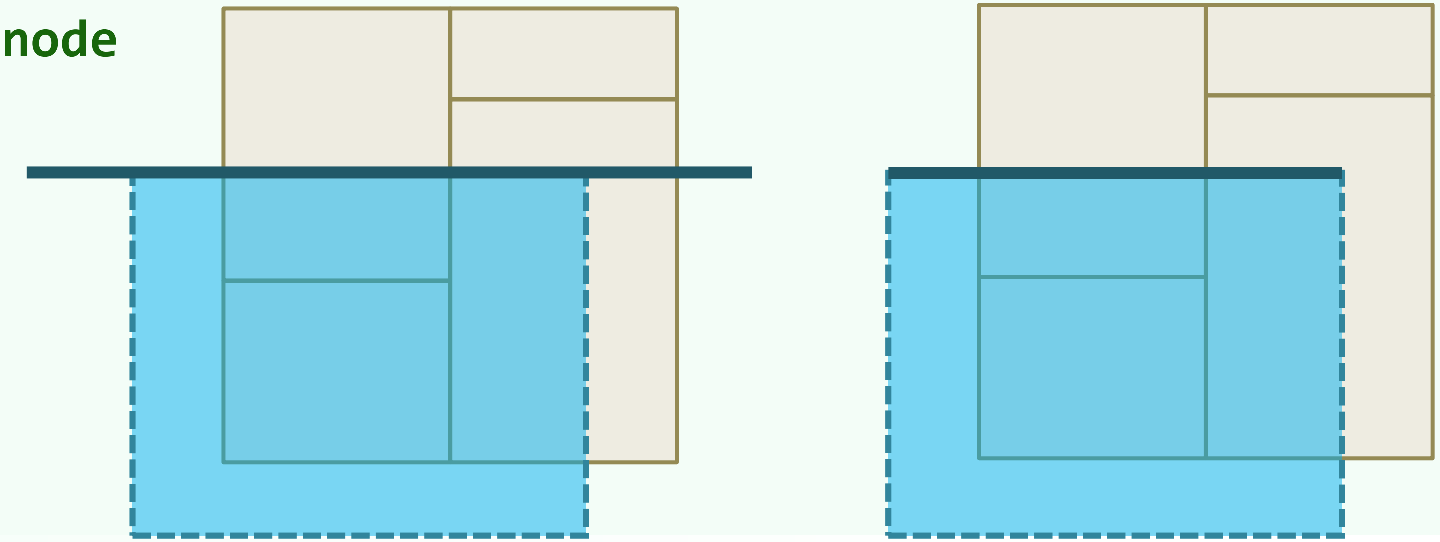
- $$\begin{aligned} & \quad 1 \\ + & \quad 2 \\ + & \quad 4 \\ + & \quad \ldots \\ + & \ \mathcal{O}(2^{\log n}) \\ = & \ \mathcal{O}(n) \end{aligned}$$



Query Time

- Claim: Report + Search = $\mathcal{O}(r + \sqrt{n})$
- The searching time depends on Q(n), the number of
  - recursive calls, or
  - sub-regions intersecting with R (at all levels)
- No more than 2 of the 4 grandchildren of each node will recurse
  - $Q(1) = \mathcal{O}(1)$
  - $Q(n) = 2 \cdot Q(n/4) + \mathcal{O}(1)$
- Solve to $Q(n) = \mathcal{O}(\sqrt{n})$





Beyond 2D

- Can 2d-tree be extended to kd-tree and help **HIGHER** dimensional GRS?

  If yes, how efficiently can it help?

- A kd-tree in k-dimensional space is constructed by

  recursively divide $\mathcal{E}^d$ along the $\boxed{1^{st}, 2^{nd}, ..., k^{th}}$ dimensions

- An orthogonal range query on a set of n points in $\mathcal{E}^d$

  - can be answered in $\mathcal{O}(r + n^{1-1/d})$ time,

  - using a kd-tree of size $\mathcal{O}(n)$, which

  - can be constructed in $\mathcal{O}(n \log n)$ time

07-C

BST Application

Multi-Level Search Tree

几株不知名的树，已脱下了黄叶

只有那两三片，多么可怜在枝上抖怯

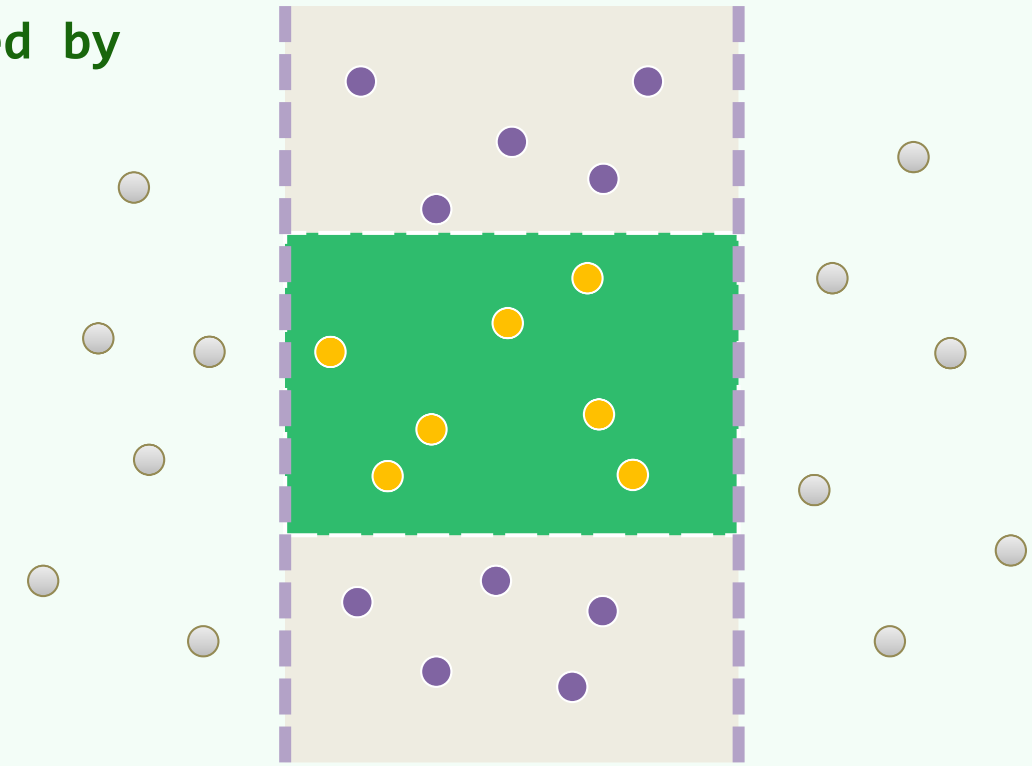
它们感到秋来到，要与世间离别

邓俊辉

deng@tsinghua.edu.cn

2D Range Query = x-Query + y-Query

- Is there any structure which answers range query **FASTER** than kd-trees?
- An m-D orthogonal range query can be answered by the **INTERSECTION** of m 1D queries
- For example, a 2D range query can be divided into two 1D range queries:
  - find all points in $[x_1, x_2]$; and then
  - find in these candidates those lying in $[y_1, y_2]$



Worst Cases

❖ Using kd-trees needs $\mathcal{O}(1+\sqrt{n})$ time. But here ...

❖ The x-query returns

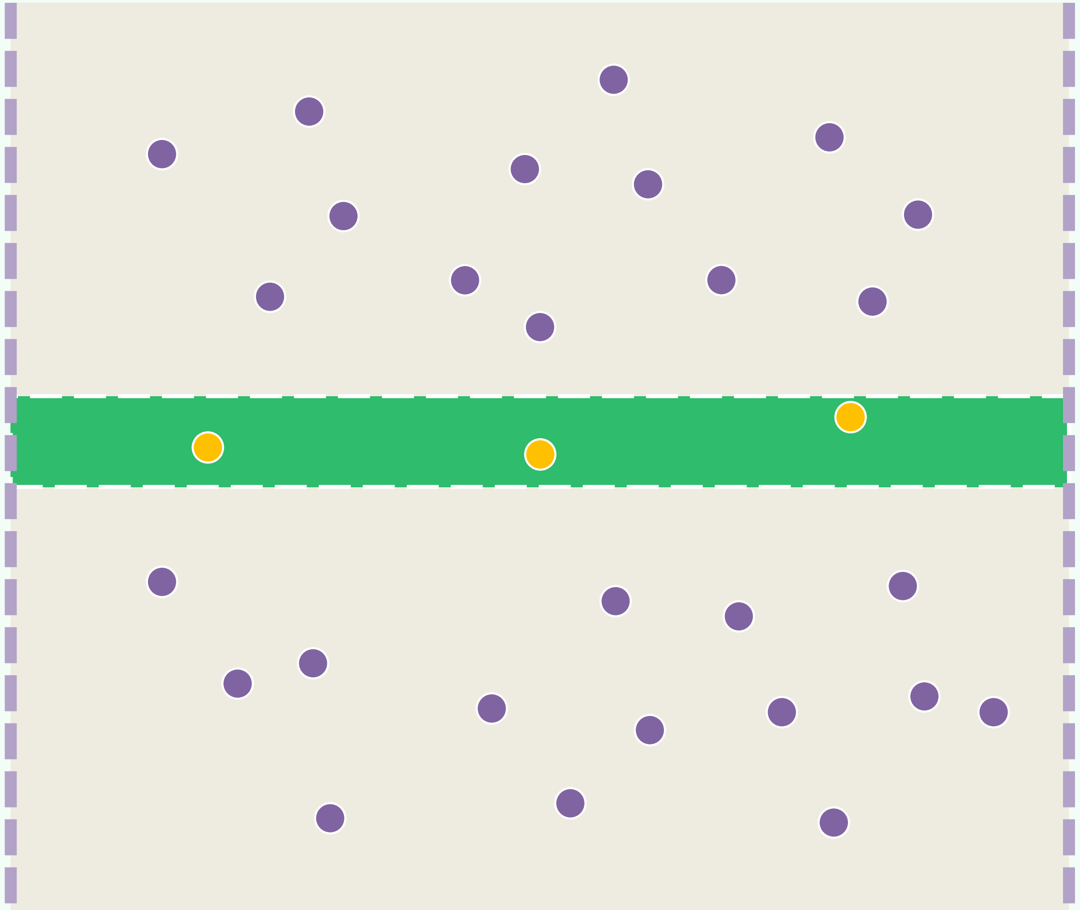
(almost) all points whereas

the y-query rejects

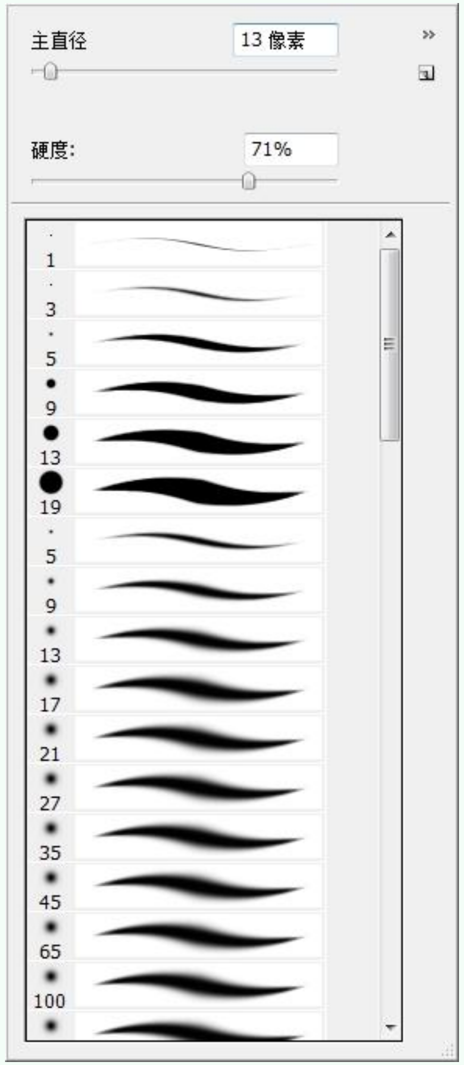
(almost) all

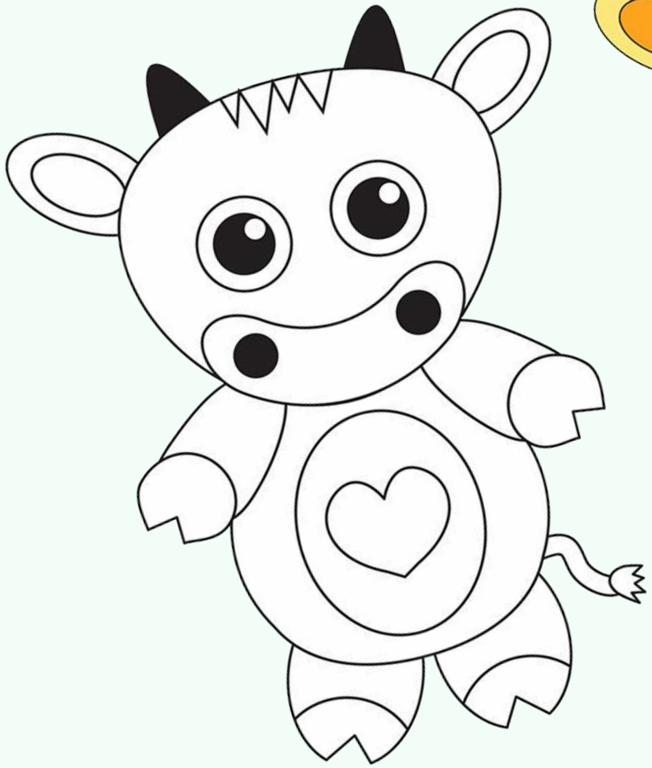
❖ We spent Ω(n) time

before getting r = 0 points



Painters' Strategy





2D Range Query = x-Query * y-Query

❖ Tree of trees

- build a 1D BBST (called x-tree)

  for the first range query (x-query);

- for each node v in the x-range tree,

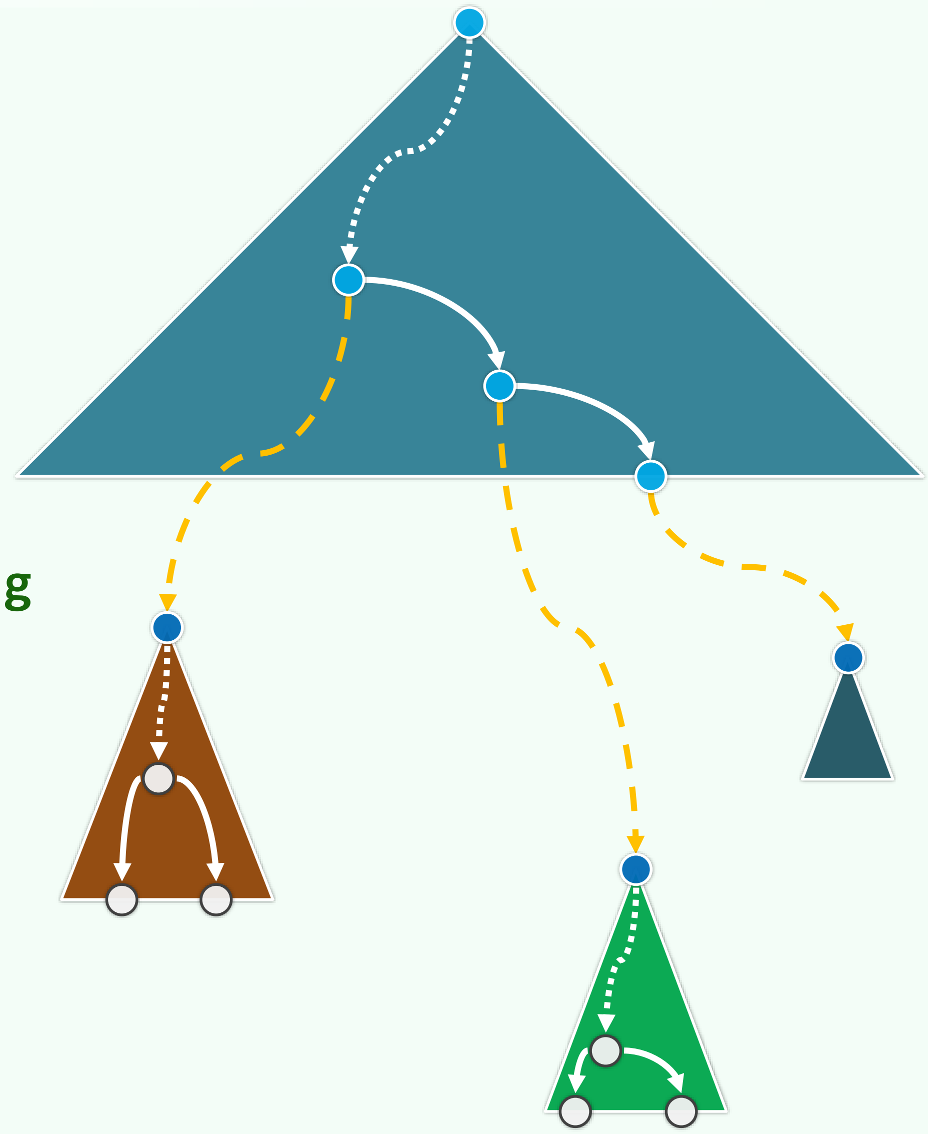
  build a y-coordinate BBST (y-tree), containing

  the canonical subset associate with v

❖ It's an x-tree of (a number of) y-trees,

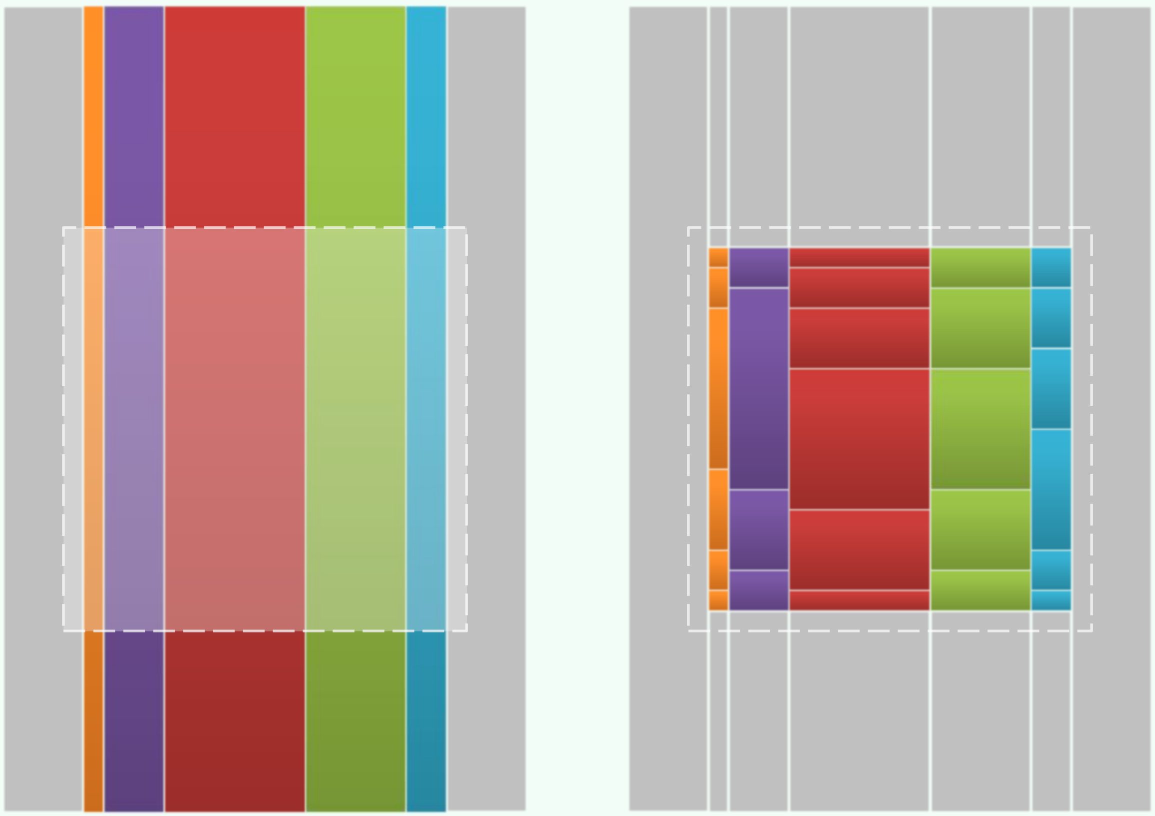
  called a Multi-Level Search Tree

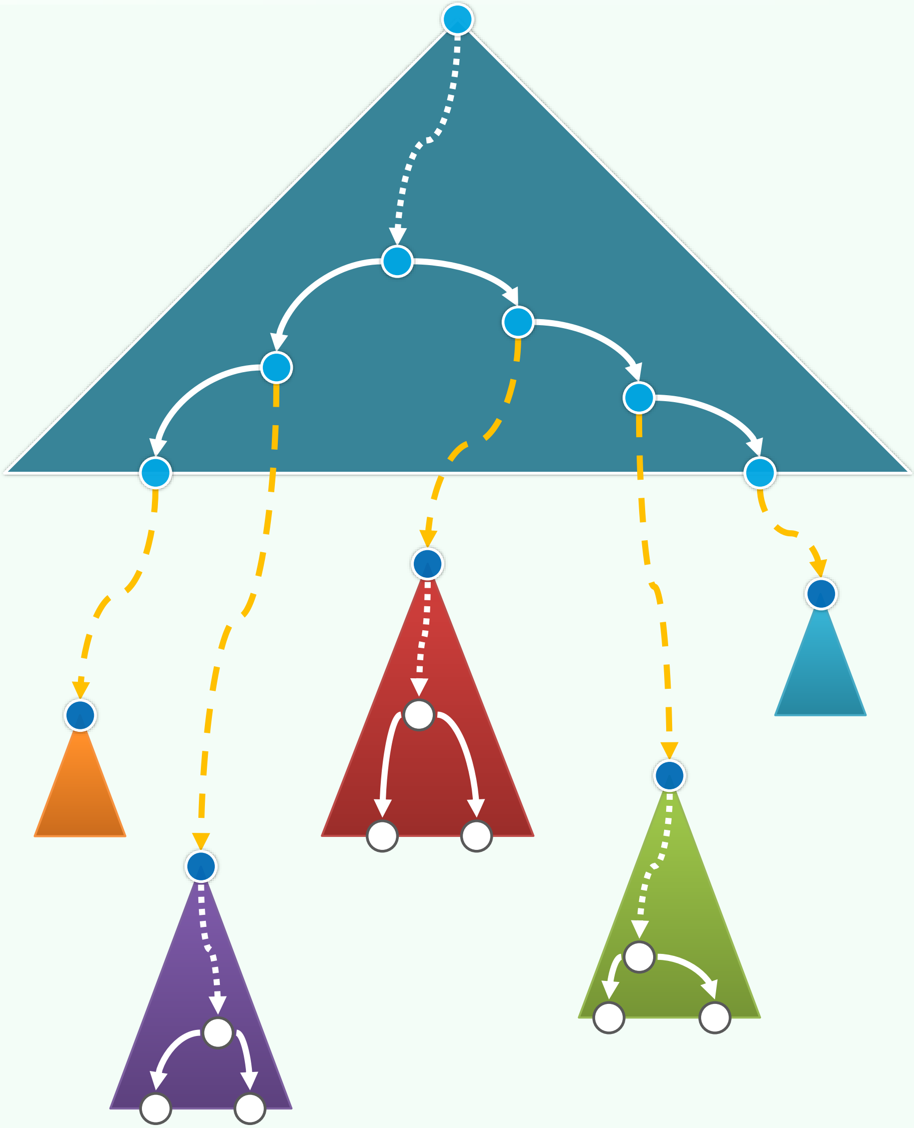
❖ How to answer range queries with such an MLST?



2D Range Query = x-Query * y-Queries

❖ **Query Time** = $\mathcal{O}(r + \log^2 n)$ ~ $\mathcal{O}(r + \log n)$





Query Algorithm

1. **Determine the canonical subsets of points that satisfy the first query**

   // there will be $\mathcal{O}(\log n)$ such canonical sets,

   //   each of which is just represented as

   //   a node in the x-tree

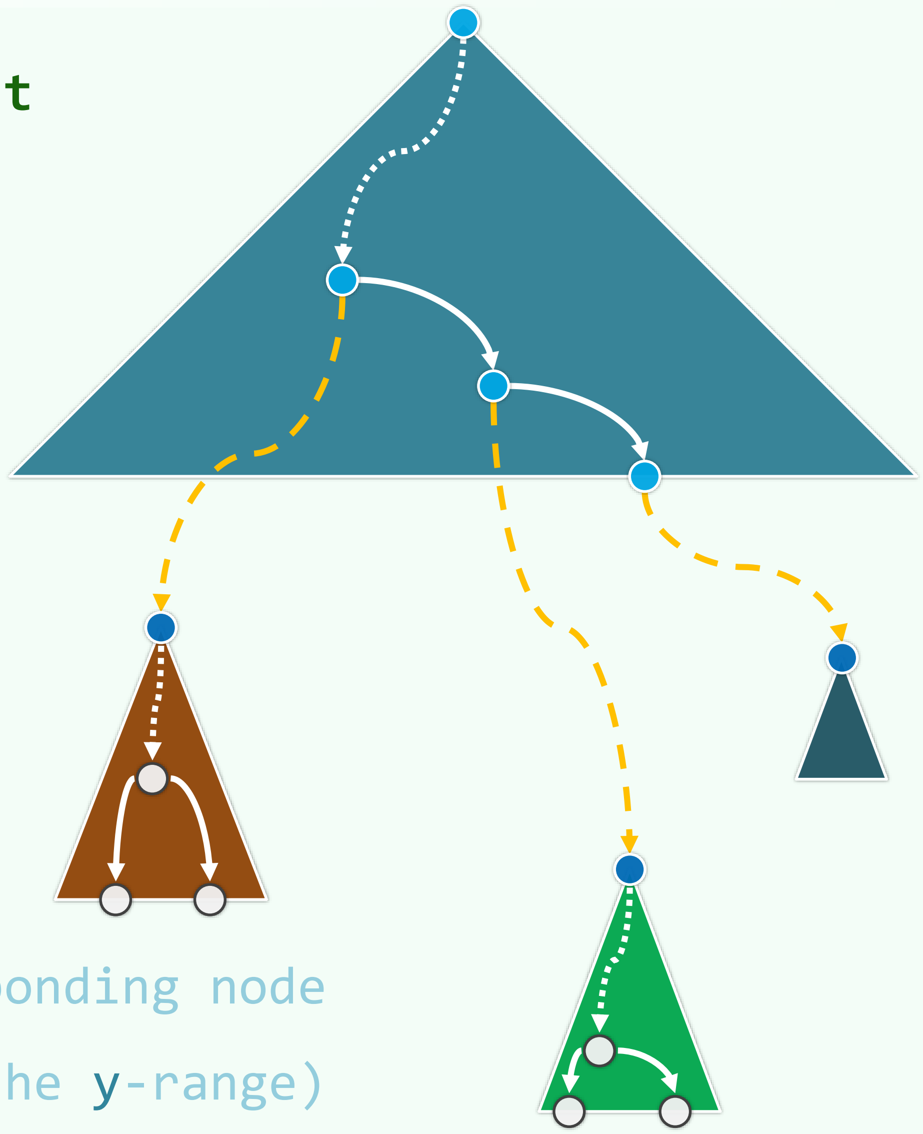
2. **Find out from each canonical subset which points lie within the y-range**

   // To do this,

   //   for each canonical subset,

   //   we access the y-tree for the corresponding node

   // this will be again a 1D range search (on the y-range)



Complexity: Preprocessing Time + Storage

❖ A 2-level search tree

for n points in the plane

can be built

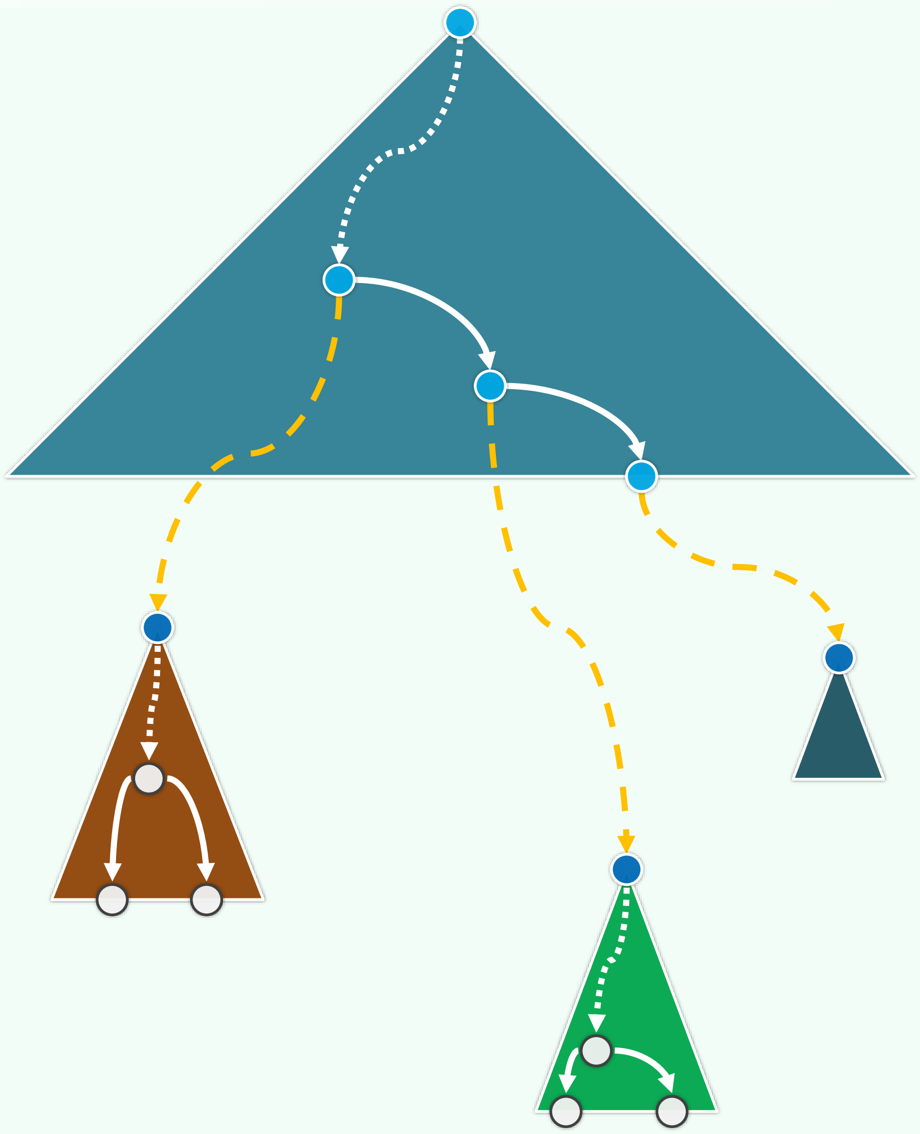
in $\mathcal{O}(n \log n)$ time

❖ Each input point is stored in $\mathcal{O}(\log n)$ y-trees

❖ A 2-level search tree

for n points in the plane

needs $\mathcal{O}(n \log n)$ space



Complexity: Query Time

❖ Claim: A 2-level search tree for n points in the plane

answers each planar range query

in $\mathcal{O}(r + \log^2 n)$ time

❖ The **x**-range query needs $\mathcal{O}(\log n)$ time

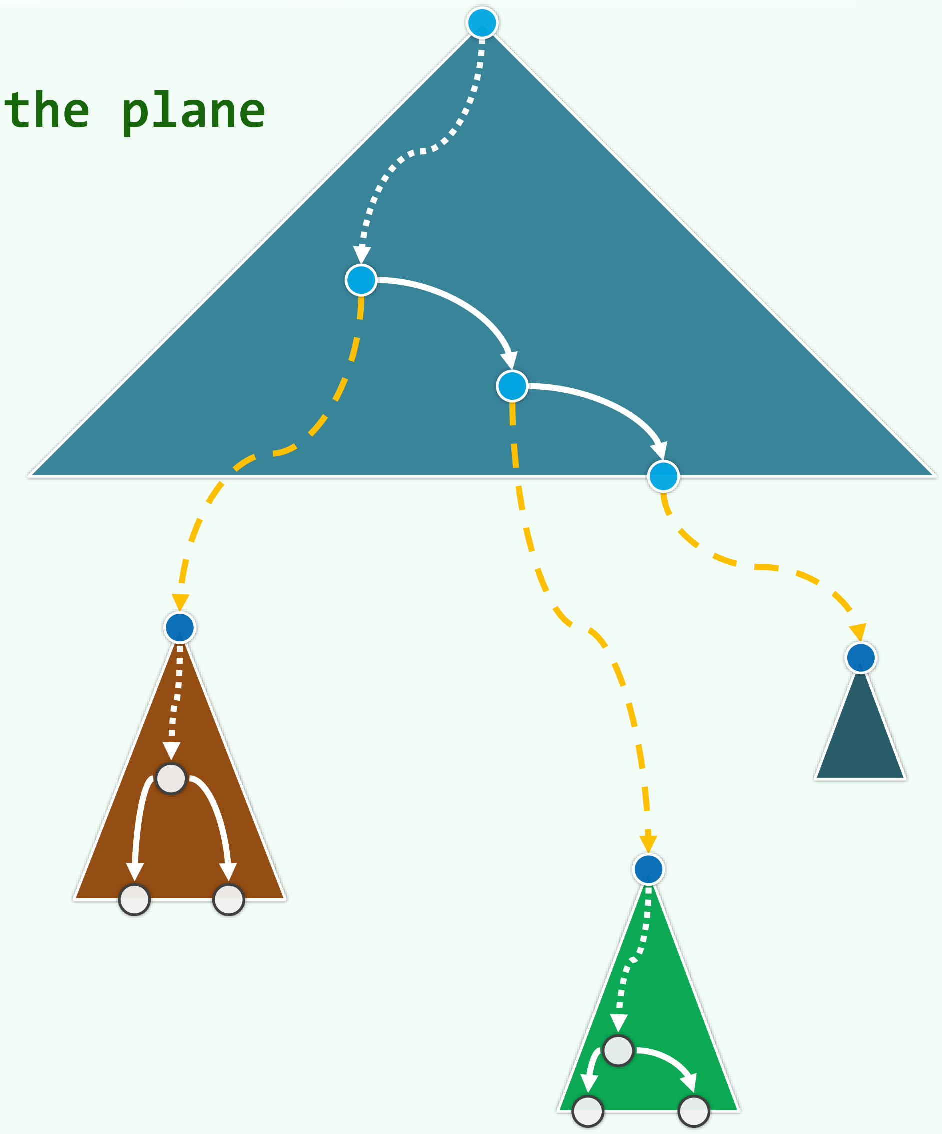
to locate the $\mathcal{O}(\log n)$ nodes

representing the canonical subsets

❖ Then for each of these nodes,

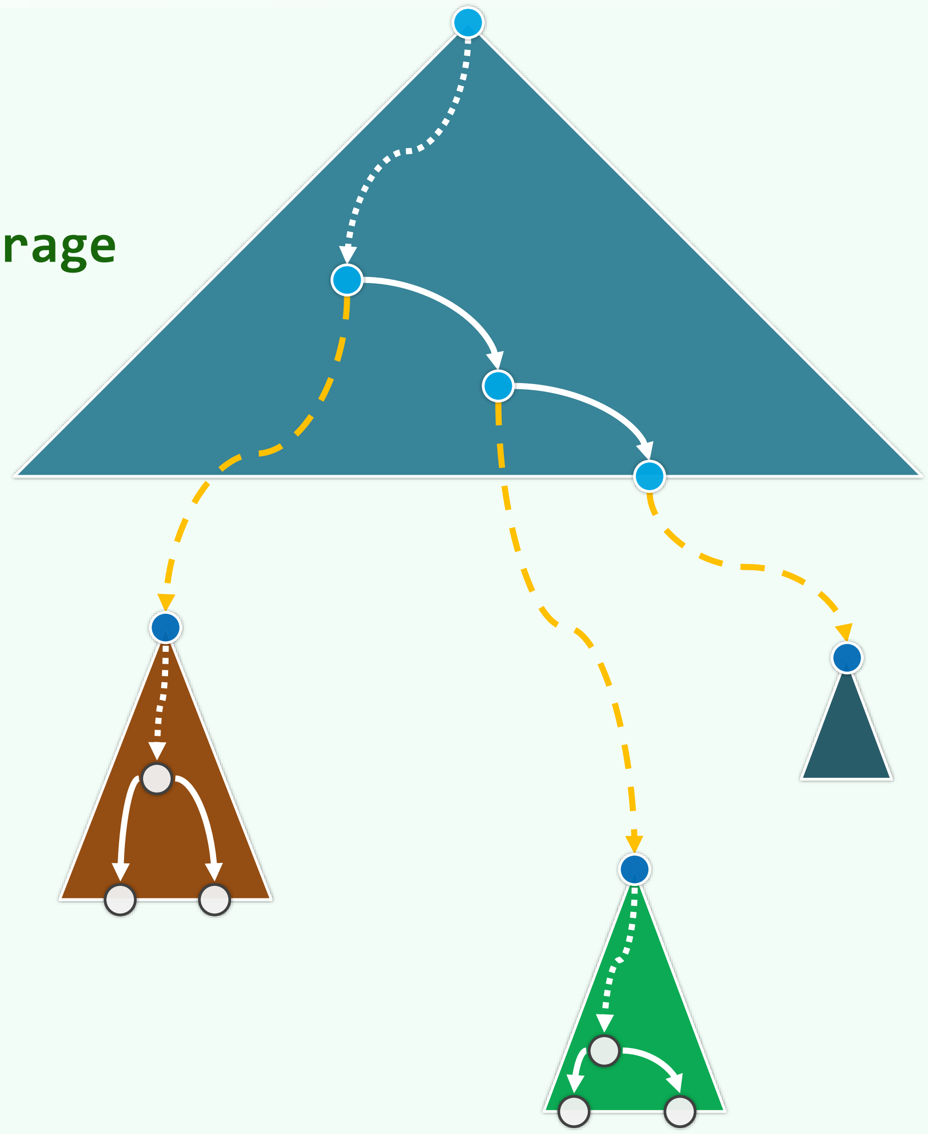
a **y**-range search is invoked,

which needs $\mathcal{O}(\log n)$ time



Beyond 2D

- Let S be a set of n points in $\mathcal{E}^d$, d ≥ 2
  - A d-level tree for S uses $\mathcal{O}(n \cdot \log^{d-1} n)$ storage
  - Such a tree can be constructed in $\mathcal{O}(n \cdot \log^{d-1} n)$ time
  - Each orthogonal range query of S can be answered in $\mathcal{O}(r + \log^{d} n)$ time
- For planar case, can the query time be improved to, say, $\mathcal{O}(\log n)$?



07-XA

BST Application

Range Tree

邓俊辉
deng@tsinghua.edu.cn

顺藤摸瓜

Coherence

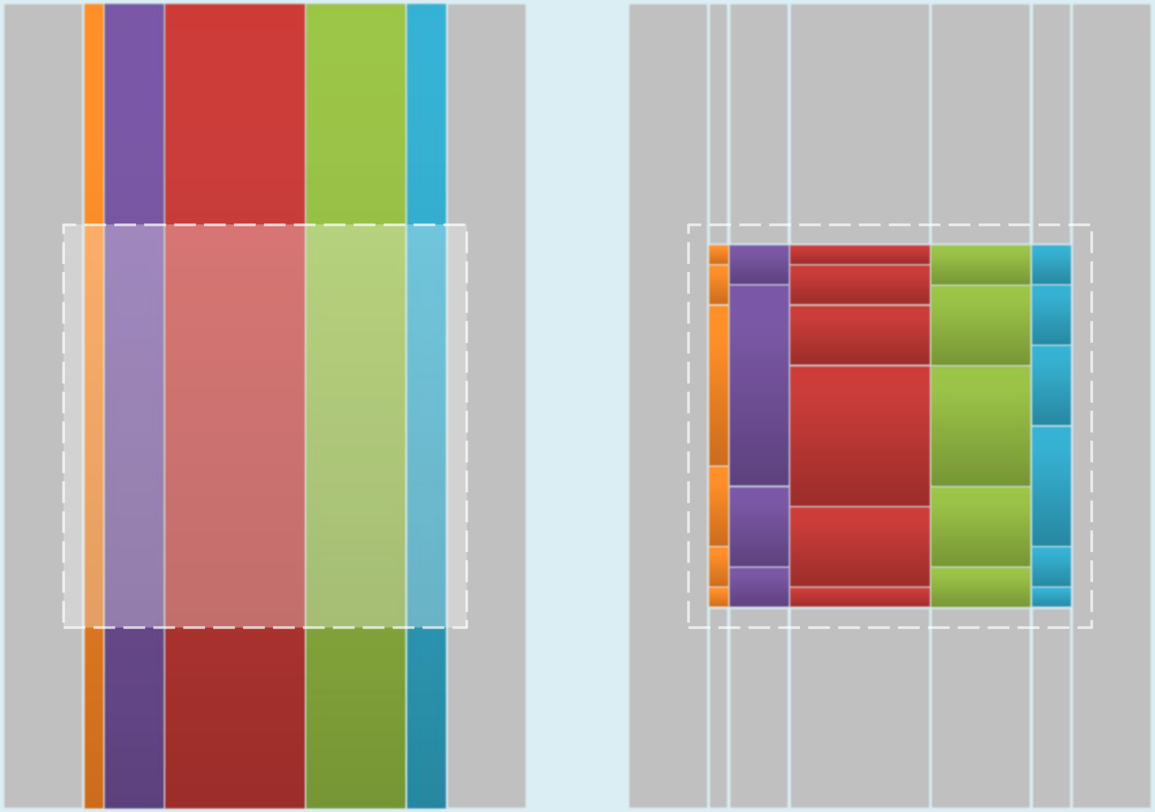
❖ For each query, we

- need to repeatedly search

  DIFFERENT y-lists,

- but always with

  the SAME key



BBST<BBST<T>> --> BBST<List<T>>

❖ Each y-search is just

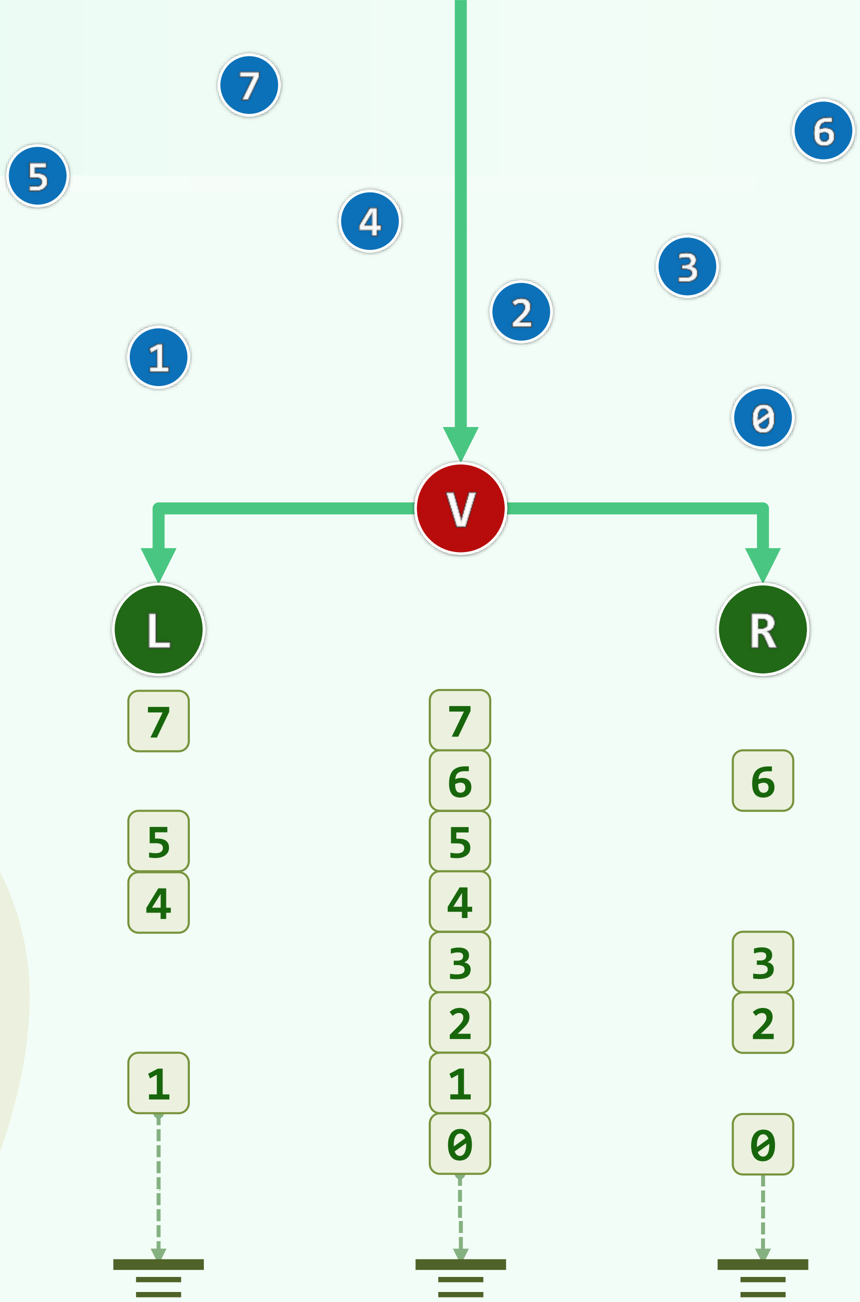
a **1D** query without further recursions

❖ So it not necessary

to store each canonical subset

as a BBST

❖ Instead, a sorted y-**list** simply works



Links Between Lists

- We can **CONNECT** all the different lists

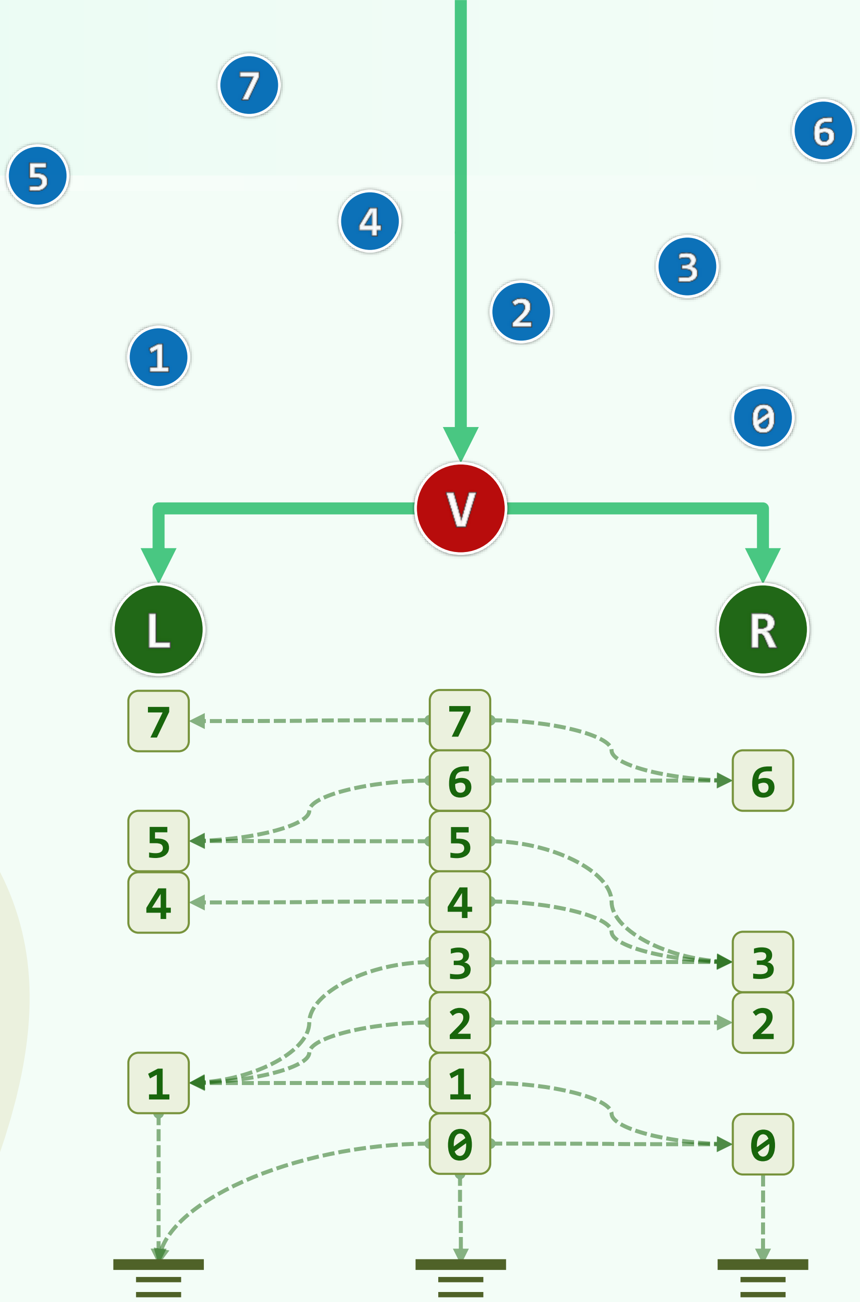
  into a **SINGLE** massive list

- Thus, once a parent y-list is searched,

  we can get, in $\mathcal{O}(1)$ time,

  the entry for child y-list by

  following the link between them



Using Coherence

- To answer a 2D range query, we will

  do an $\mathcal{O}(\log n)$ search

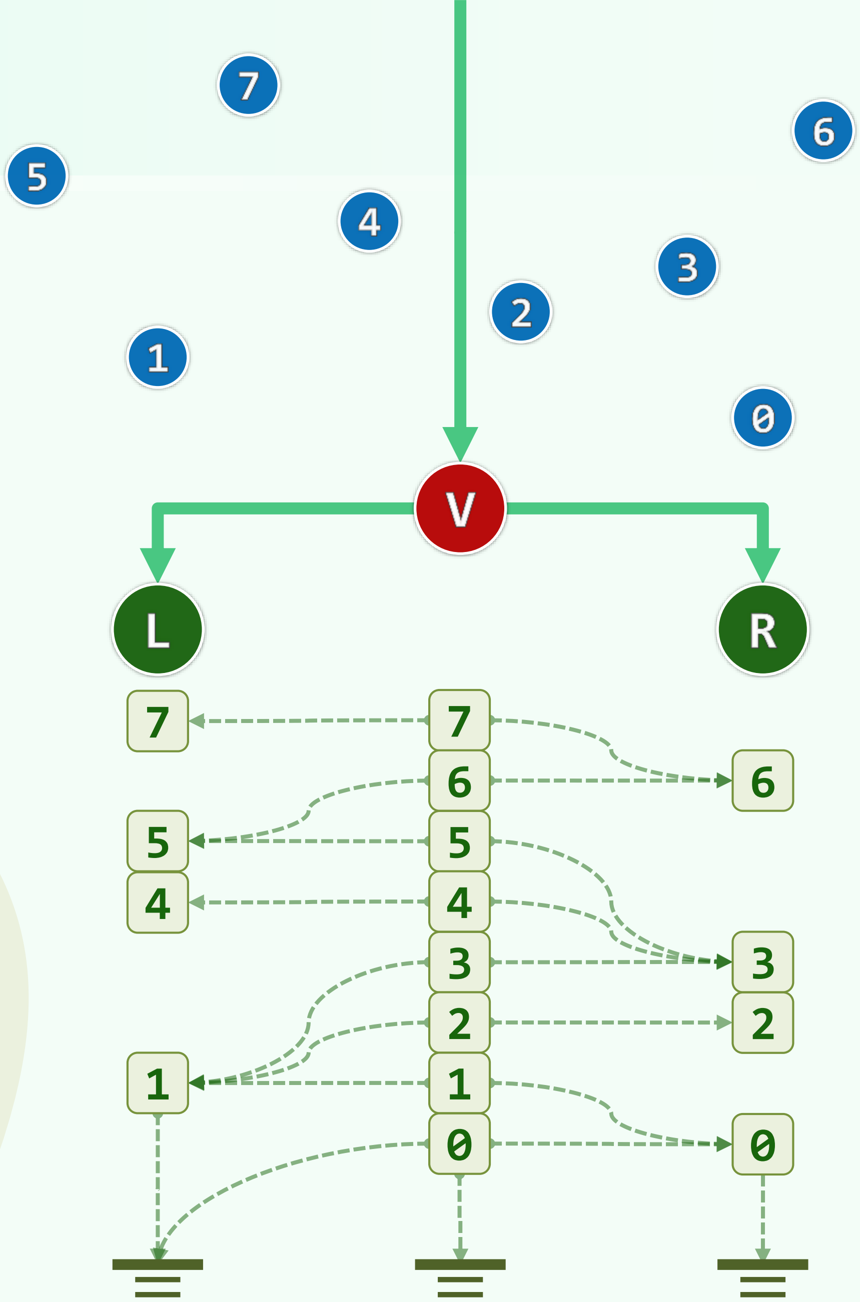
  on the y-list for the LCA

- Thereafter, while descending the x-tree, we can

  keep track of the position of y-range

  in each successive list in $\mathcal{O}(1)$ time

- This technique is called ......



Fractional Cascading

❖ For each item in A_v,

we store two pointers to

the item of NLT value

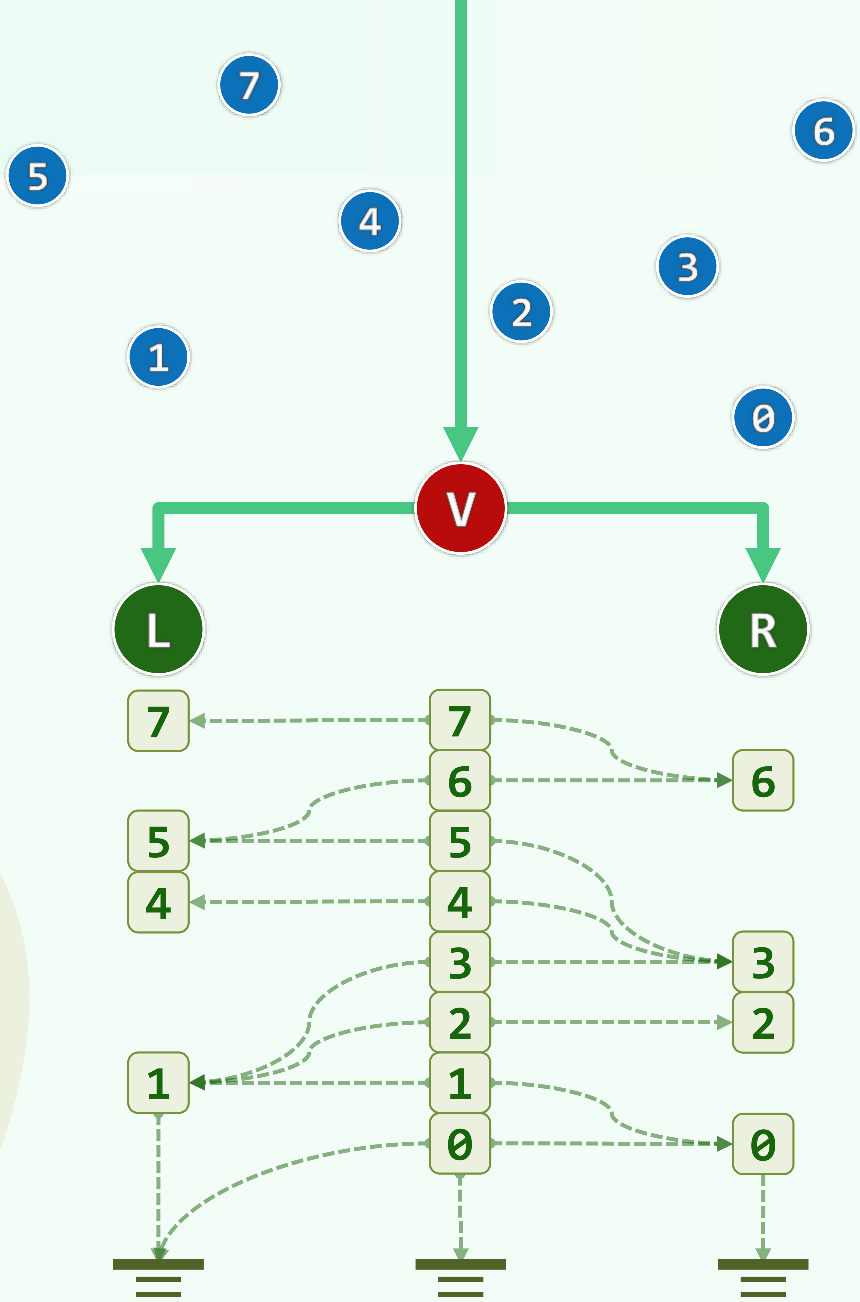
in A_L and A_R resp.

❖ Hence for any y-query with q_y,

once we know its entry in A_v, we can

determine its entry in either A_L or A_R

in $\mathcal{O}(1)$ additional time



Construction By 2-Way Merging

❖ Let V be an internal node in the **x**-tree

with L/R its left/right child resp.

❖ Let A_v be the **y**-list for v and

A_L/A_R be the **y**-lists for its children

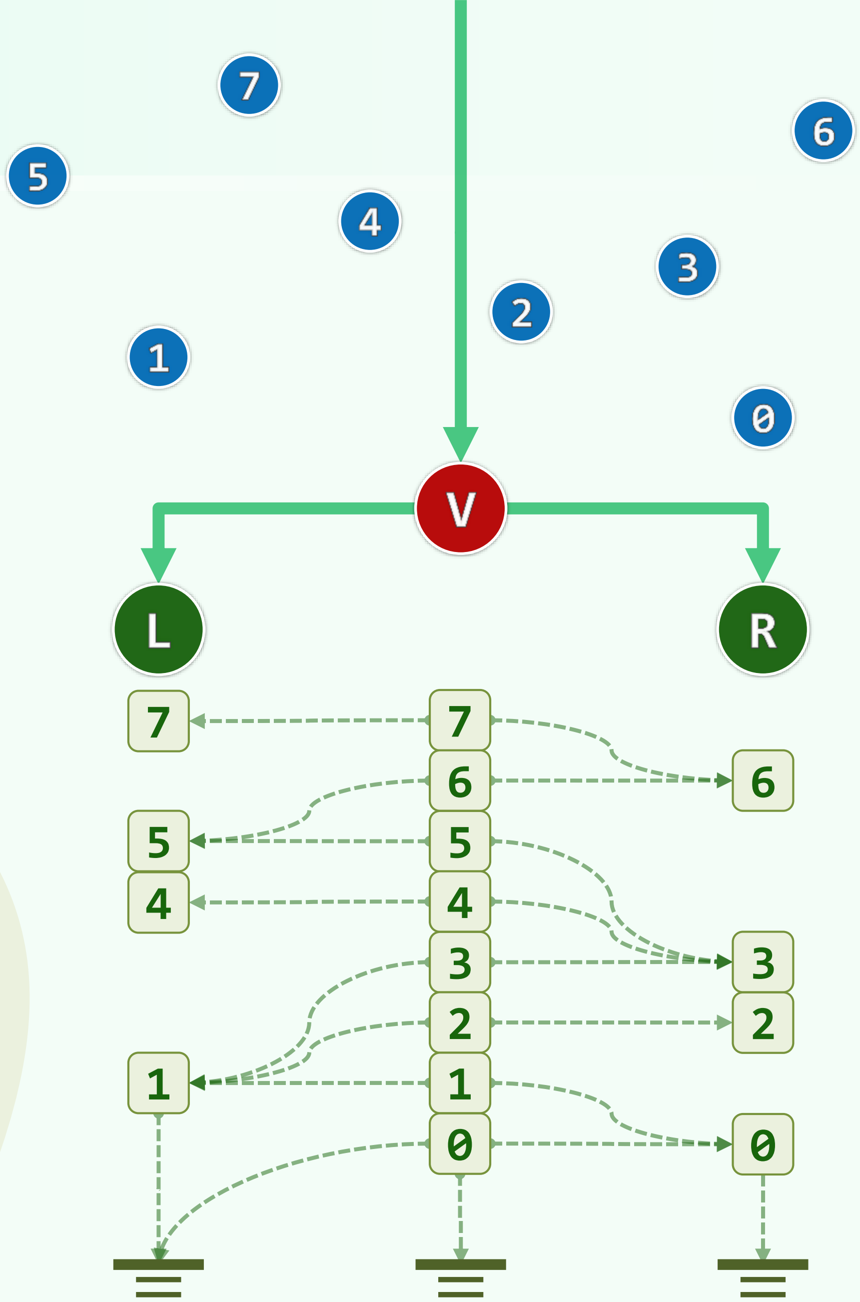
❖ Assuming no duplicate **y**-coordinates, we have

- A_v is the disjoint union

  of A_L and A_R, and hence

- A_v can be obtained by

  merging A_L and A_R (in **linear** time)



Complexity

❖ An MLST with fractional cascading

is called a **range tree**

❖ A y-search for root is done in $\mathcal{O}(\log n)$ time

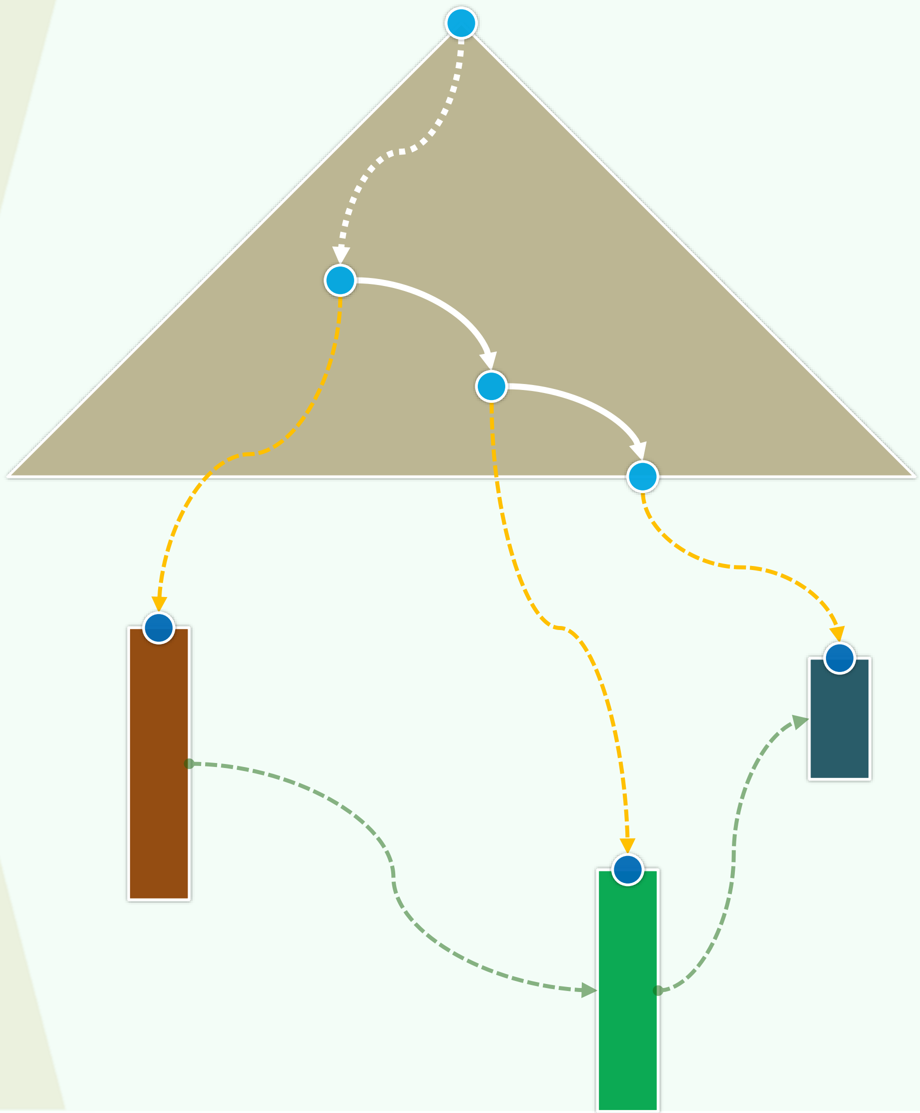
❖ To drop down to each next level, we can

get, in $\mathcal{O}(1)$ time, the current y-interval

from that of the prior level

❖ Hence, each 2D orthogonal range query

- can be answered in $\mathcal{O}(r + \log n)$ time
- from a data structure of size $\mathcal{O}(n \cdot \log n)$,
- which can be constructed in $\mathcal{O}(n \cdot \log n)$ time



Beyond 2D

- Unfortunately, the trick of fractional cascading

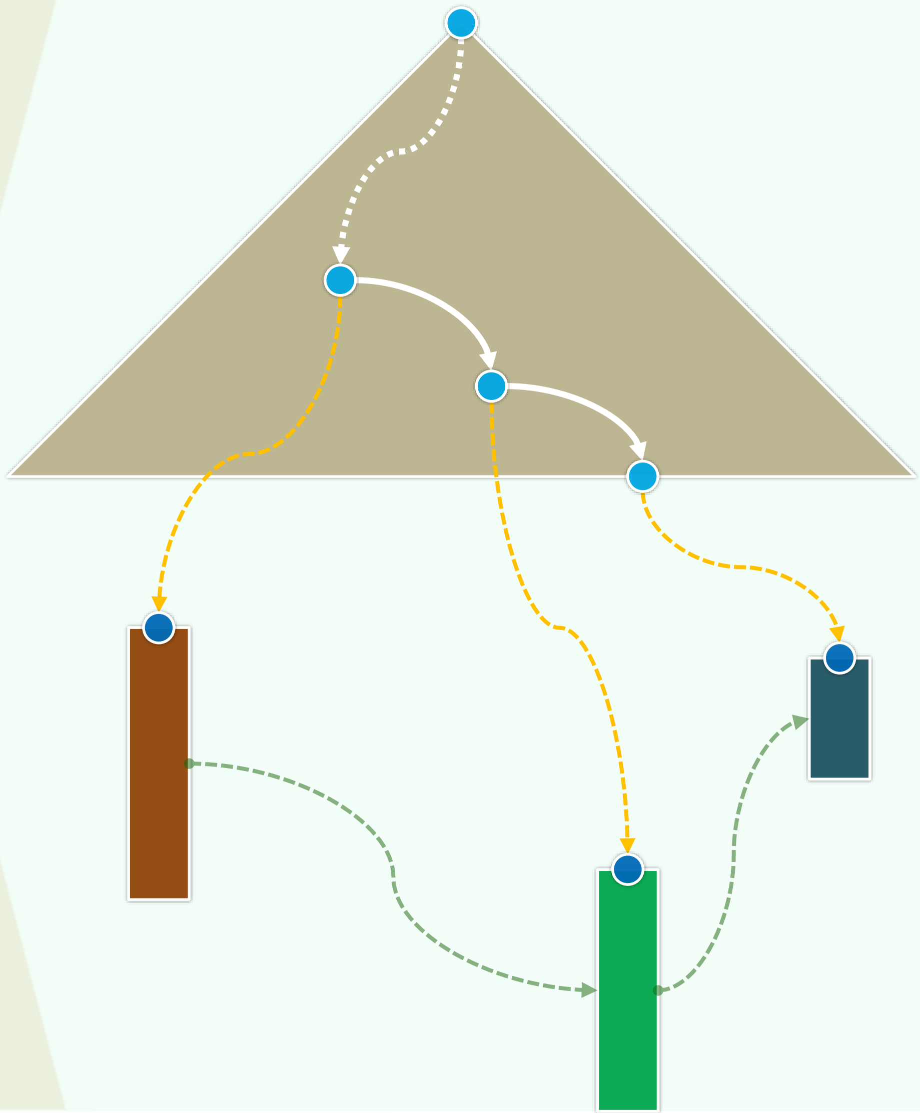
  can **ONLY** be applied to

  the **LAST** level the search structure

- Given a set of n points in $\mathcal{E}^d$,

  an orthogonal range query

  - can be answered in $\mathcal{O}(r + \log^{d-1} n)$ time
  - from a data structure of size $\mathcal{O}(n \cdot \log^{d-1} n)$,
  - which can be constructed in $\mathcal{O}(n \cdot \log^{d-1} n)$ time



07-XB

BST Application

Interval Tree

Your instinct, rather than precision stabbing, is more about just random bludgeoning.

邓俊辉
deng@tsinghua.edu.cn

Stabbing Query

❖ Given a set of intervals in general position

on the **x**-axis:

$$S = \{s_i = [x_i, x_i'] \mid 1 \leq i \leq n\}$$

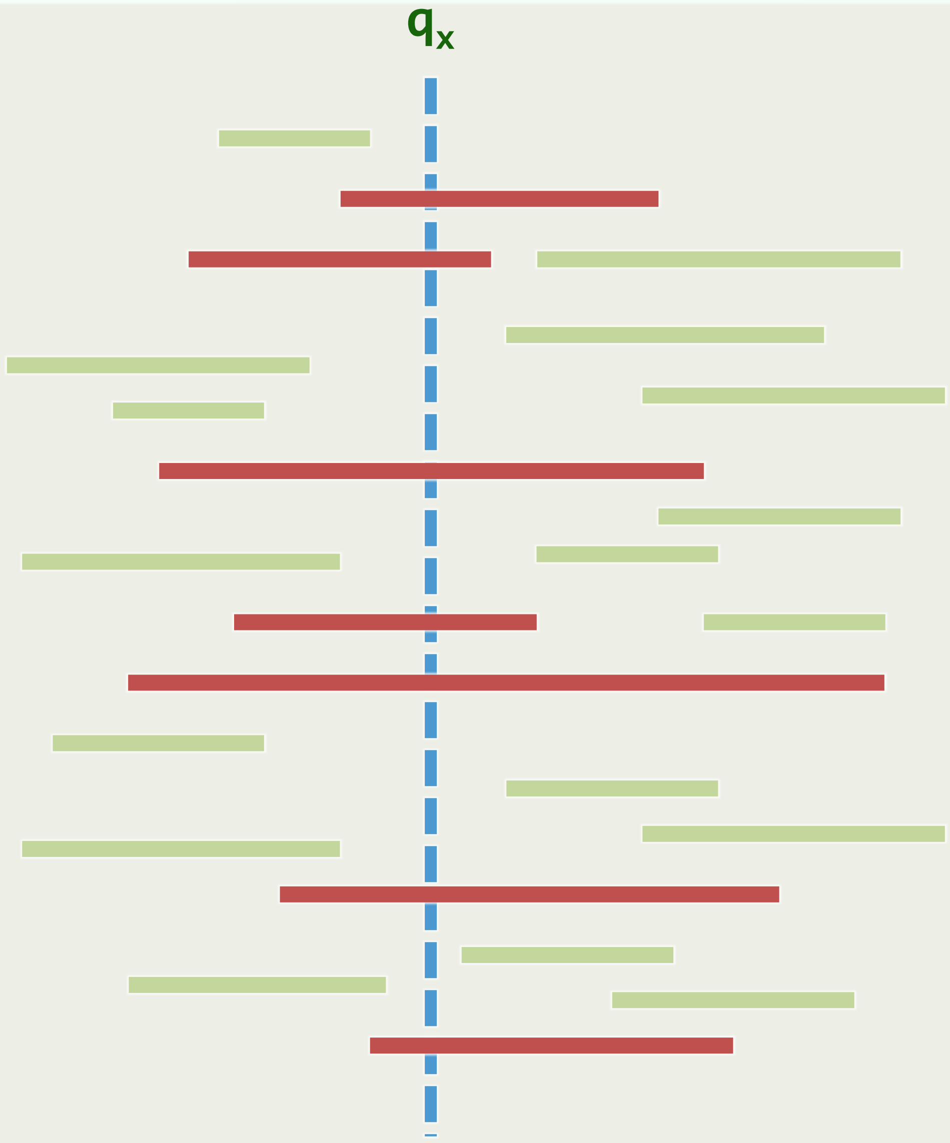
and a query point q_x

❖ Find all intervals that contain q_x

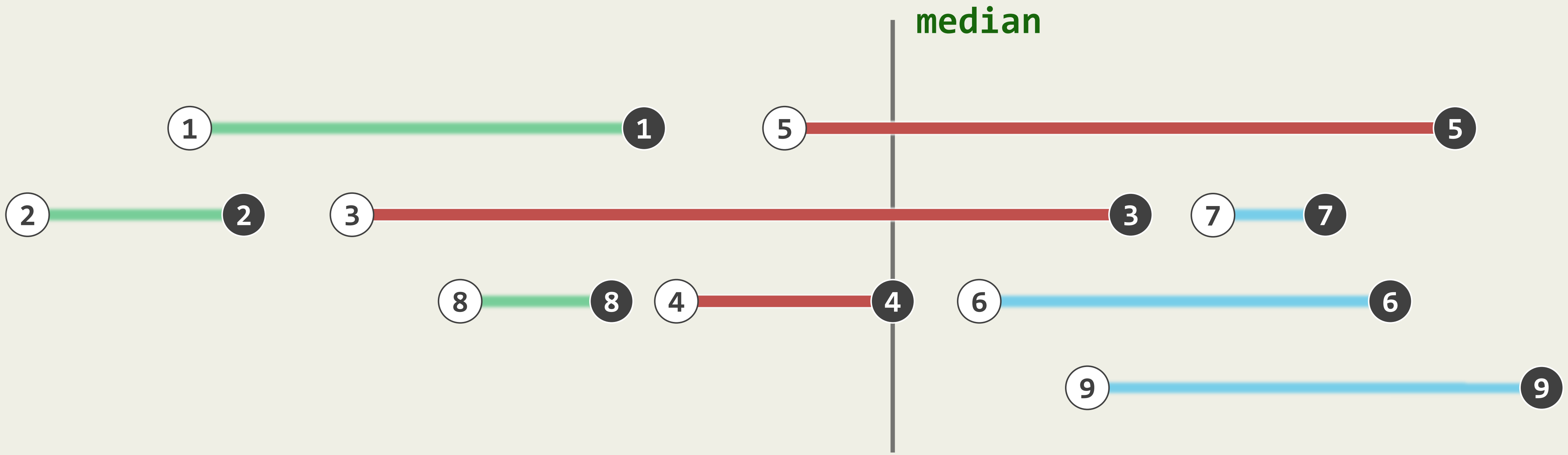
$$\{s_i = [x_i, x_i'] \mid x_i \leq q_x \leq x_i'\}$$

❖ To solve this query,

we will use the so-called interval tree ...



Median

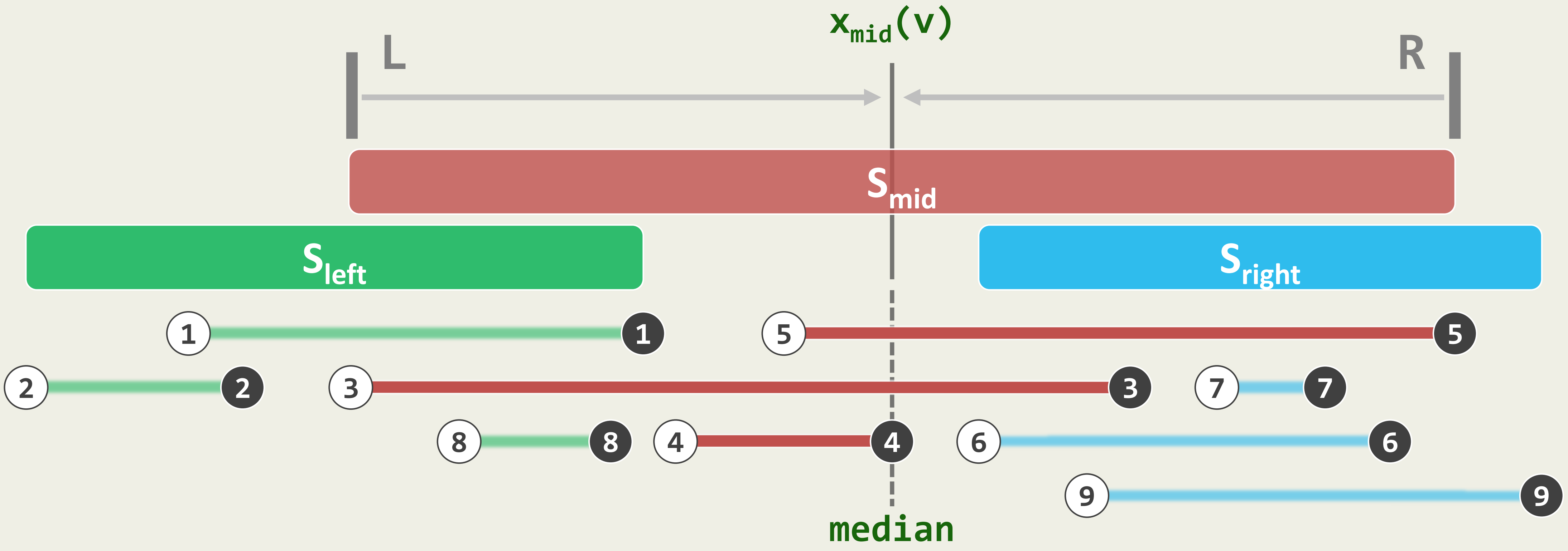


❖ Let $P = \partial S$ be the set of all endpoints

(By general position assumption, $|P| = 2n$)

❖ Let $x_{mid} = median(P)$ be the median of P

Partitioning



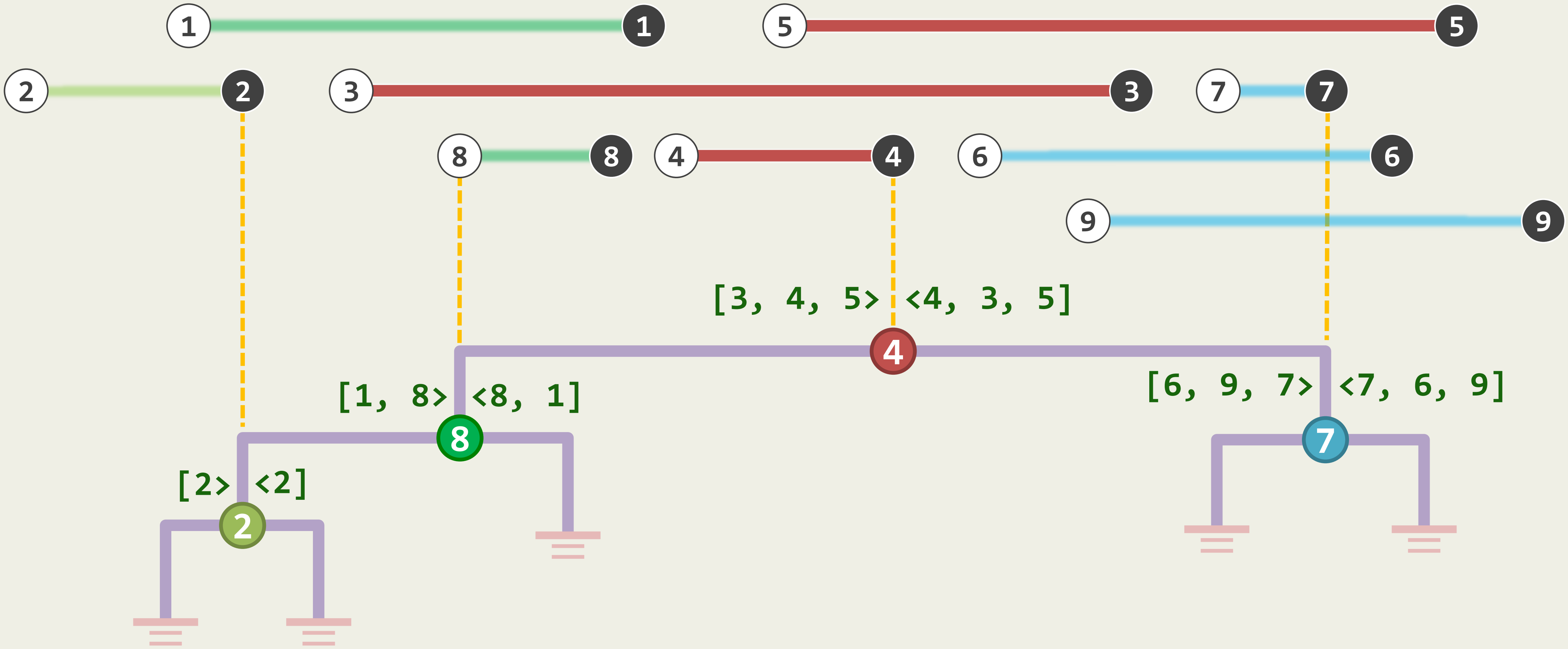
❖ All intervals can be then categorized into 3 subsets:

$$S_{left} = \{ S_i \mid x'_i < x_{mid} \} \quad S_{mid} = \{ S_i \mid x_i \le x_{mid} \le x'_i \} \quad S_{right} = \{ S_i \mid x_{mid} < x_i \}$$

❖ $S_{left/right}$ will be **recursively** partitioned until they are empty (leaves)

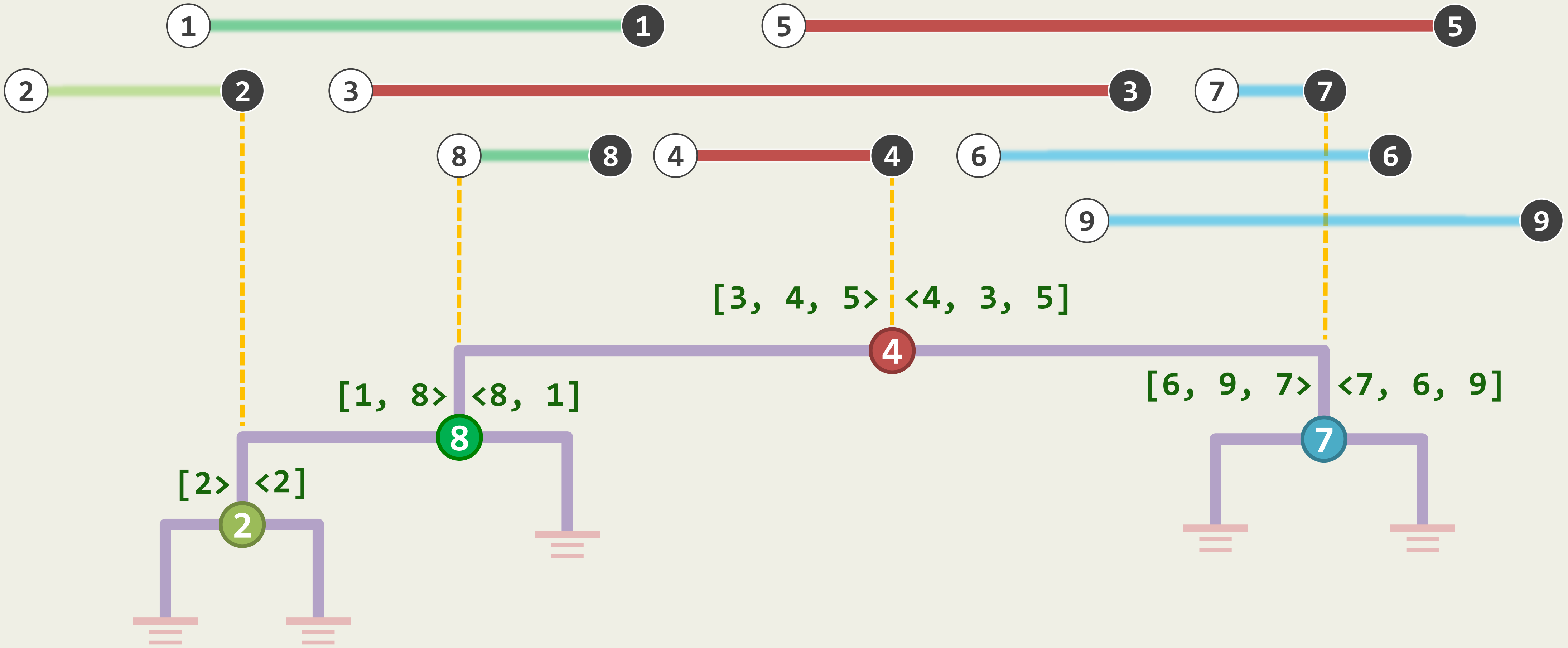
Balance & $\mathcal{O}(\log n)$ Depth

$\max\{\ |S_{left}|, |S_{right}|\ \} \le n/2$ **Best case:** $|S_{mid}| = n$ **Worst case:** $|S_{mid}| = 1$



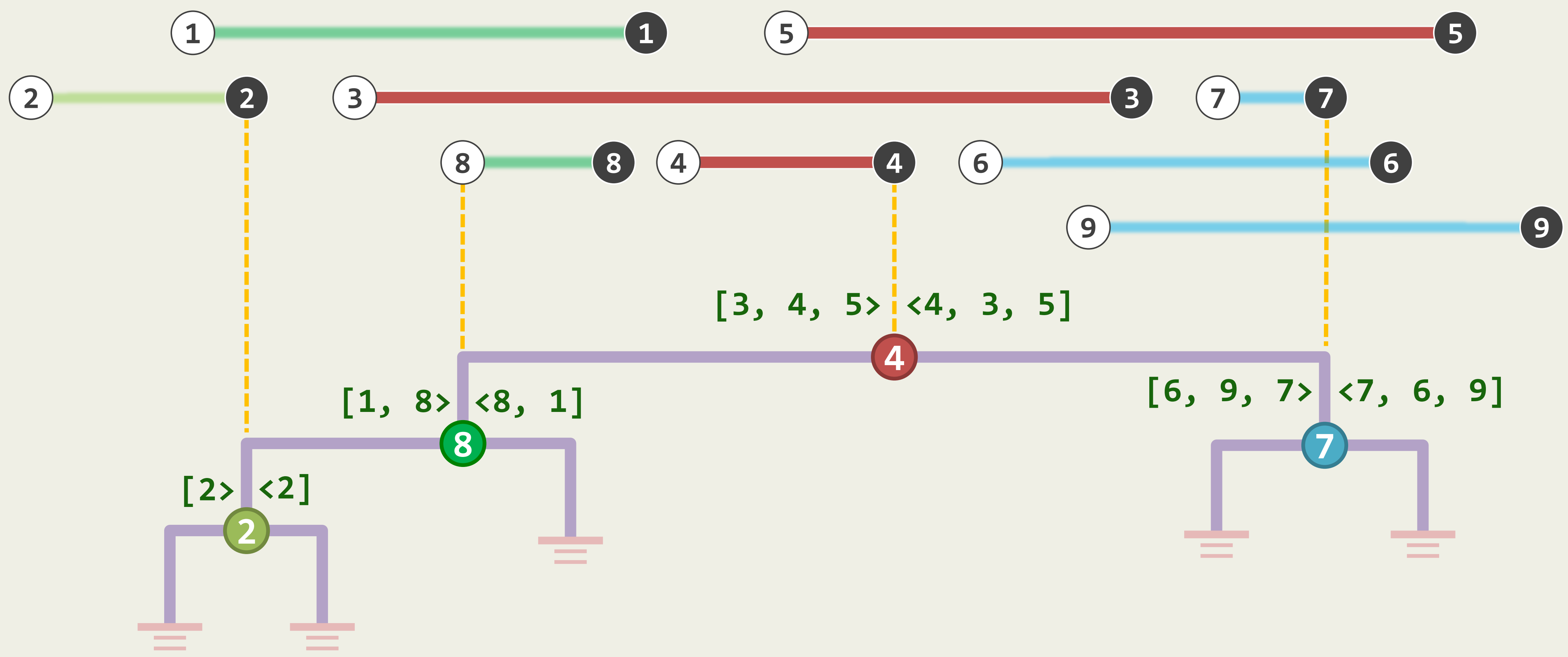
Associative Lists

- $L_{left/right}$ = all intervals of S_{mid} sorted by the left/right endpoints



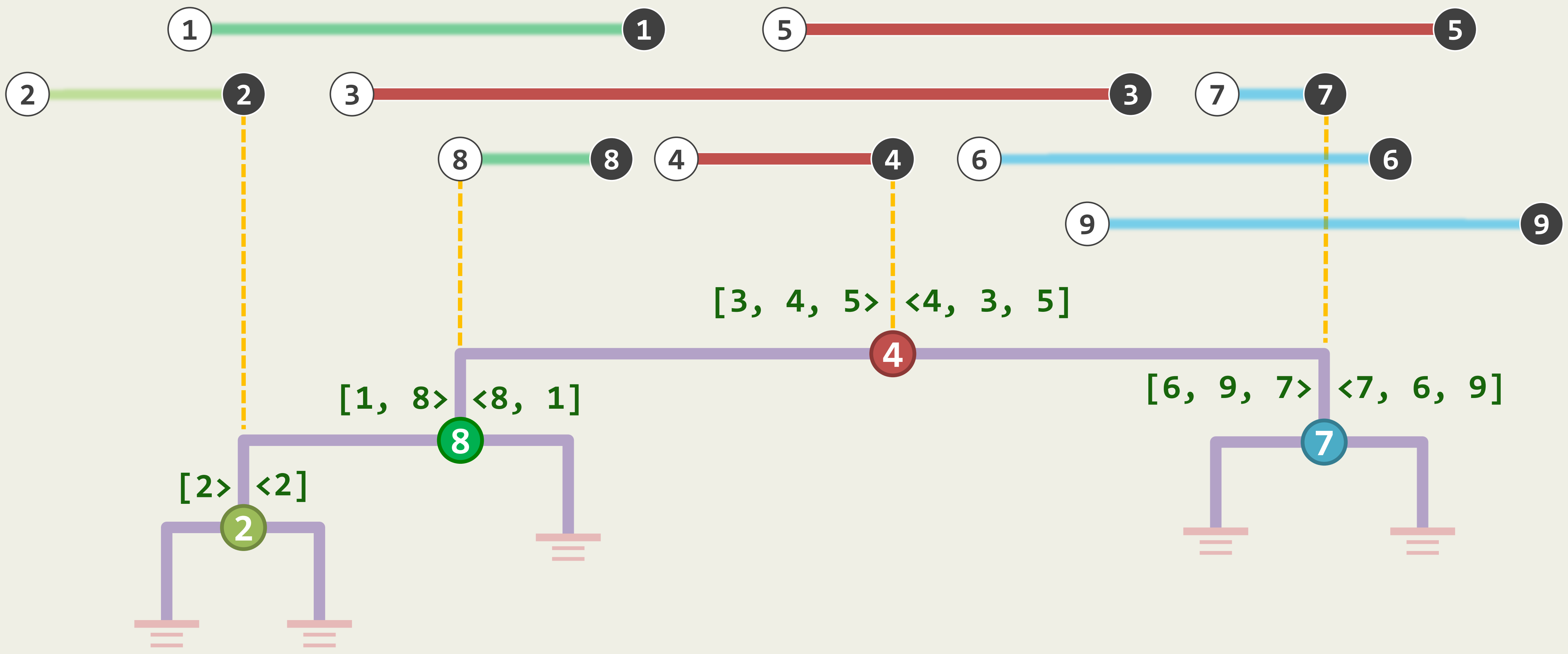
$\mathcal{O}(n)$ Size

❖ Each segment appears twice (one in each list)



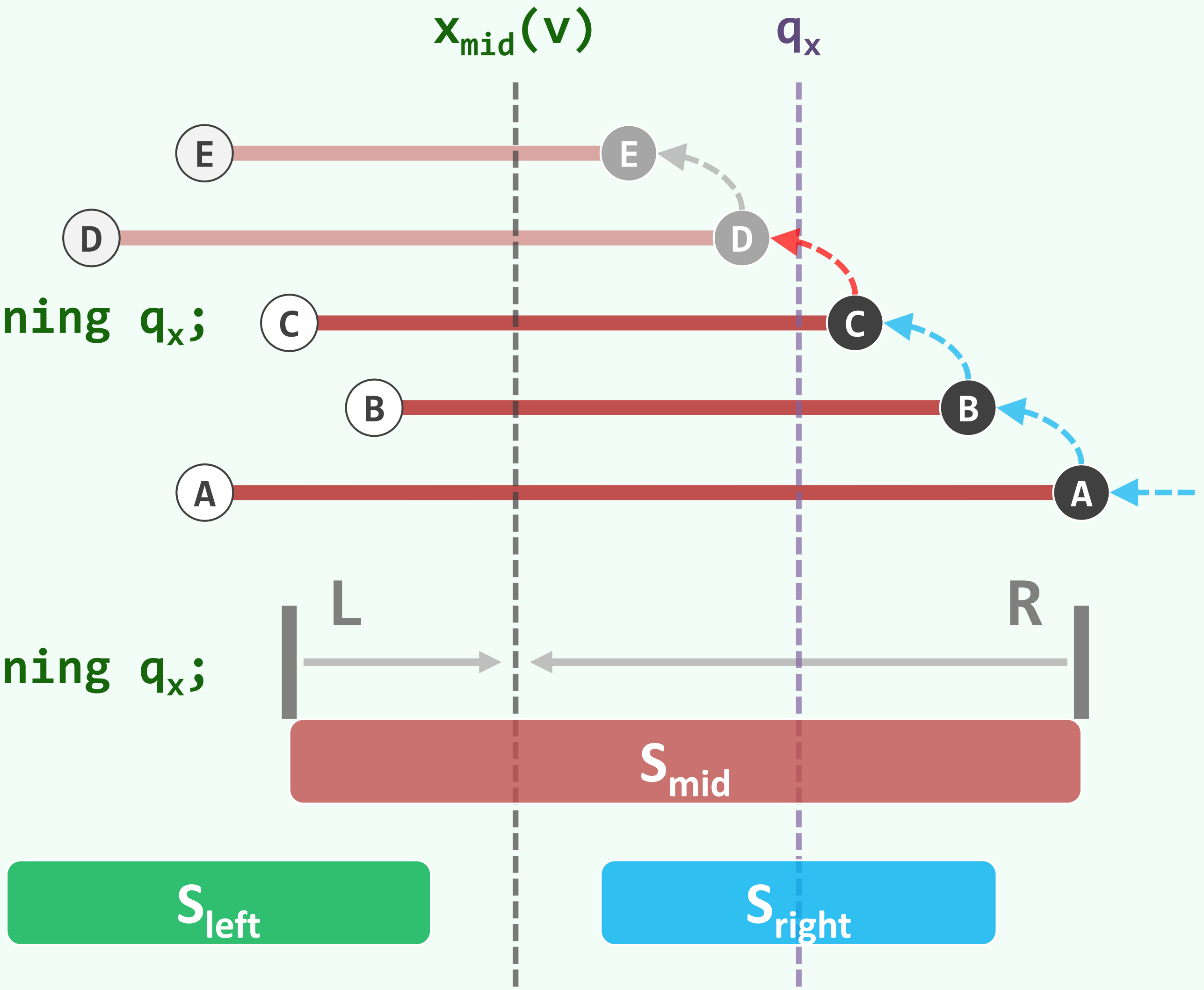
$O(nlogn)$ Construction Time

❖ Hint: avoid repeatedly sorting



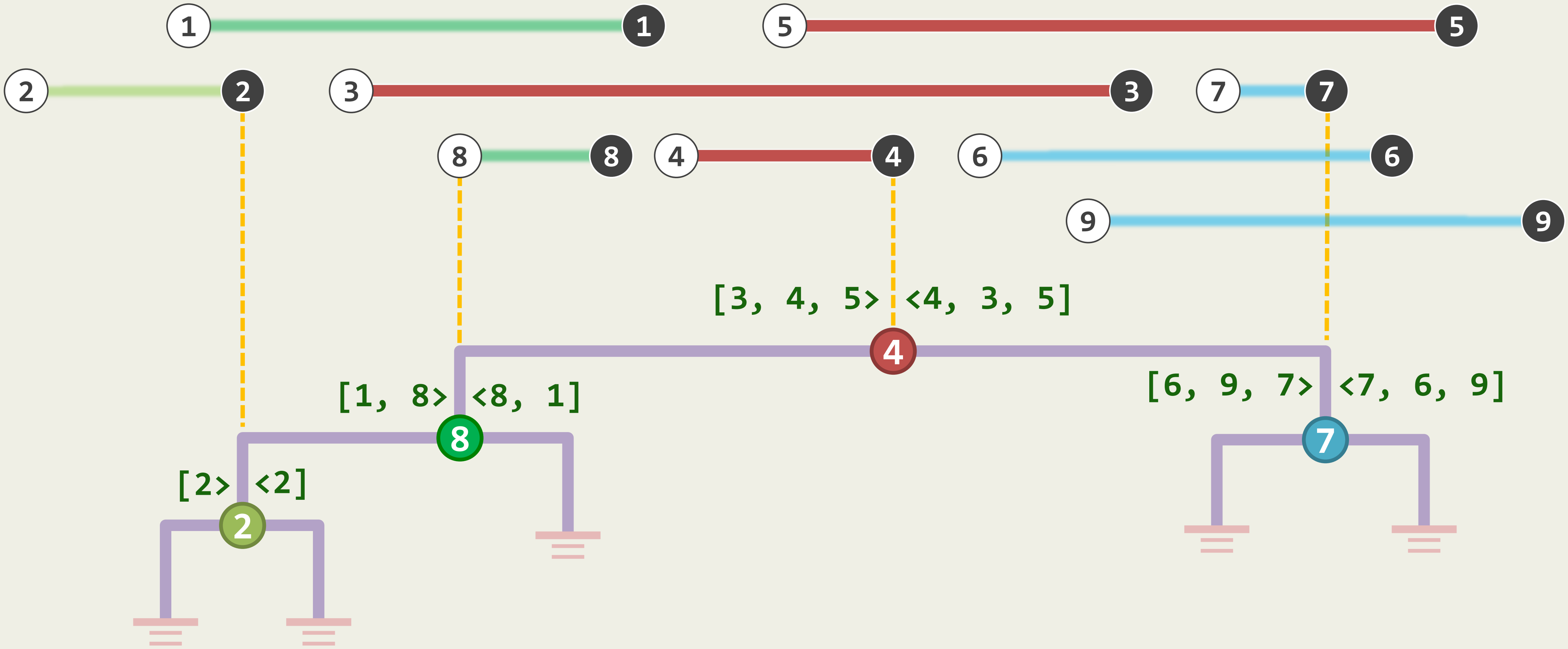
queryIntervalTree(v, q_x)

```
if ( ! v ) return;  //base

if ( qx < xmid(v) )

   report all segments of Smid(v) containing qx;

   queryIntervalTree( lc(v), qx );

else if ( xmid(v) < qx )

   report all segments of Smid(v) containing qx;

   queryIntervalTree( rc(v), qx );

else //with a probability ≈ 0

   report all segments of Smid( v ); //both rc(v) & lc(v) can be ignored
```



$\mathcal{O}$(r + logn) Query Time

❖ Each query visits $\mathcal{O}$(logn) nodes //LINEAR recursion



07-XC

BST Application

Segment Tree

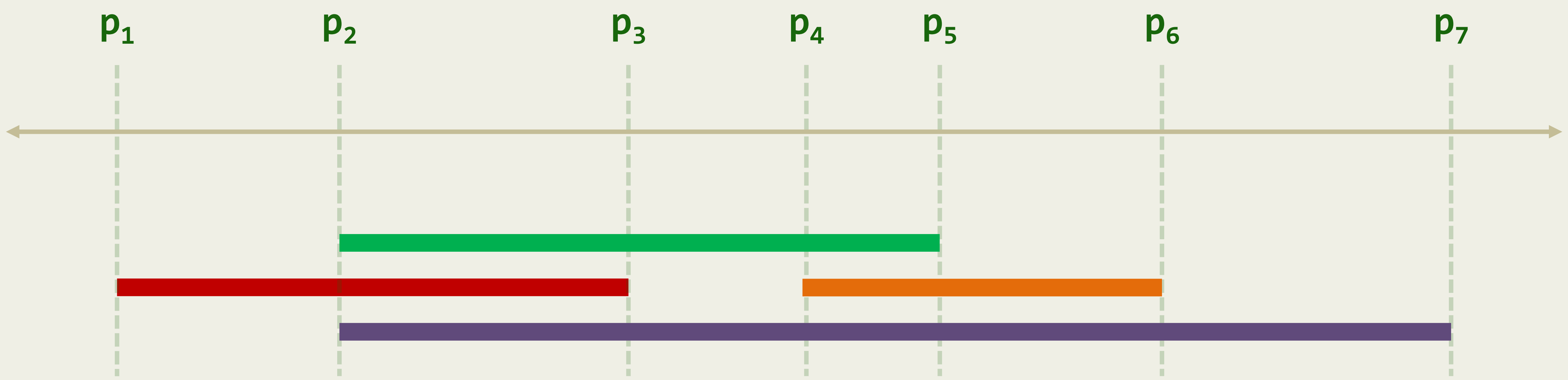
把一条线分割成不相等的两段，再把这两段按照同样的比例再分成两个部分。假设第一次分出来的两段中，一段代表可见世界，另一段代表理智世界。然后再看第二次分成的两段，他们分别代表清楚与不清楚的程度，你便会发现，可见世界那一段的第一部分是它的影像。

邓俊辉

deng@tsinghua.edu.cn

Elementary Intervals

- Let $I = \{ [x_i, x_i'] \mid i = 1, 2, 3, \ldots, n \}$ be n intervals on the x-axis
- Sort all the endpoints into $\{ p_1, p_2, p_3, \ldots, p_m \}$, $m \leq 2n$



- m+1 elementary intervals are hence defined as:

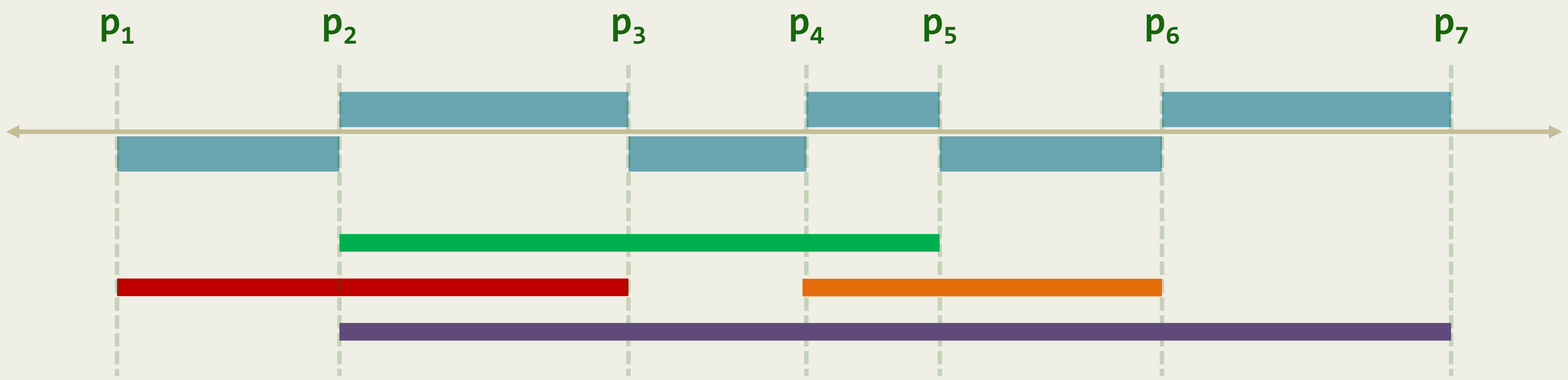
$$(-\infty, p_1],\ (p_1, p_2],\ (p_2, p_3],\ \ldots,\ (p_{m-1}, p_m],\ (p_m, +\infty]$$

Discretization

- Within each EI, all stabbing queries share a same output

∴ If we **sort** all EI's into a vector and

store the corresponding **output** with each EI, then ...

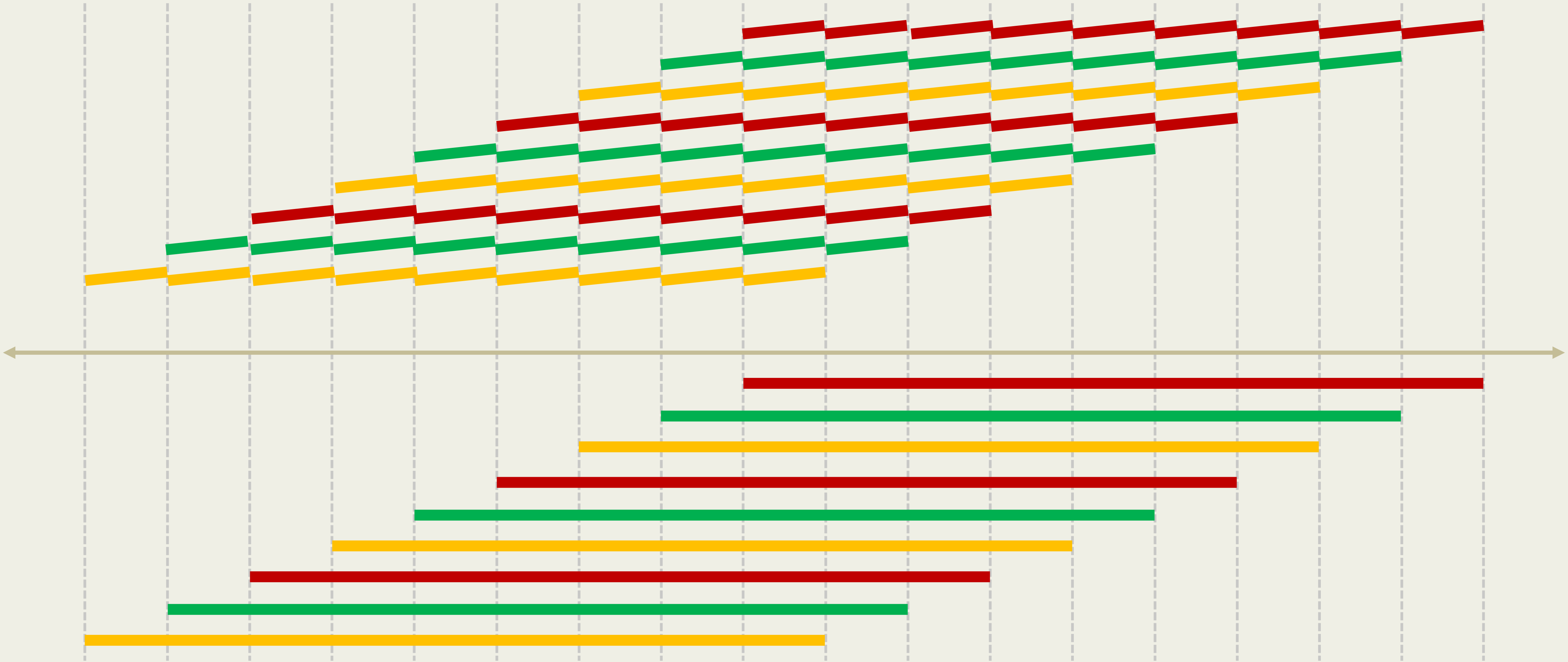


∴ Once a query position is determined, //by an $\mathcal{O}(\log n)$ time binary search

the output can then be returned directly //$\mathcal{O}(r)$

Worst Case

❖ Every interval spans $\Omega(n)$ EI's and a total space of $\Omega(n^2)$ is required

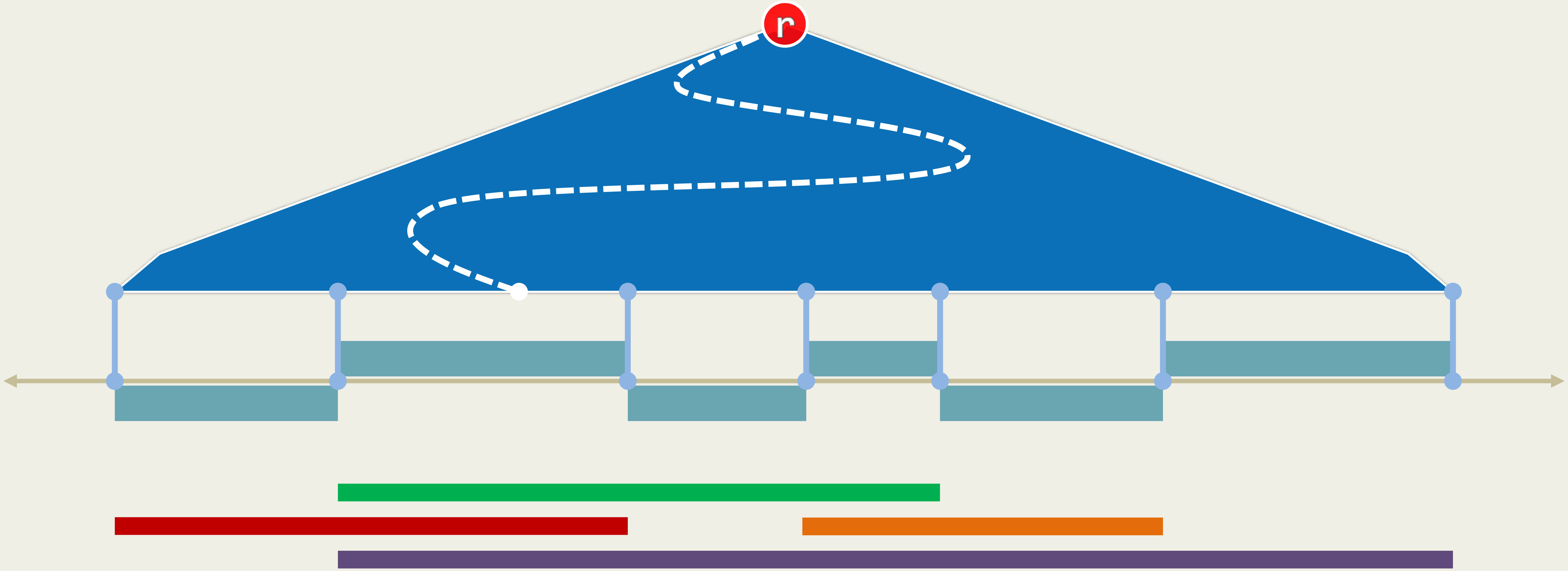


Sorted Vector --> BBST

❖ For each leaf v,

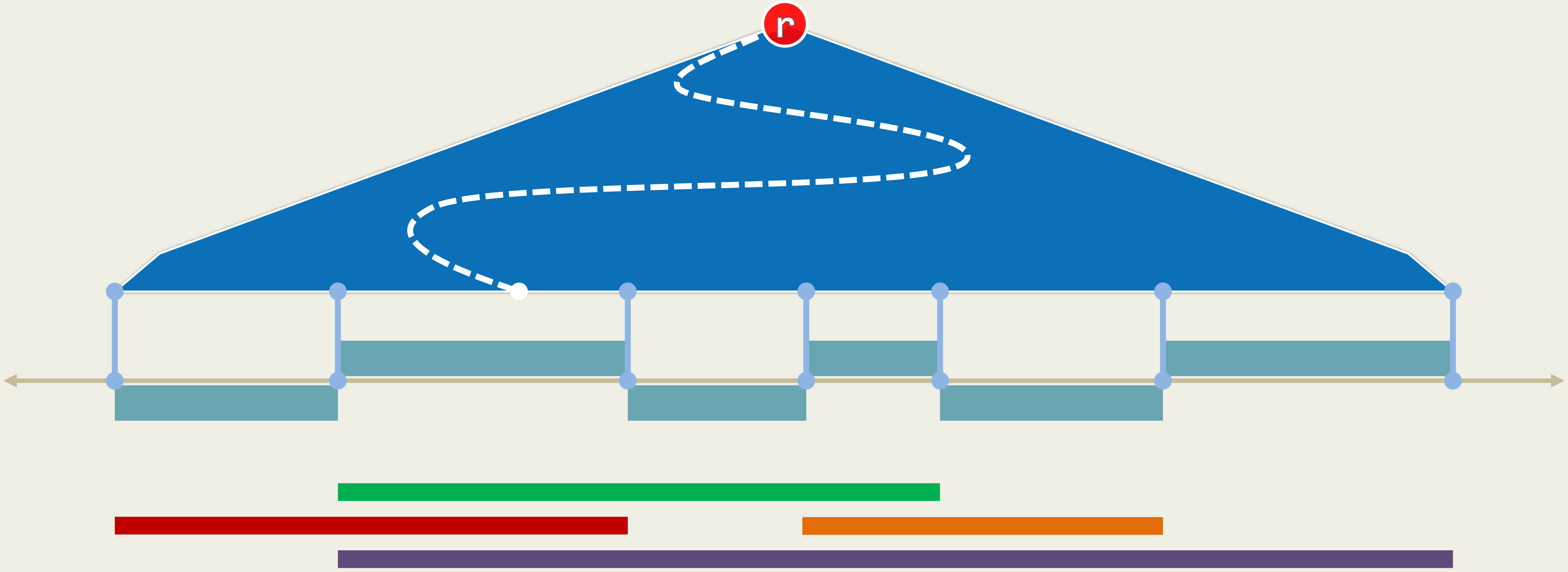
denote the corresponding elementary interval as R(v), //range of domination

denote the subset of intervals spanning R(v) as Int(v) and store Int(v) at v

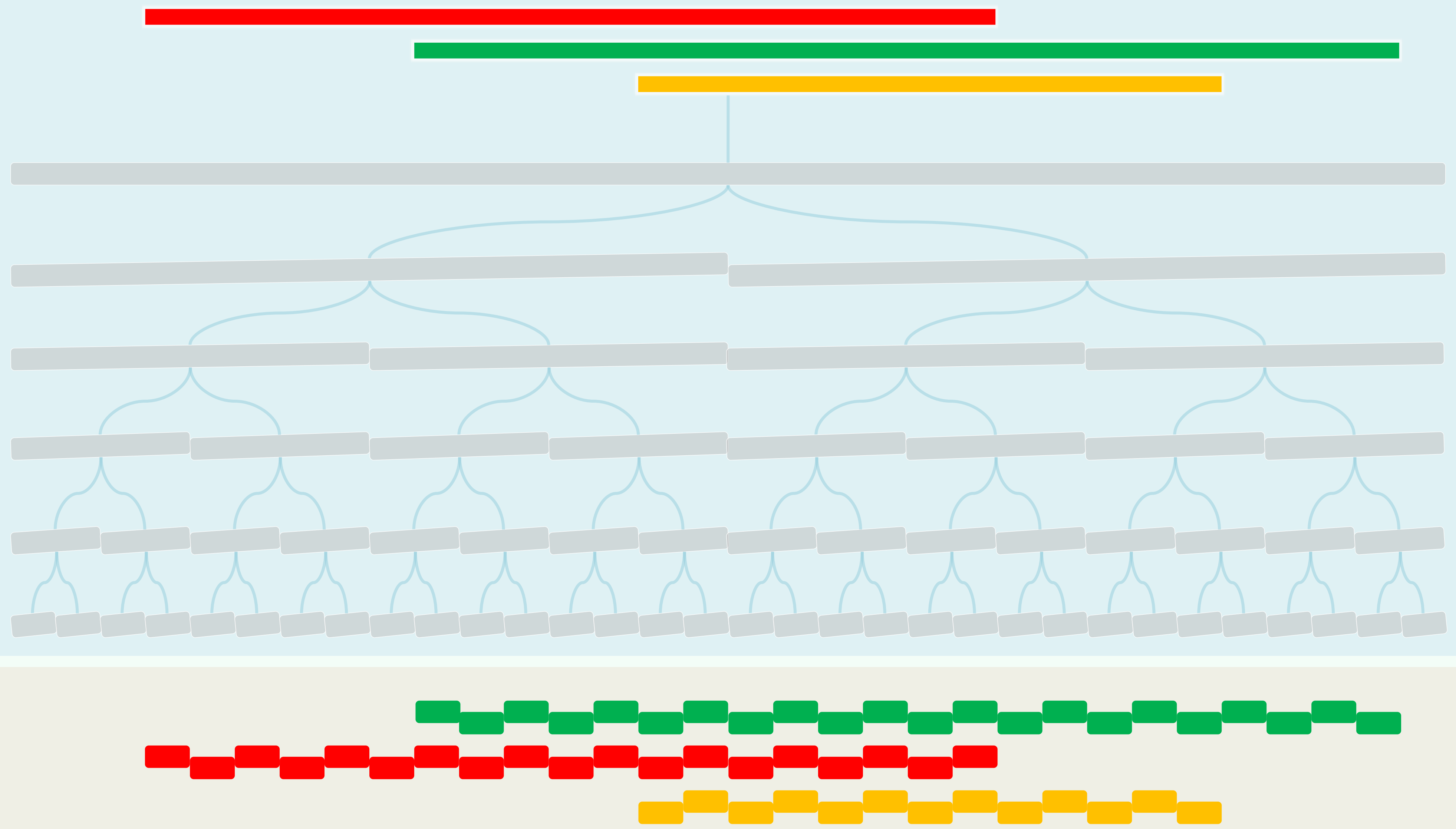


1D Stabbing Query with BBST

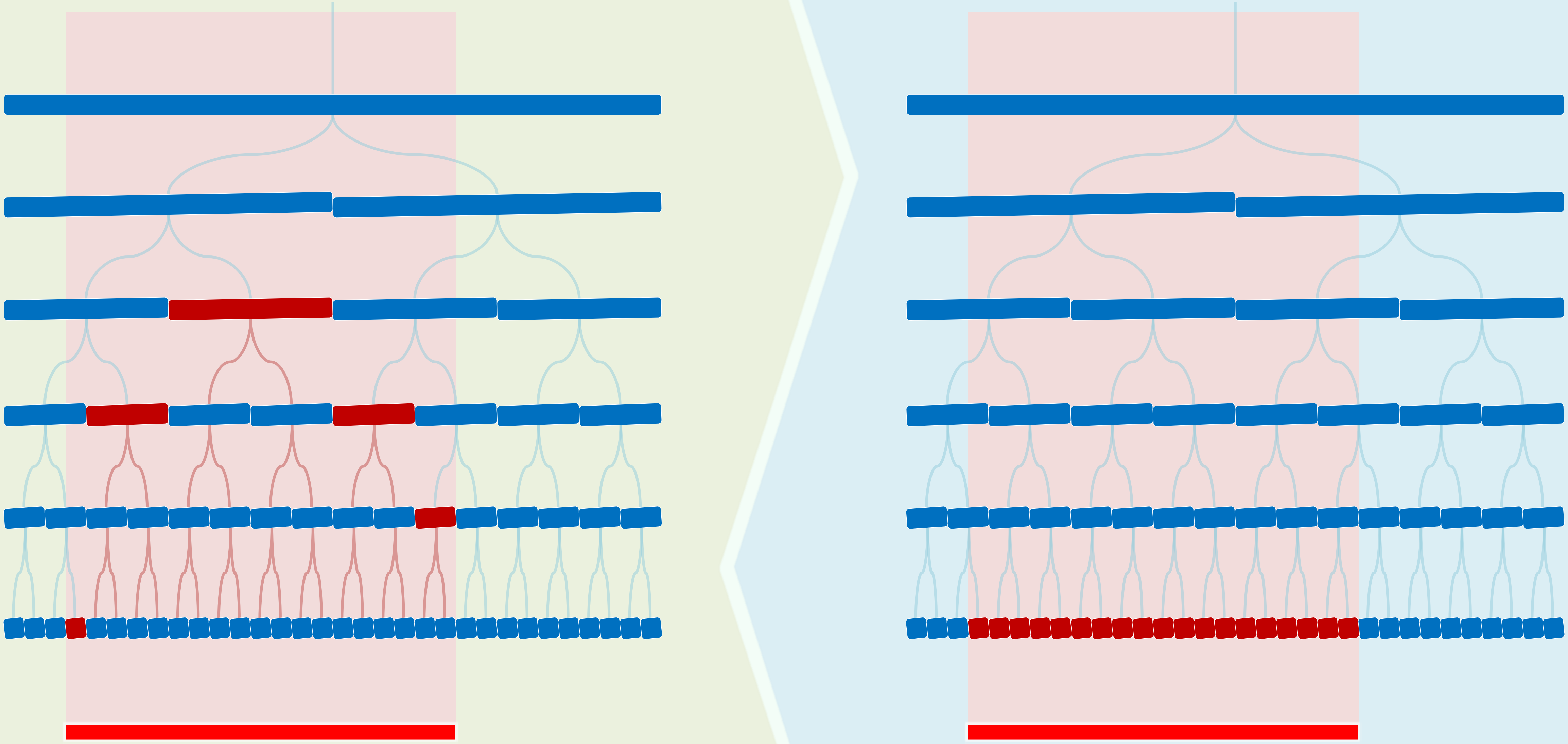
- To find all intervals containing q_x, we can
  - find the leaf v whose R(v) contains q_x // $\mathcal{O}(\log n)$ time for a BBST
  - and then report Int(v) // $\mathcal{O}(1 + r)$ time



$\Omega(n^2)$ Total Space In The Worst Cases



Store each interval at $\mathcal{O}(\log n)$ common ancestors by greedy merging

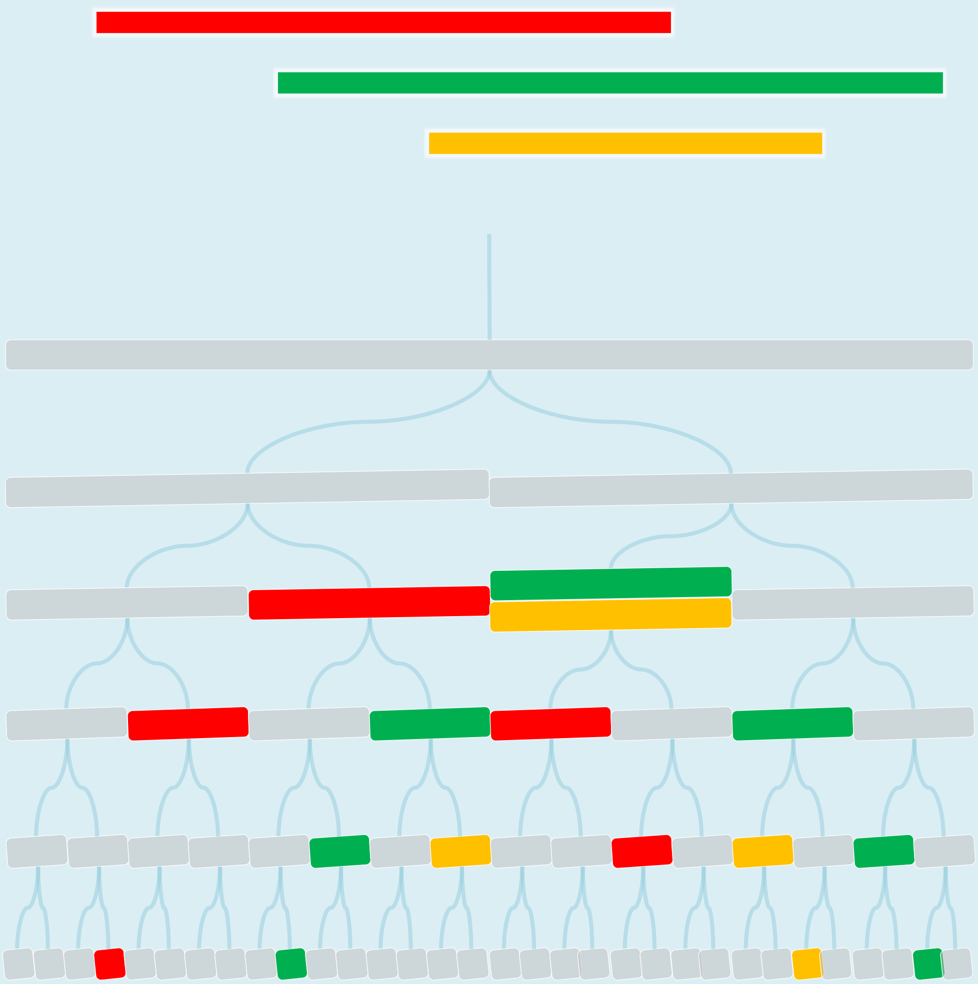


Canonical Subsets with $\mathcal{O}(n\log n)$ Space

❖ Denote the interval subset stored at node v as Int(v)

BuildSegmentTree(I)

```
Sort all endpoints in I before
   determining all the EI's //O(nlogn)
Create T a BBST on all the EI's //O(n)
Determine R(v) for each node v
   //O(n) if done in a bottom-up manner
For each s of I
   InsertSegment( T.root, s )
```



InsertSegment(v , s)

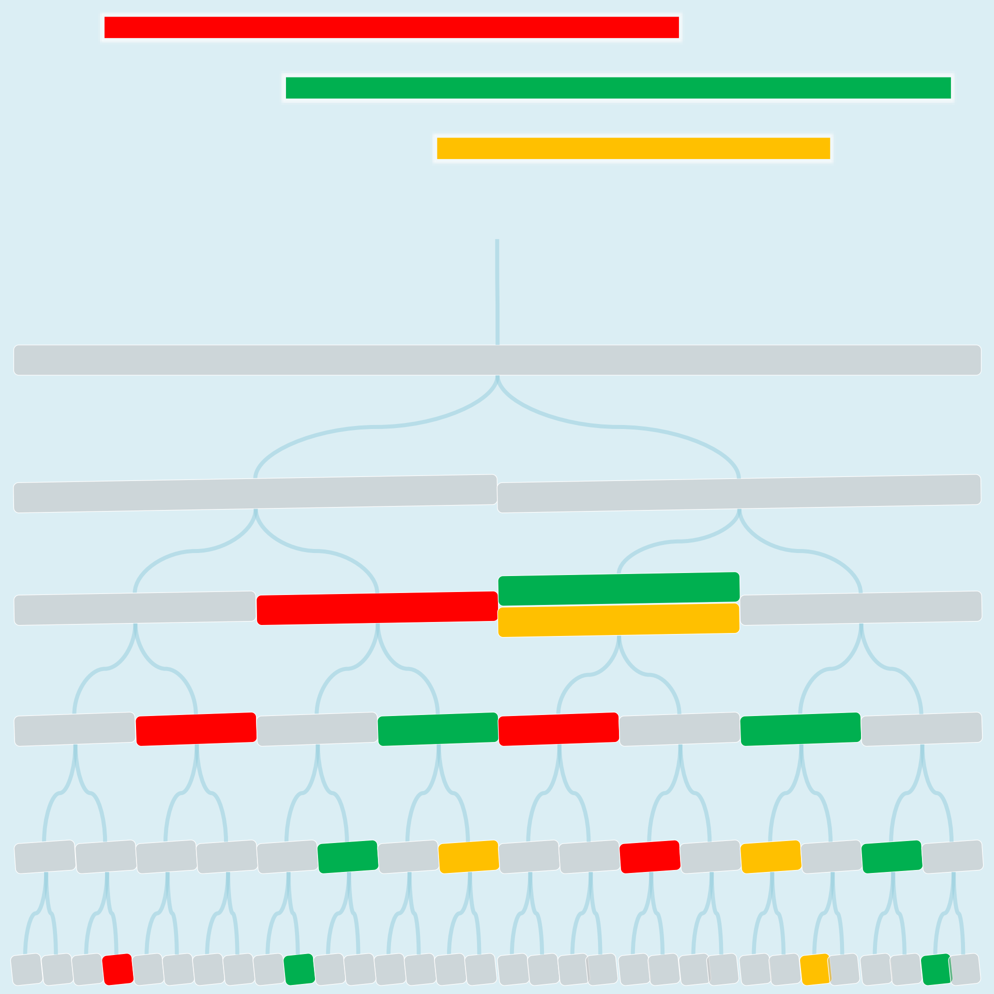
```
if ( R(v) ⊆ s ) //greedy by top-down
   store s at v and return;
if ( R( lc(v) ) ∩ s ≠ ∅ ) //recurse
   InsertSegment( lc(v), s );
if ( R( rc(v) ) ∩ s ≠ ∅ ) //recurse
   InsertSegment( rc(v), s );
```

At each level,

$\leq$ 4 nodes are visited

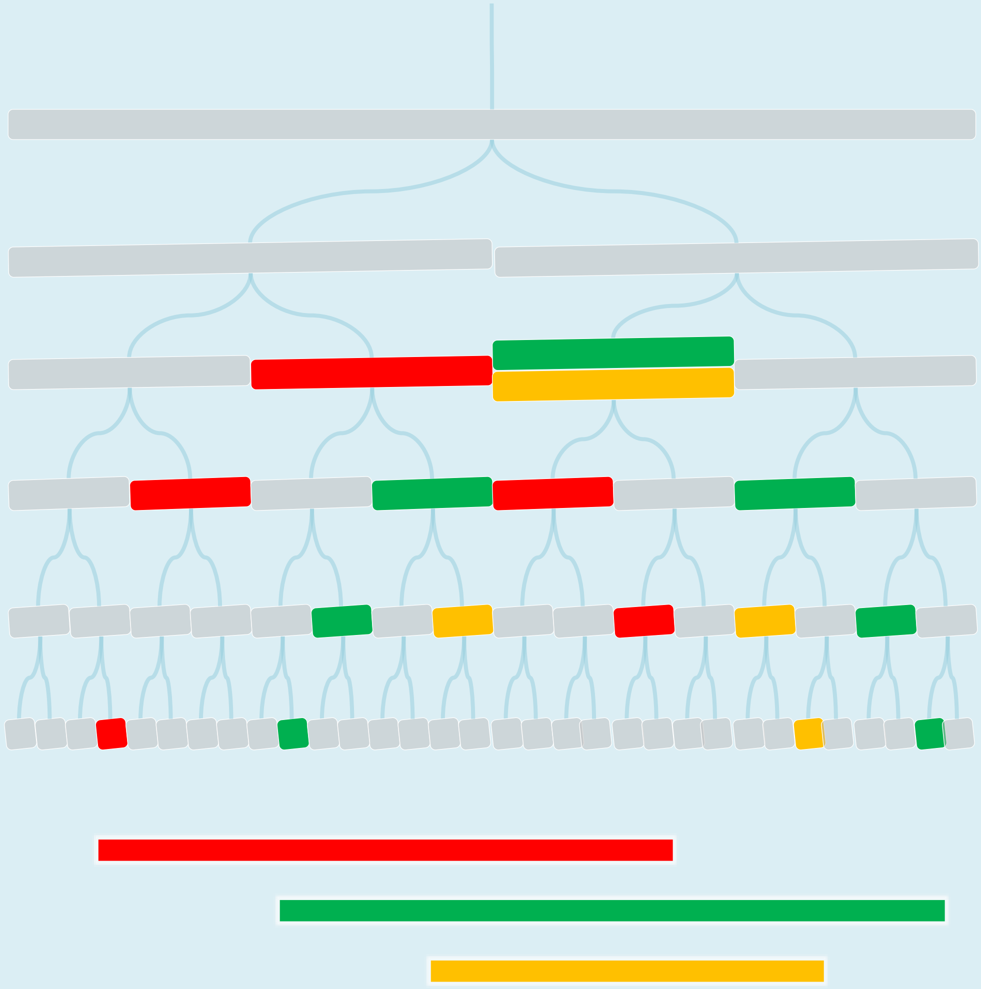
(2 stores + 2 recursions)

$\therefore$ $\mathcal{O}(\log n)$ time



Query(v , q_x)

```
report all the intervals in Int(v)

if ( v is a leaf )

   return

if ( qx ∈ Int( lc(v) ) )

   Query( lc(v), qx )

else

   Query( rc(v), qx )
```



$\mathcal{O}(r + \log n)$

- Only one node is visited per level,

  altogether $\mathcal{O}(\log n)$ nodes

- At each node v

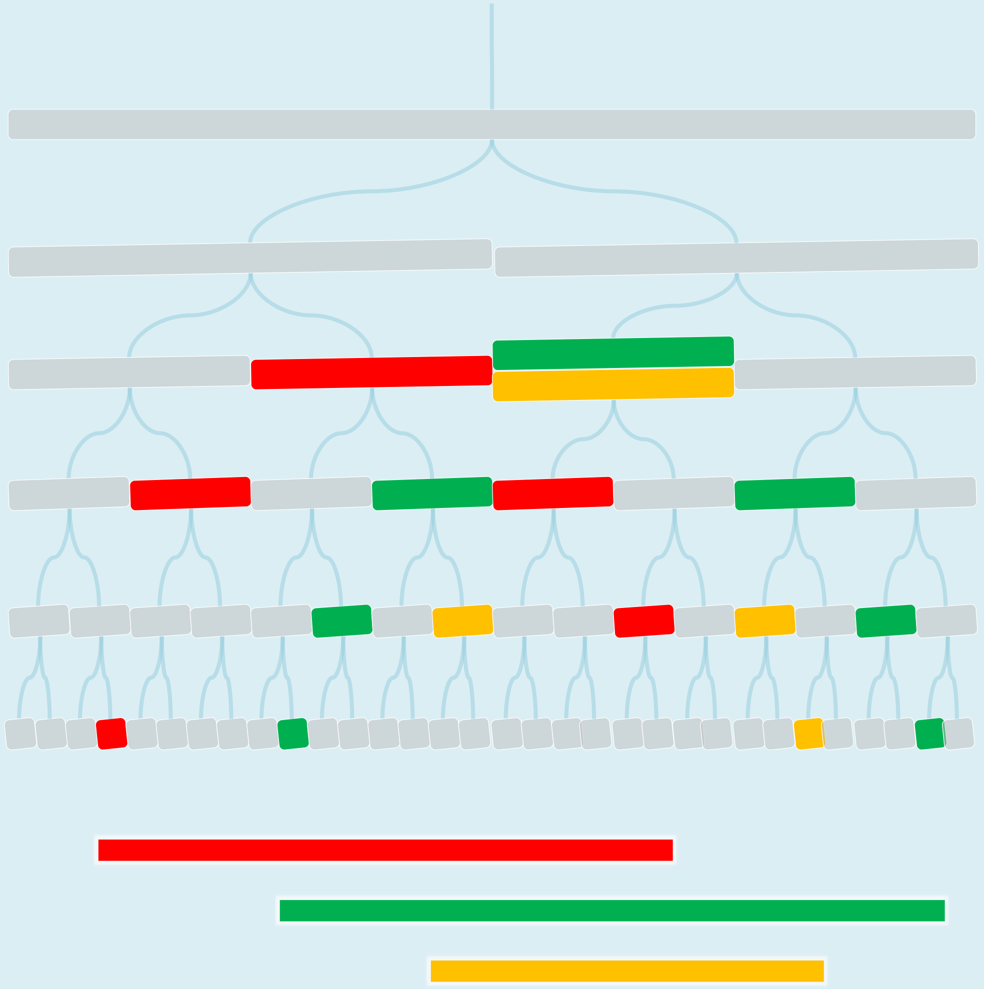
  - the CS Int(v) is reported

  - in time

    $$1 + |\mathrm{Int}(v)| = \mathcal{O}(1 + r_v)$$

∴ Reporting all the intervals

costs $\mathcal{O}(r)$ time



Conclusion

- For a set of n intervals,
  - a segment tree of size $\mathcal{O}(n\log n)$
  - can be built in $\mathcal{O}(n\log n)$ time
  - which reports all intervals

    containing a query point

    in $\mathcal{O}(r + \log n)$ time