IP Lookup and Range Searching

Haim Kaplan

Tel Aviv University

Joint with: Lars Arge, Pankaj Agarwal, Moshik Hershcovitch, Eyal Molad, Bob Tarjan, Ke Yi

Longest Prefix Forwarding

- Packet has a destination address
- Router identifies the longest prefix of the destination address to find the next hop

forwarding table

destination

12.34.158.5 →

4.0.0.0/8
4.83.128.0/17
12.0.0.0/8
12.34.158.0/24
126.255.103.0/24

→ outgoing link

The table is dynamic

Routing protocols insert and delete prefixes

132.66.235.0/24

4.0.0.0/8

forwarding table

destination

12.34.158.5

4.0.0.0/8
4.83.128.0/17
12.0.0.0/8
12.34.158.0/24
126.255.103.0/24

outgoing link

The longest prefix problem

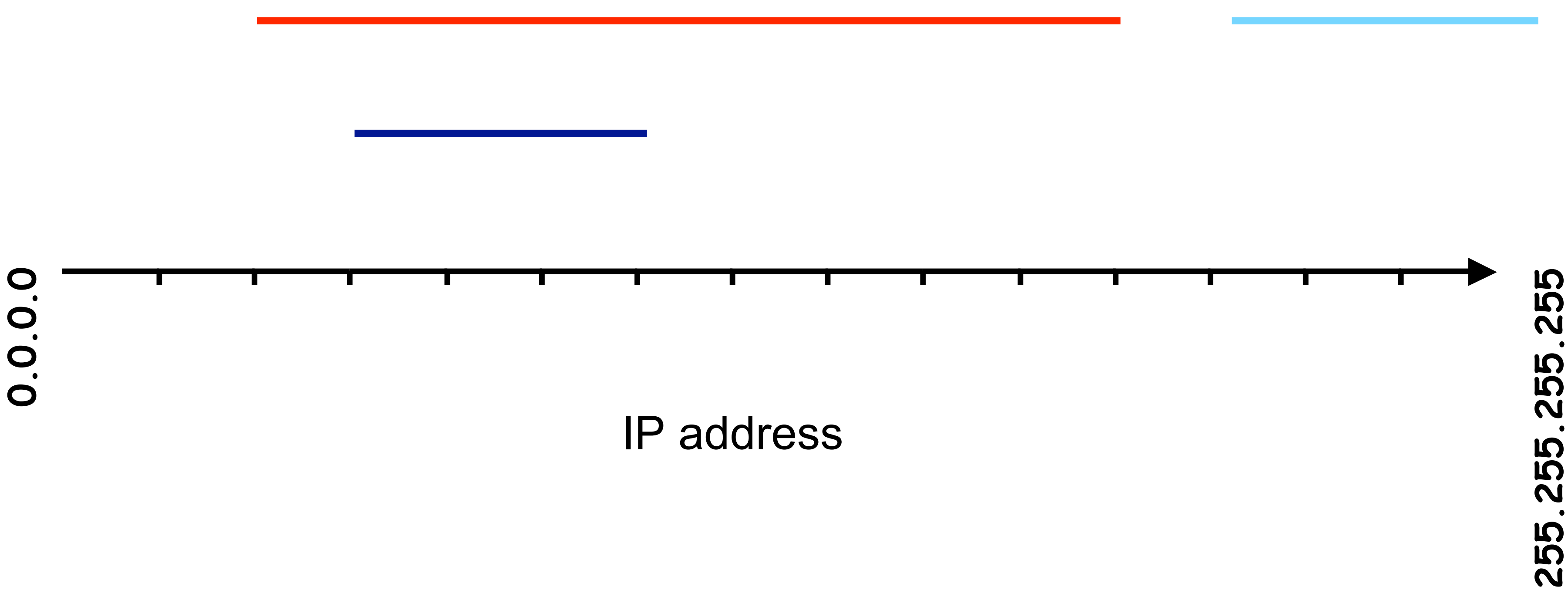
Given a set of strings $S = \{p_1,\ldots,p_n\}$ (prefixes) build a data structure such that

Given a string q we can find (efficiently) the longest prefix of q in S

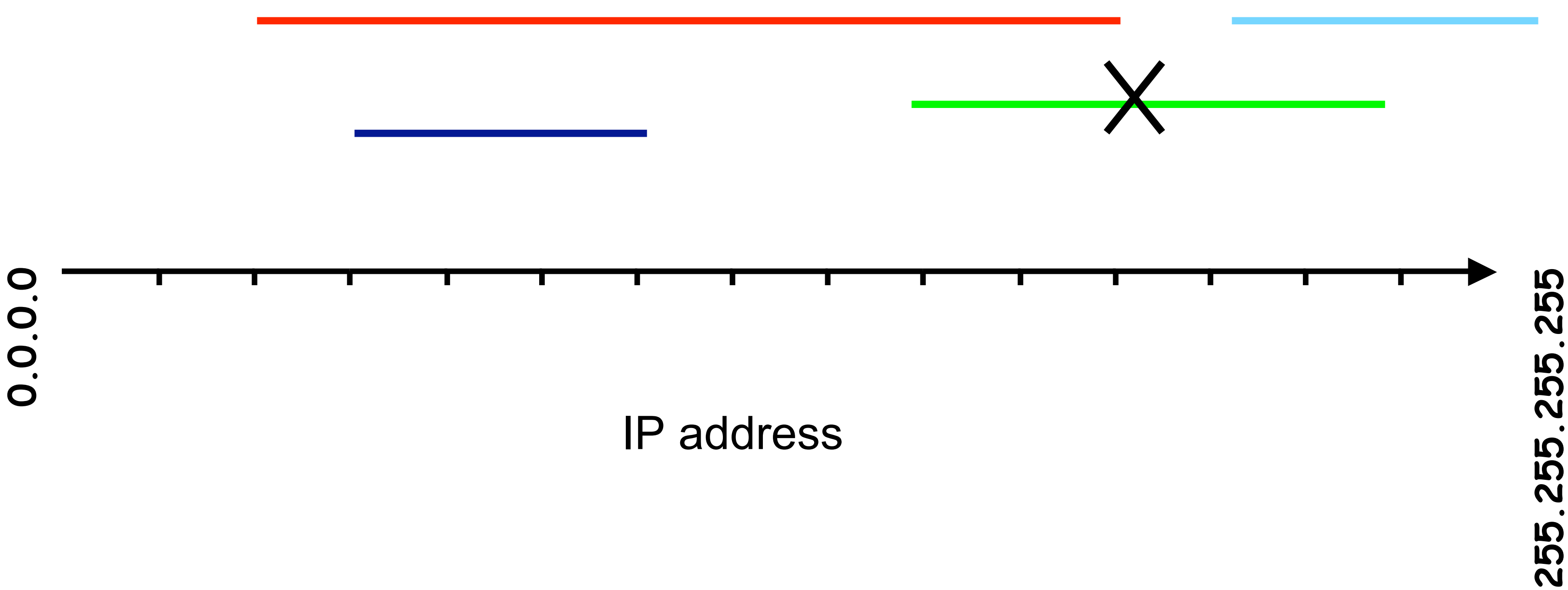
Updates – insert or delete a prefix

We can model this as follows

Each segment corresponds to a prefix

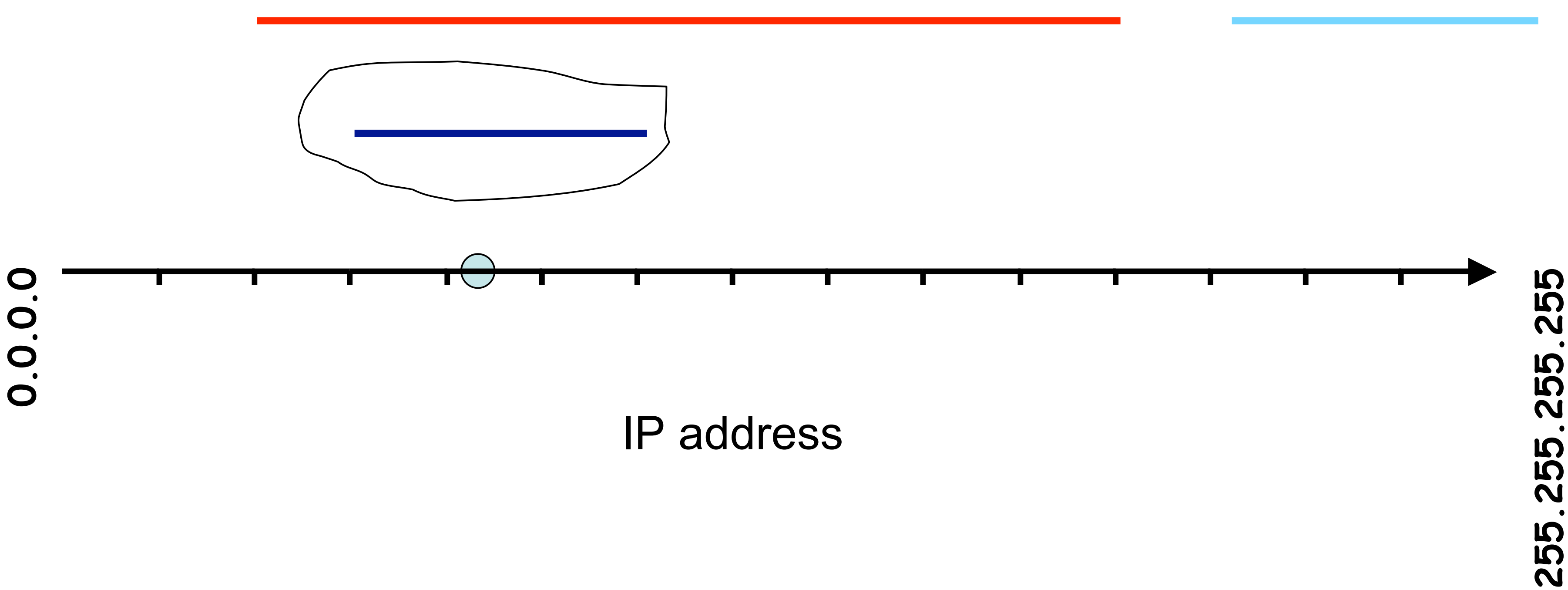


Segments are nested

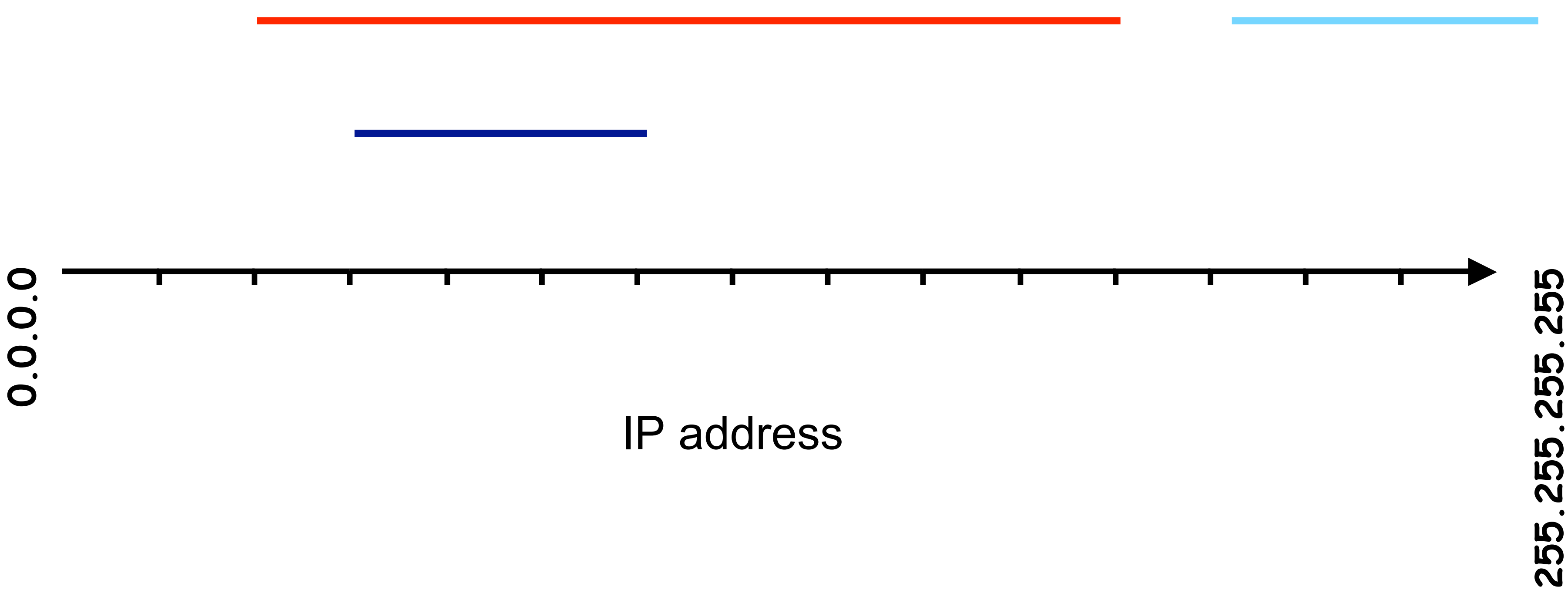


A packet is a point

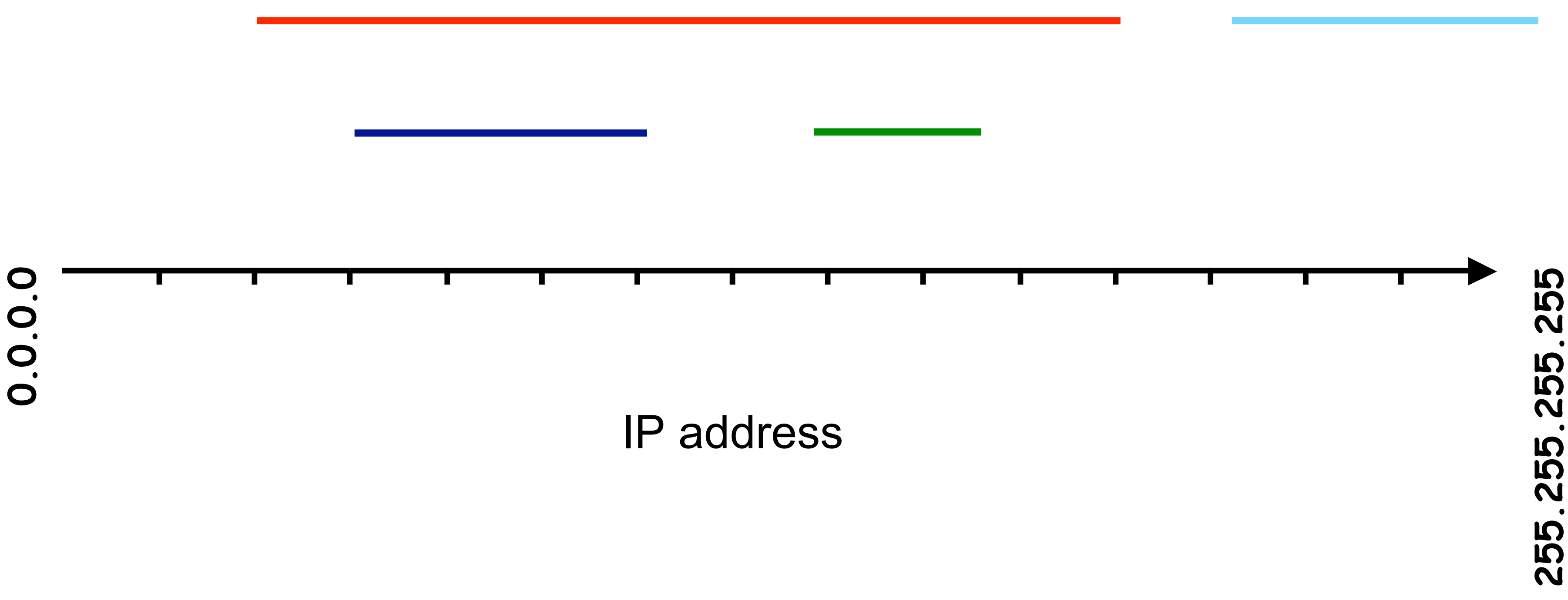
Want the shortest segment that contains the packet



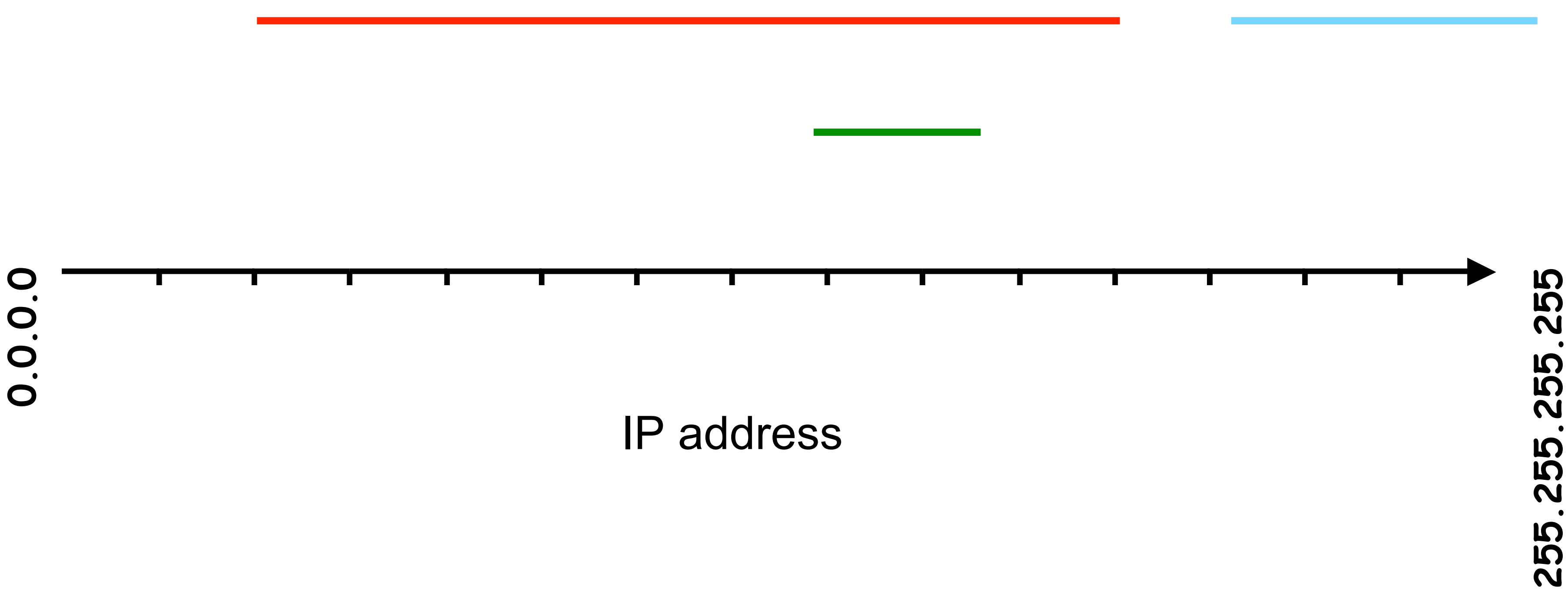
Want to be able to insert/delete segments



Want to be able to insert/delete segments



Want to be able to insert/delete segments



Discussion

- In the segment-stabbing problem we assume that we can compare endpoints in $O(1)$ time
- This may be reasonable if strings are short
- It is less reasonable if we try to solve the longest prefix problem for arbitrary strings

Results (1) (SWAT 2008, HK)

- A very simple data structure for shortest segment in a nested family
  $O(\log(n))$ time, and $O(\log_B(n))$ I/Os per op

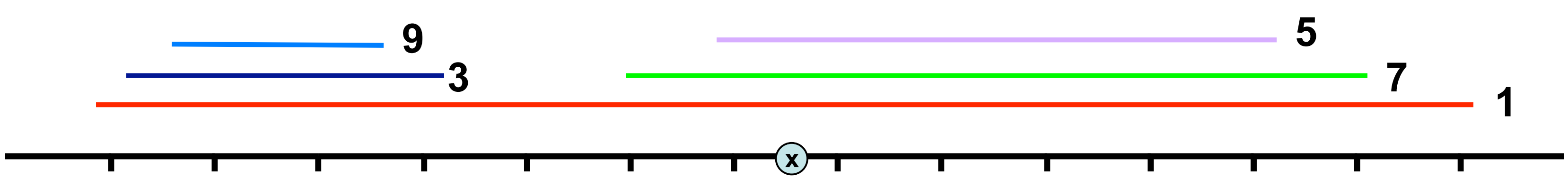
- A data structure for longest prefix in a collection of arbitrary strings
  $O(\log(n) + |q|)$ time and $O(\log_B(n) + |q|/B)$ I/Os per op

both take linear space

Generalizations (1)

Given a set S of nested segments, each with priority assigned to it, build a structure that allows efficient queries of the from:

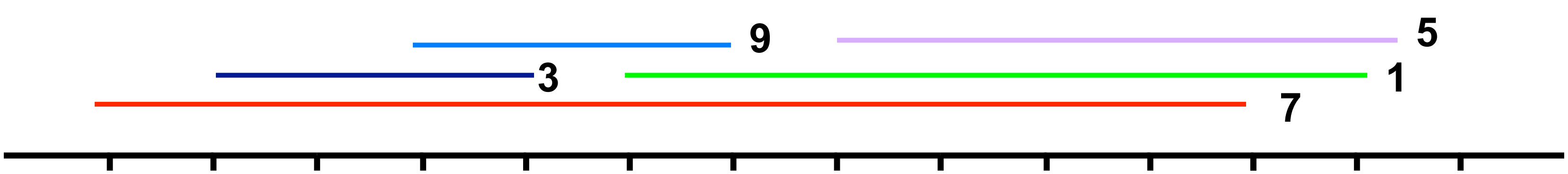
- Given a point x find segment with minimum priority containing it.
- Updates – insert or delete a segment



Generalizations (2)

Given a set S of ~~nested~~ segments, each with priority assigned to it, build a structure that allows efficient queries of the from:

- Given a point x find segment with minimum priority containing it.
- Updates – insert or delete a segment



Motivation for the general problem

- Firewalls
- Rules are intervals/prefixes
- In case several rules apply to a packet then decide by priority

Results (2) (STOC 2003,KMT)

- A simple data structure for nested segments with priorities
  $O(\log(n))$ time per op,
  $O(n)$ space (uses dynamic trees)

- A data structure for general segments
  $O(\log(n))$ time per query/insert but delete takes $O(\log(n)\log\log(n))$ time,
  $O(n\log\log(n))$ space

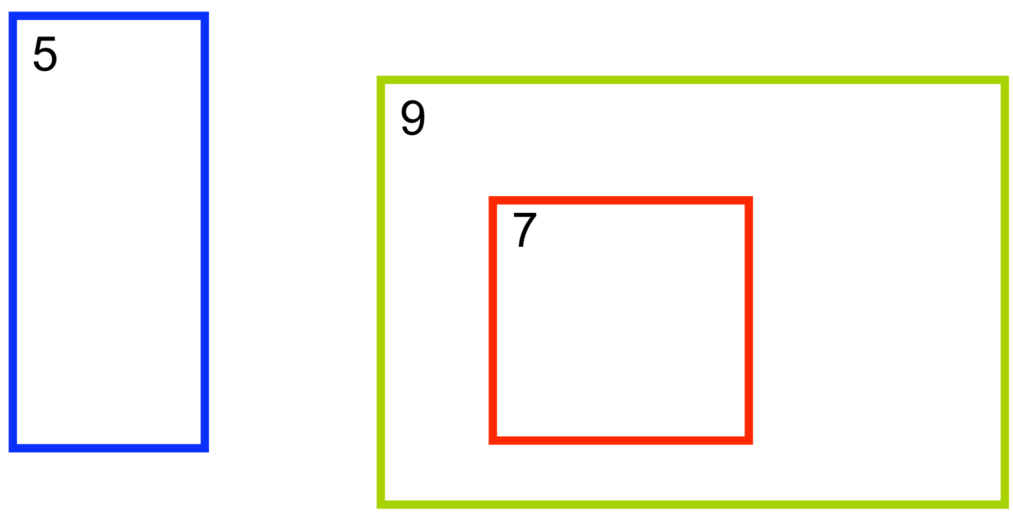
Results (3) (SODA 2005, AAY)

- A data structure for general segments $O(\log(n))$ time per query/insert but delete takes $O(\log(n)\log\cancel{\log}(n))$ time, $O(n\log\cancel{\log}(n))$ space

- $O(\log_B(n))$ I/Os per operation

Results (4) extension to 2D (M'03)

- Query → point in R^2
  - (Sender IP, receiver IP)
- interval → rectangle with priority



We can keep the query time logarithmic for nested rectangles

Previous work: Networking community

- Specific for IP addresses, assume RAM, bounds often depend on W: the length of the address

  (Sahni & Kim : O(n) space O(log n) time per op, complicated, still use RAM)

- trie based solutions
- hash based solutions

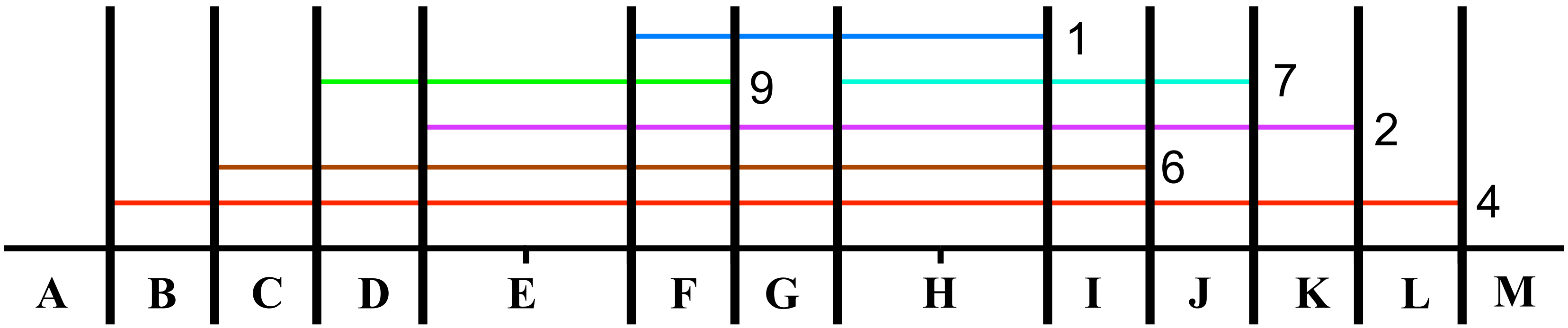
Previous work: Theory community

- Feldman & Muthukrishnan (2000), Thorup (2003) use RAM to get query time below $O(\log(n))$
- Thorup: $O(l)$ query time $O(n^{1/l})$ update time, $O(n)$ space for general priority stabbing

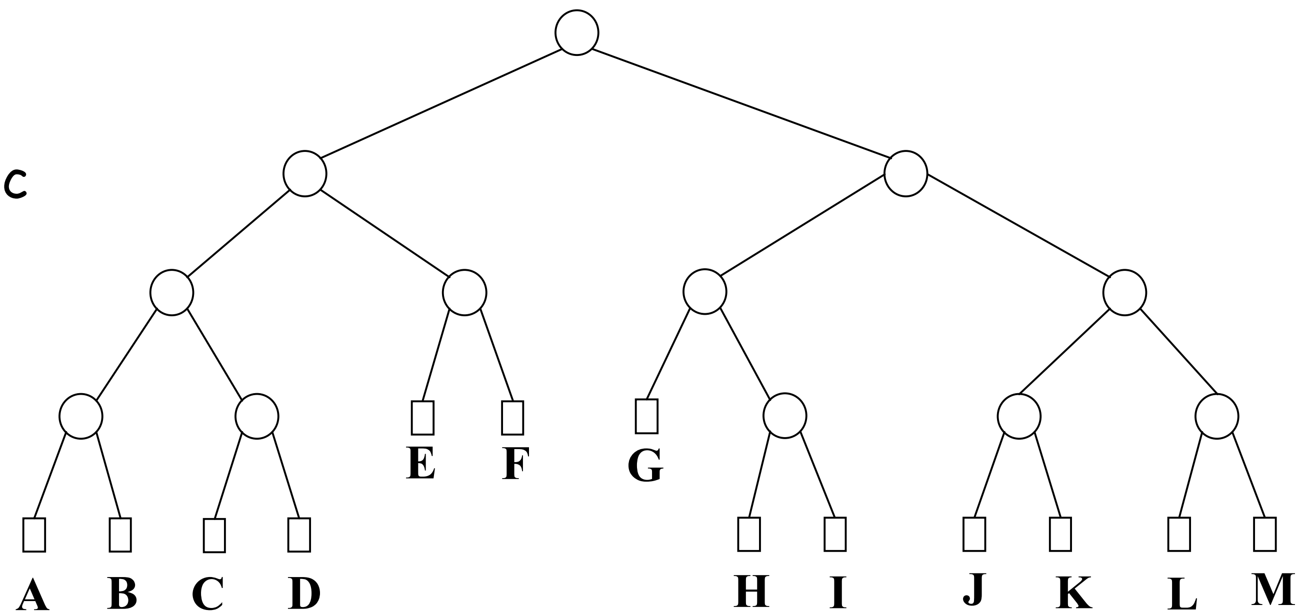
Lets get started...

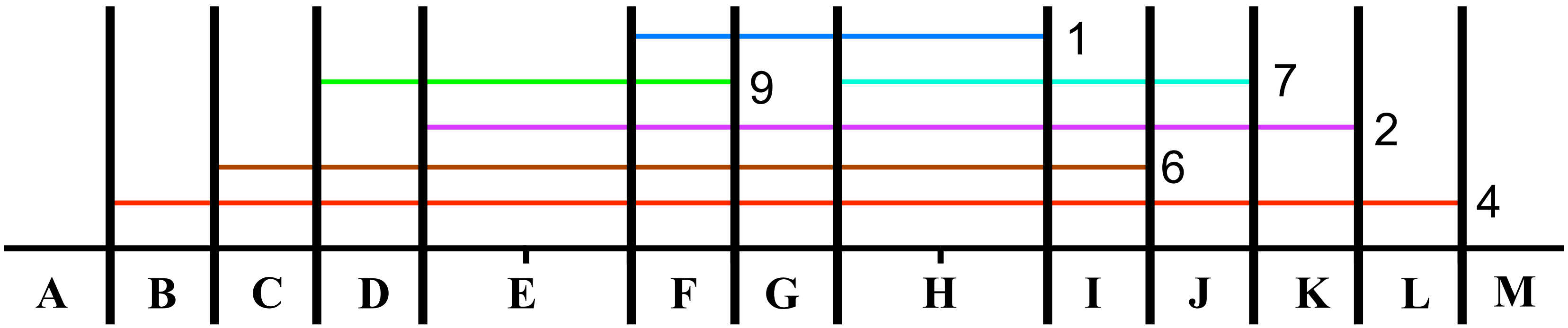
- An update time of $O(\log^2(n))$ using $O(n\log(n))$ space is easy !

Classical solution: Segment tree

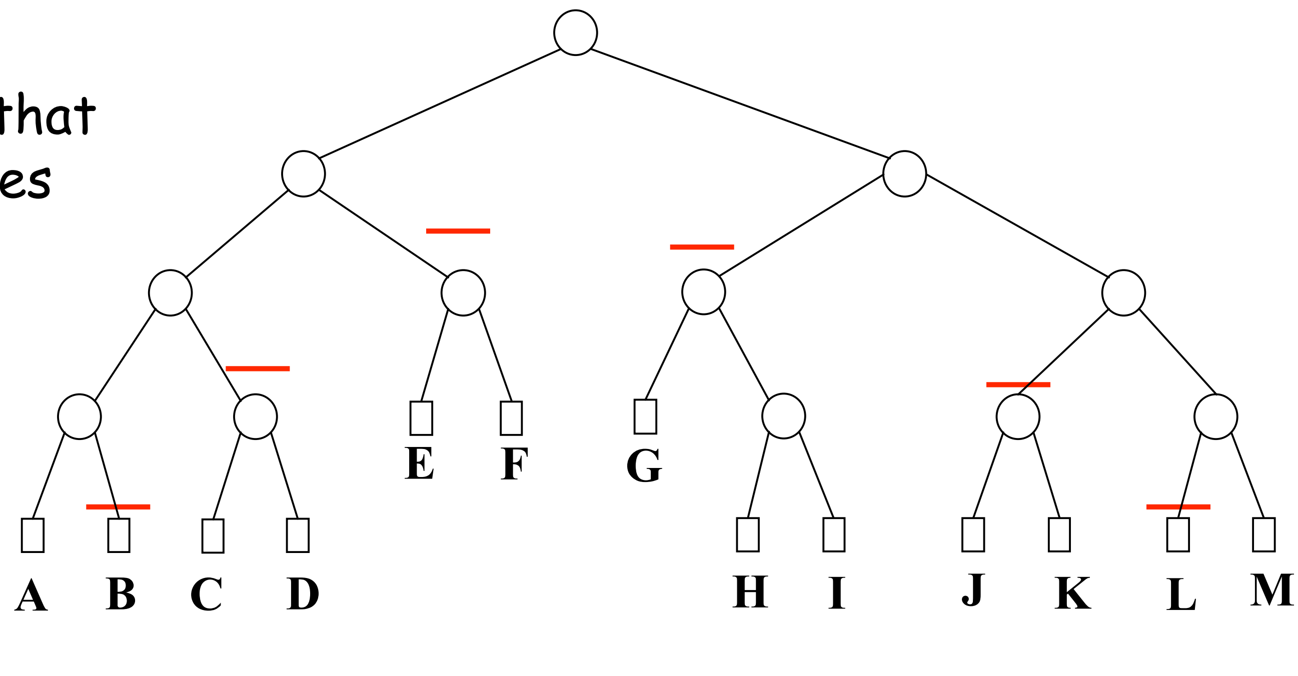


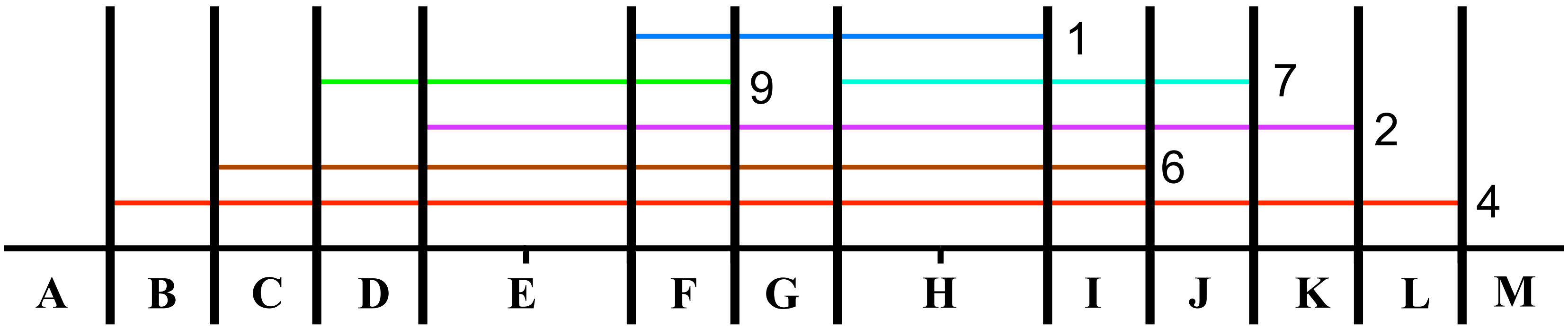
Construct a balanced binary tree over the basic intervals



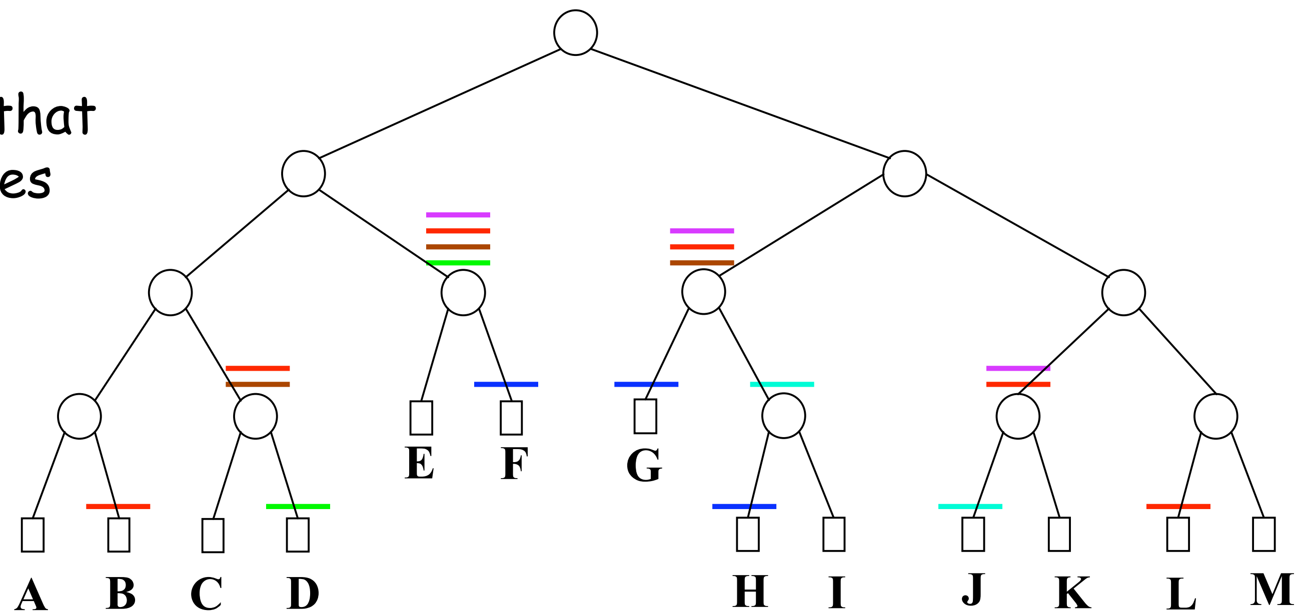


Place segment s in every node v such that s “covers v” but does not “cover p(v)”

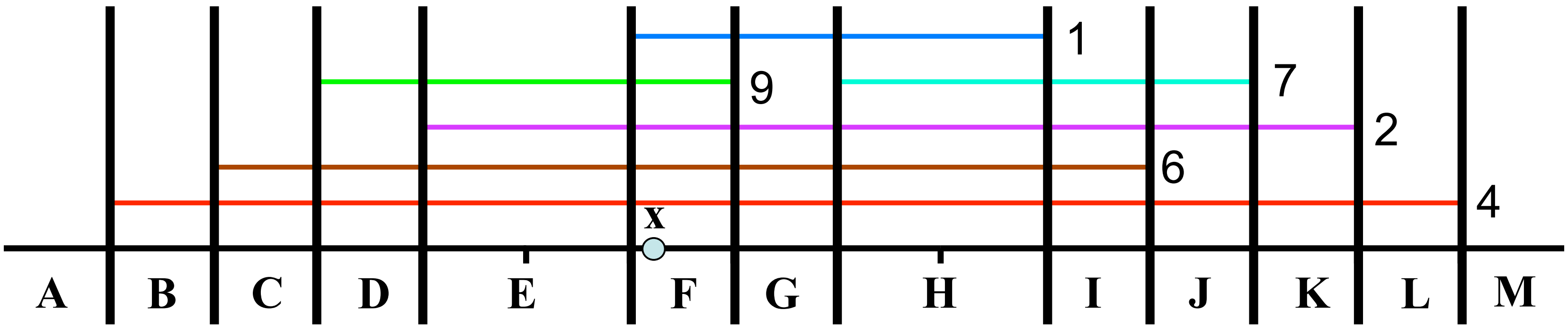




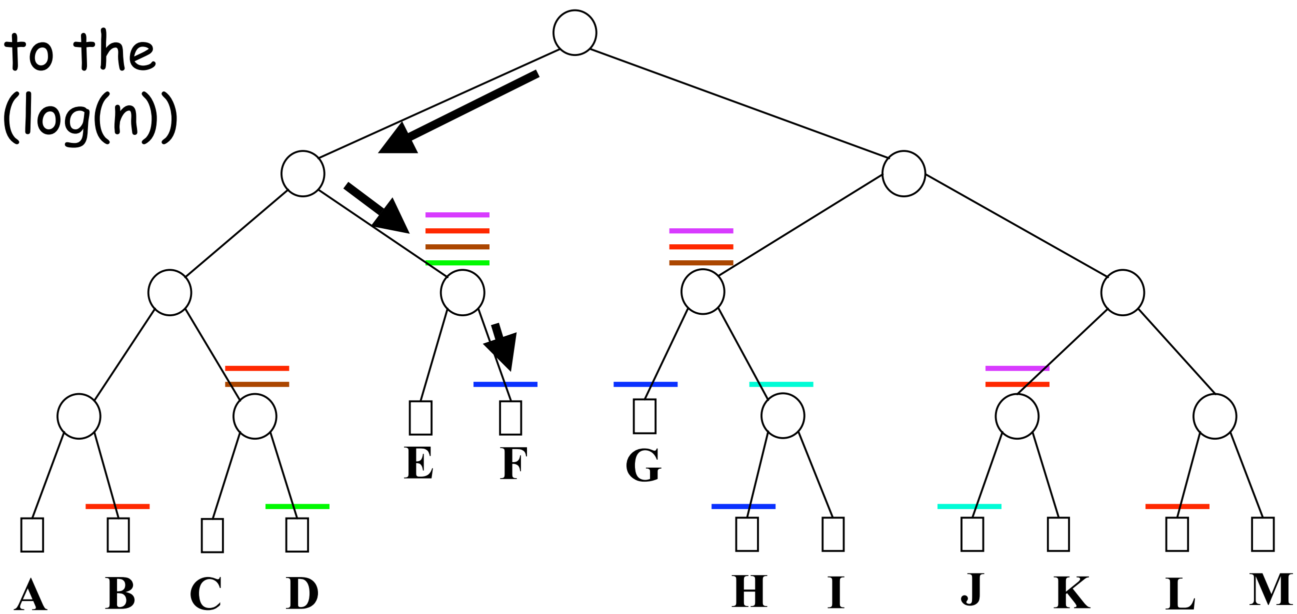
Place segment s in every node v such that s "covers v" but does not "cover p(v)"



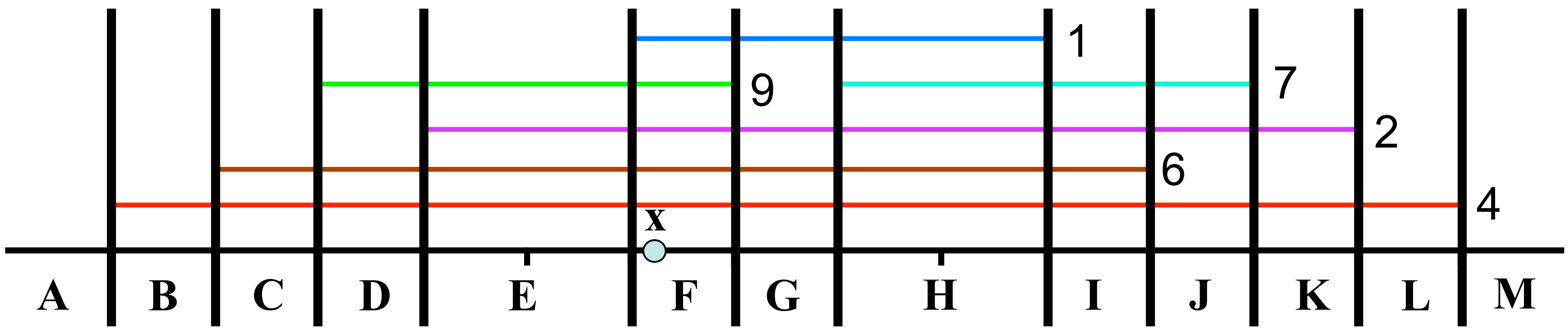
Query



Traverse the path to the leaf containg x - O(log(n)) nodes.



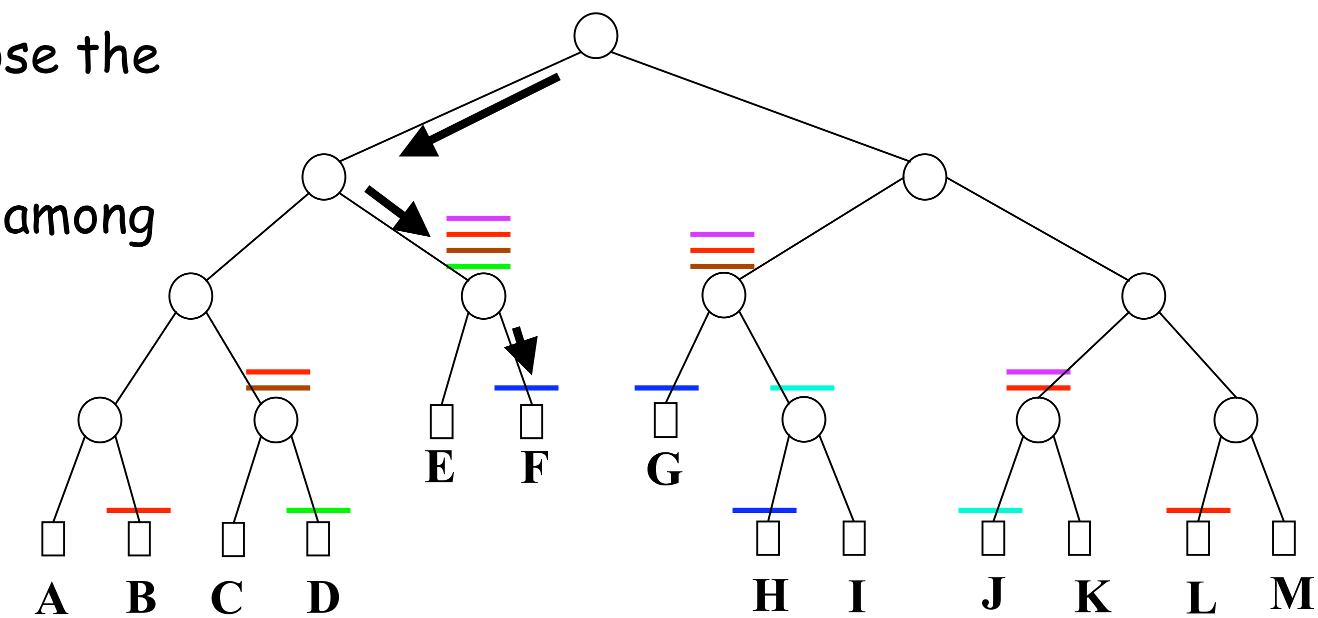
Query



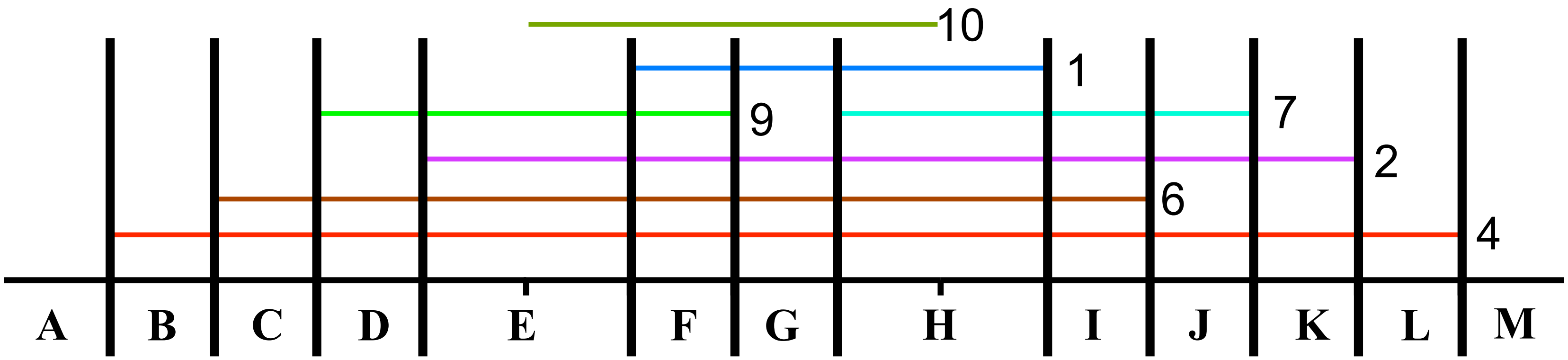
In each node choose the min segment.

Find the minimum among those.

O(log(n)) time

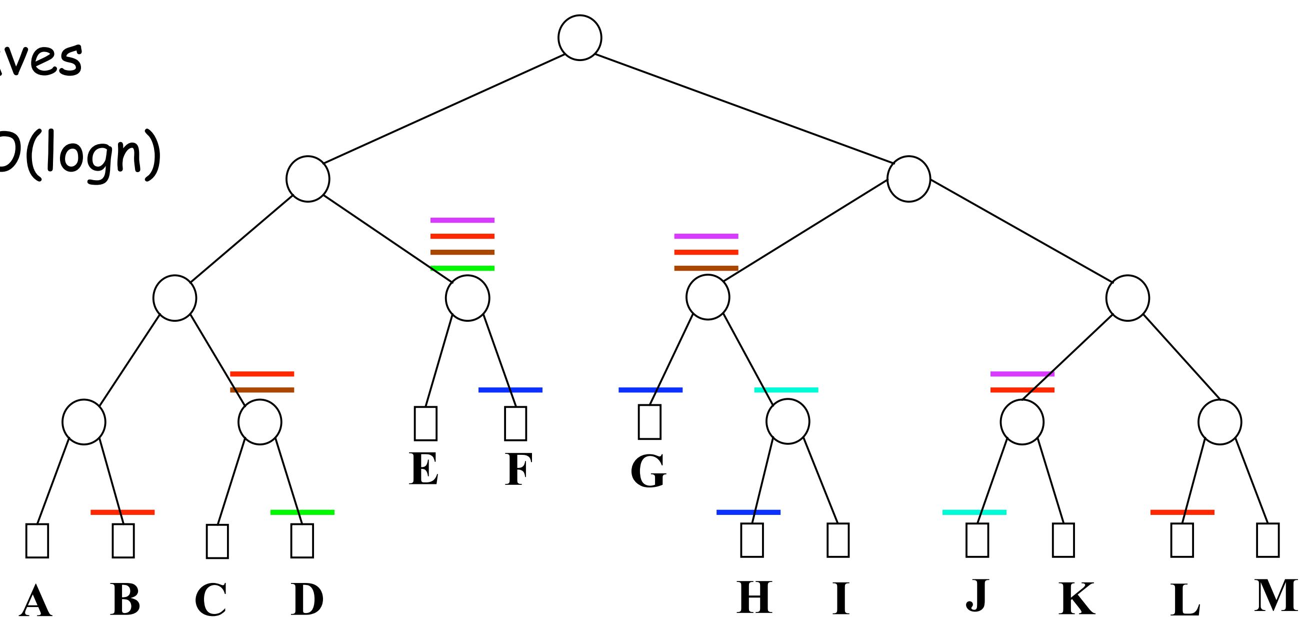


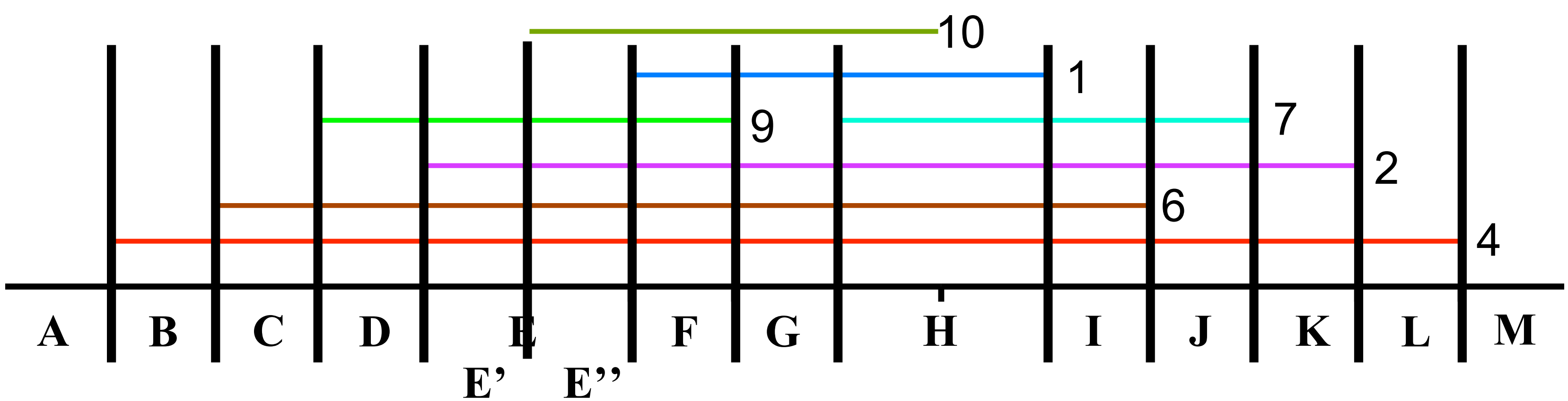
Segment tree - Insert



Insert two new leaves

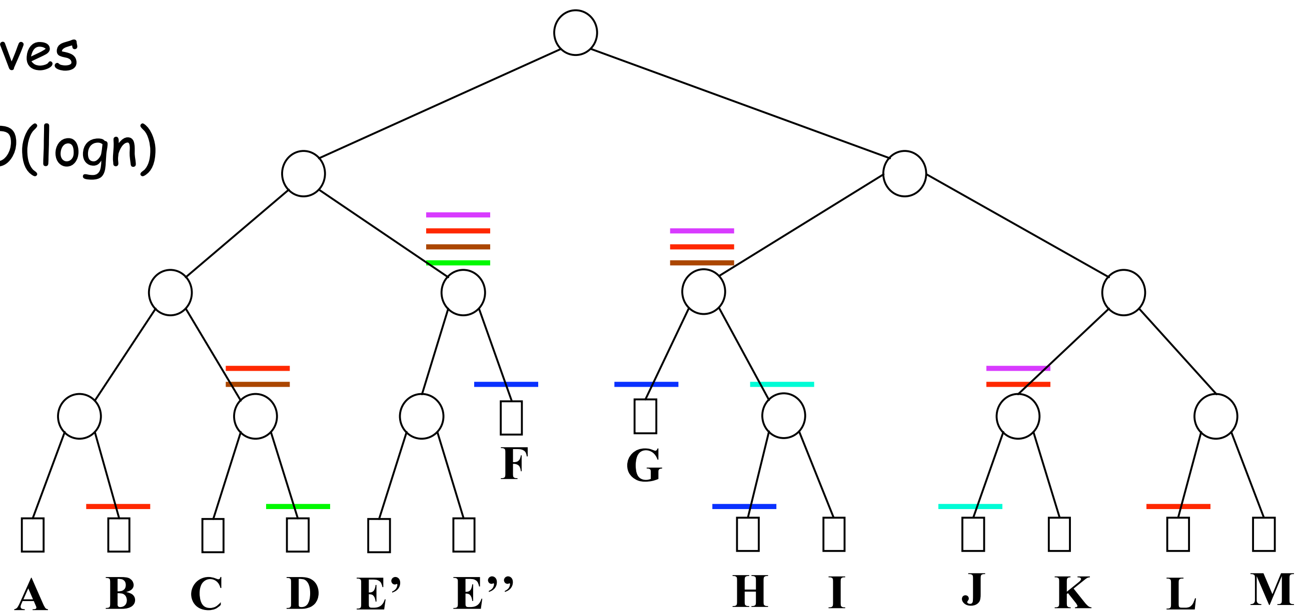
Add a segment in O(logn) nodes

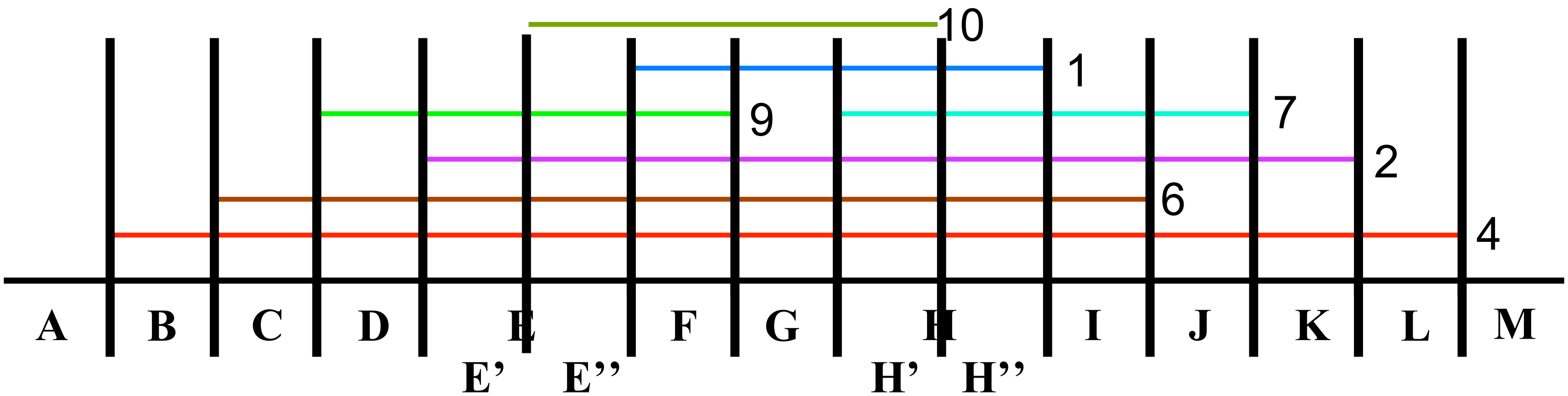




Insert two new leaves

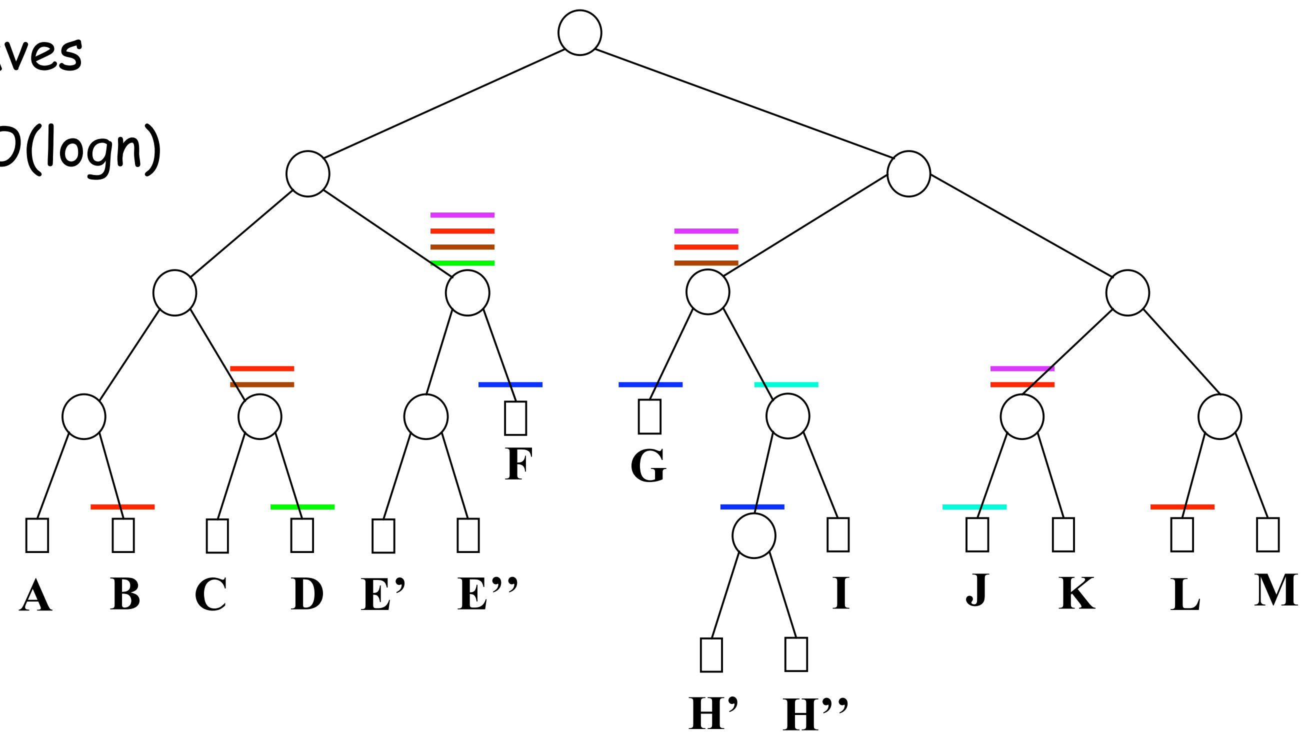
Add a segment in O(logn) nodes





Insert two new leaves

Add a segment in O(logn) nodes



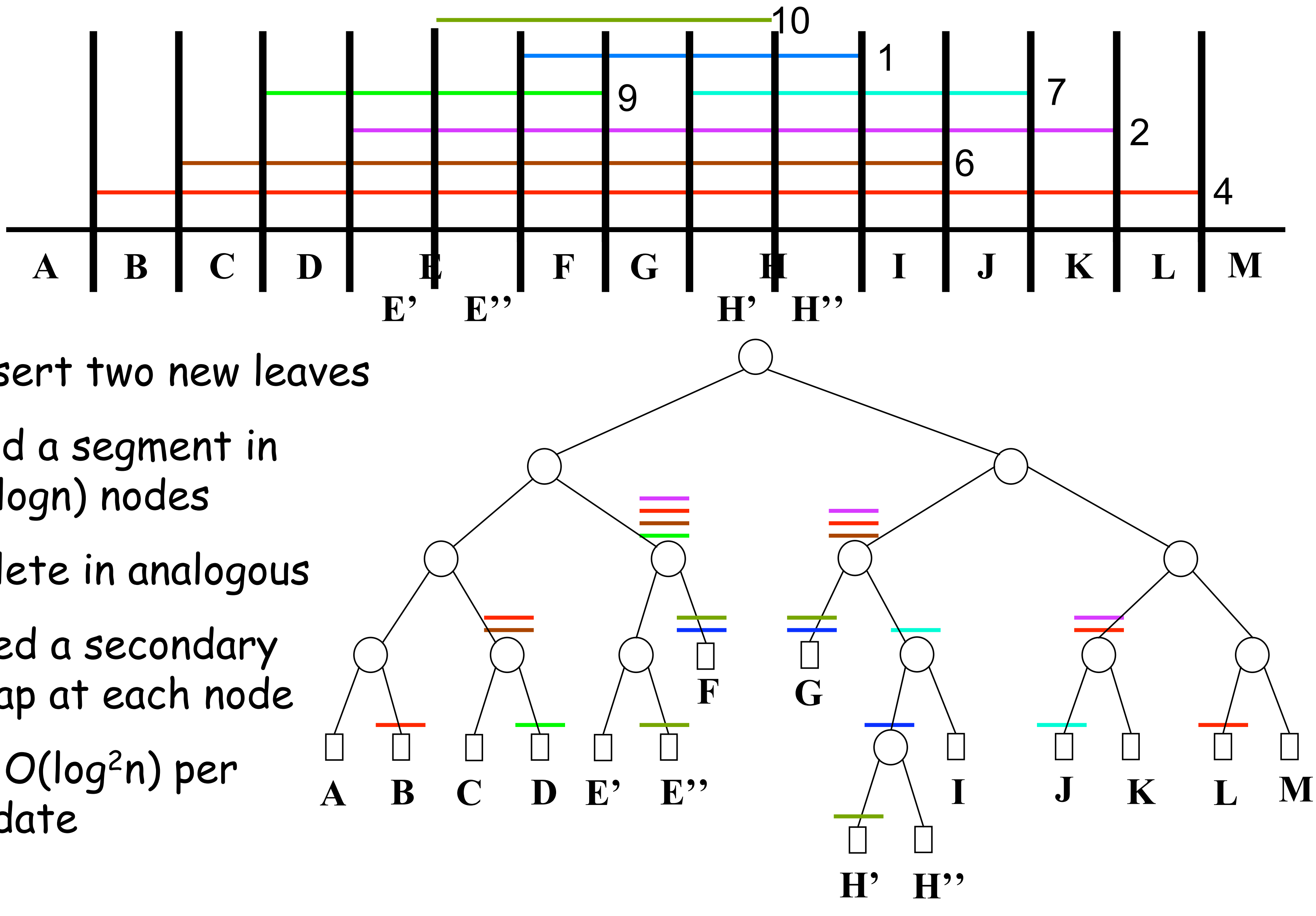
Insert two new leaves

Add a segment in O(logn) nodes

delete in analogous

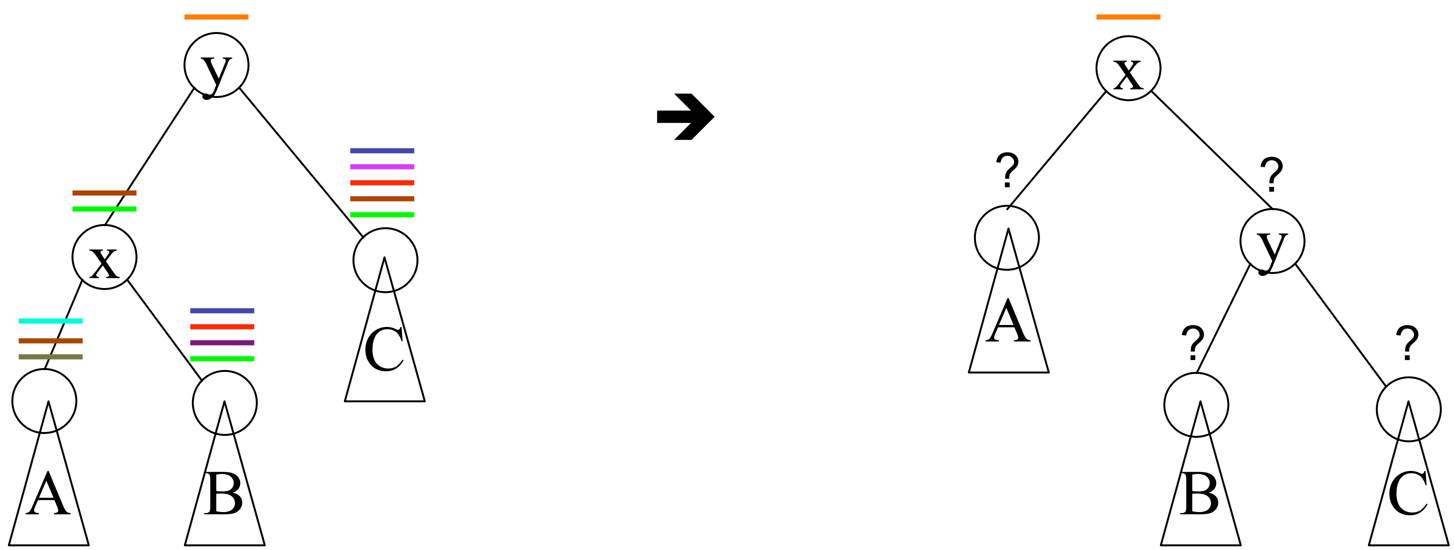
need a secondary heap at each node

➔ $O(\log^2 n)$ per update



To rebalance we have to make rotations

We have to compute the segments which are mapped to the nodes around the point of rotation



To amortize away this work use weight balance trees (BB[α])

Summary: segment tree

Query	$O(\log(n))$
Insert	$O(\log^2(n))$
Delete	$O(\log^2(n))$

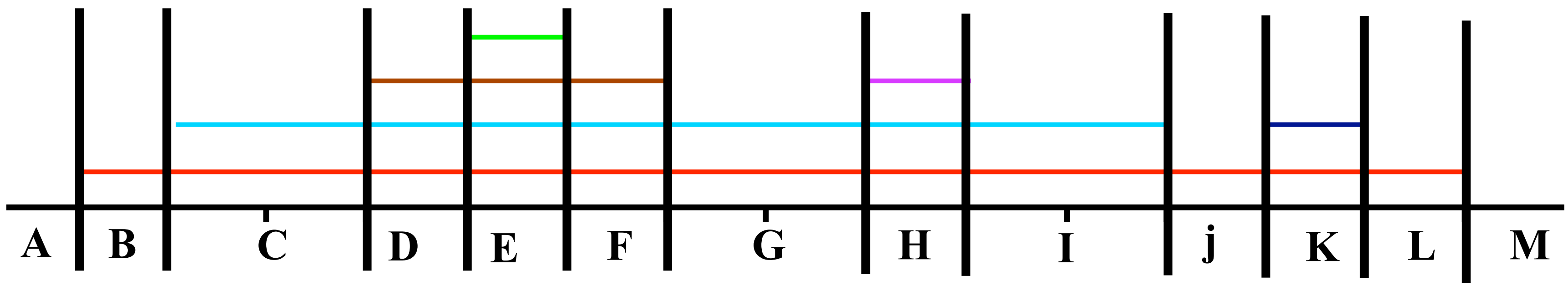
Results (1) (SWAT 2008, HK)

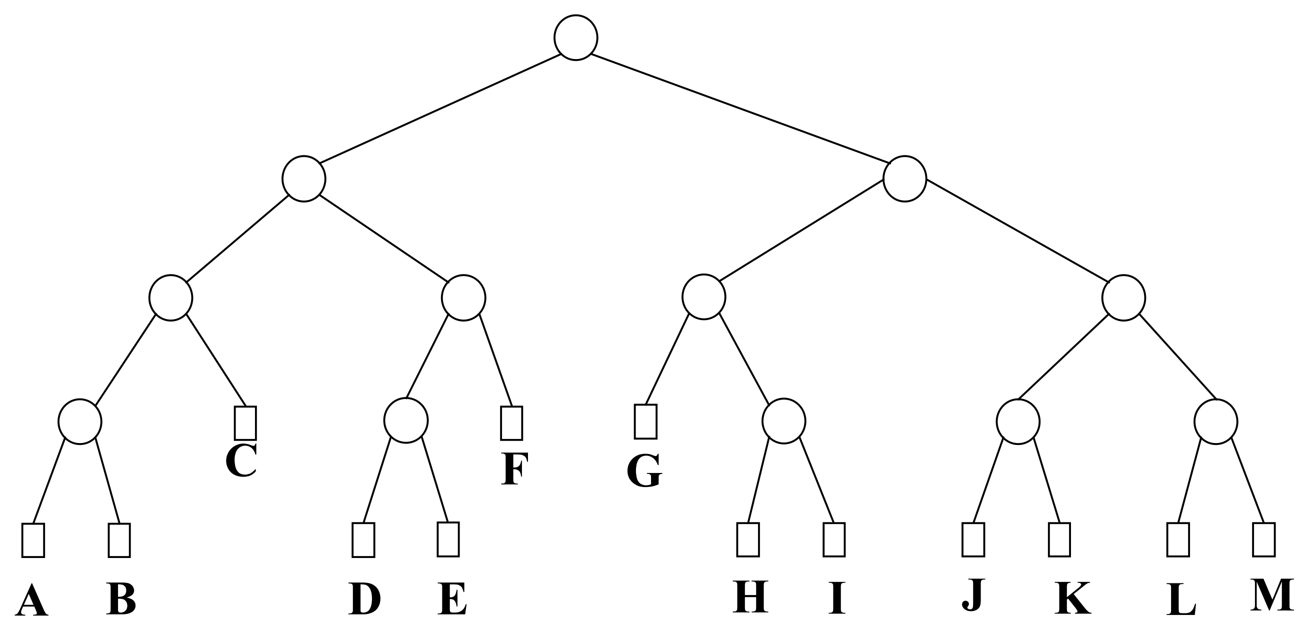
- **A very simple data structure for shortest segment (in a nested family)**
  $O(\log(n))$ time, and $O(\log_B(n))$ I/Os per op

- A data structure for longest prefix in a collection of arbitrary strings
  $O(\log(n) + |q|)$ time and $O(\log_B(n) + |q|/B)$ I/Os per op

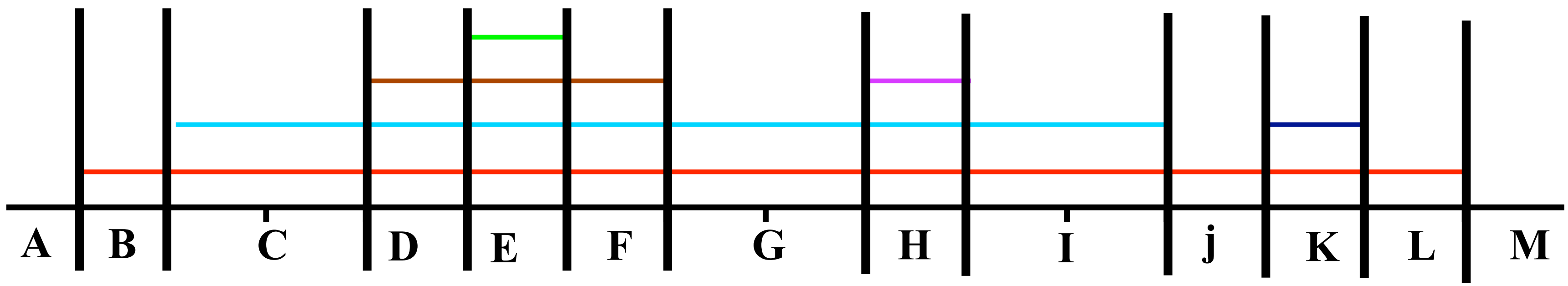
both take linear space

Shortest nested segment

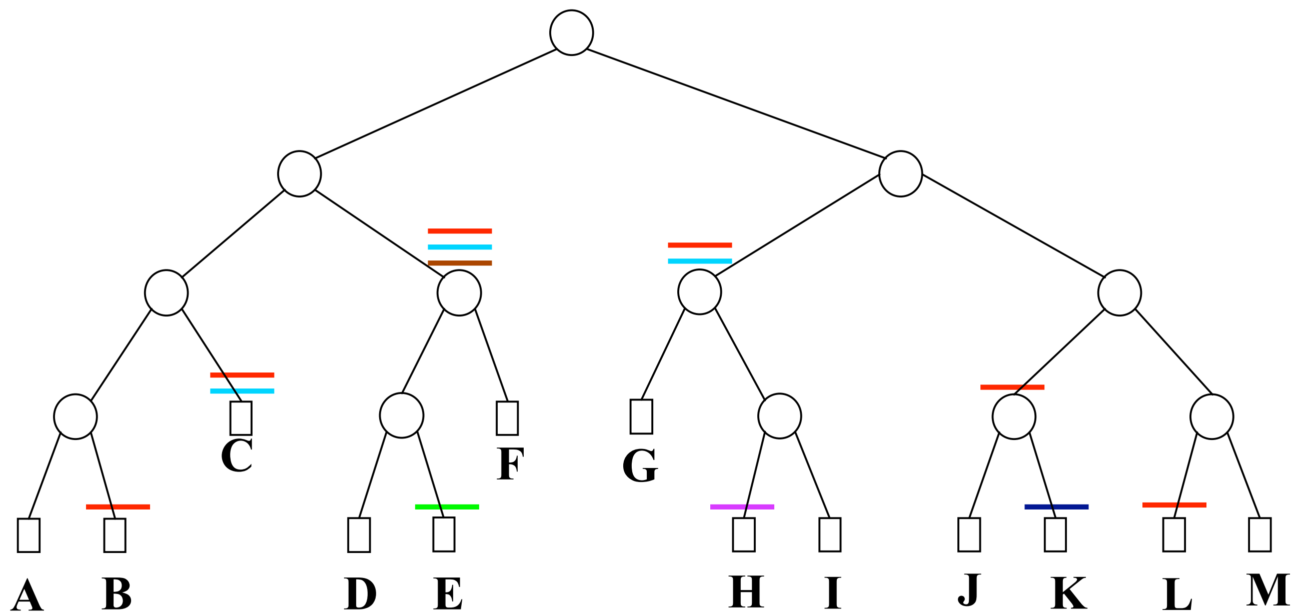




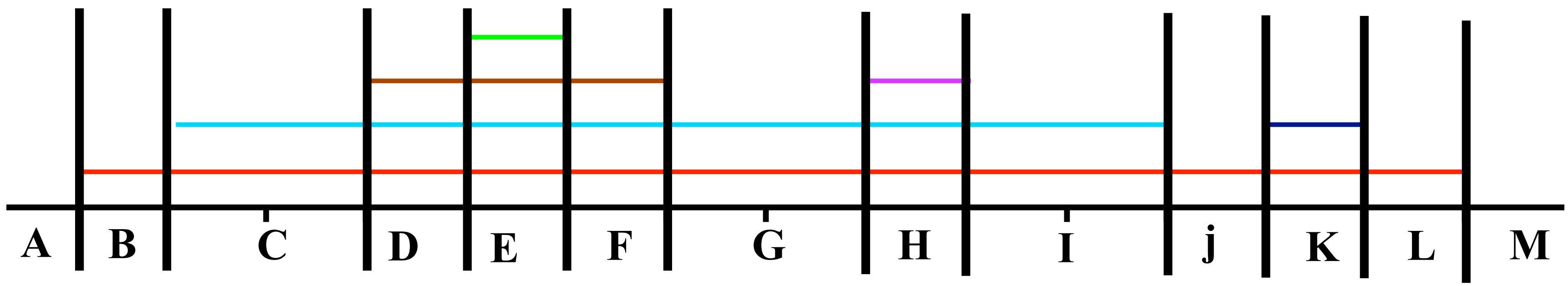
Shortest nested segment



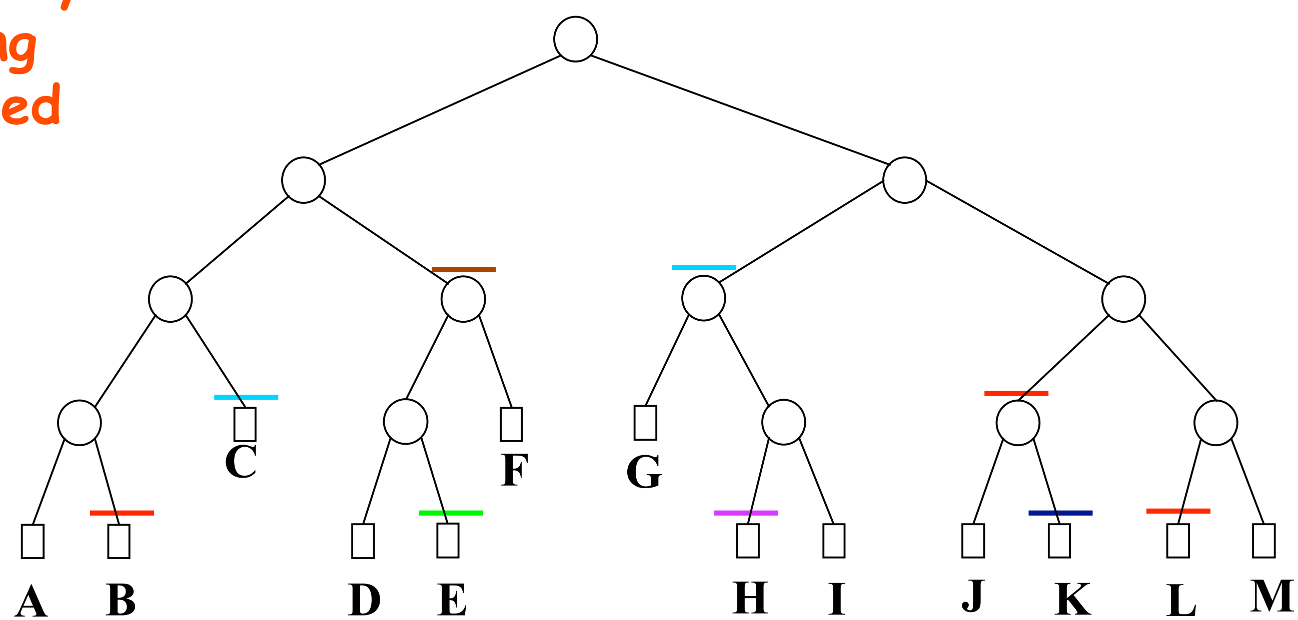
Use a segment tree
as before

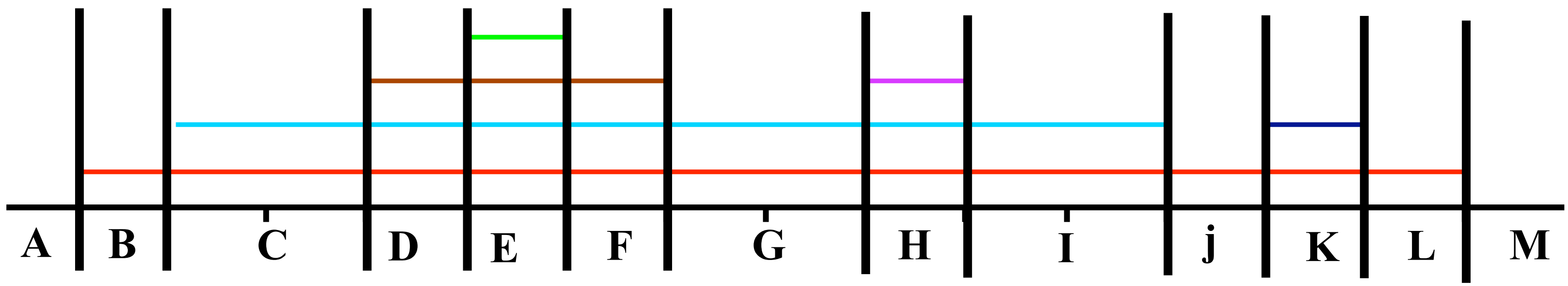


Main observation



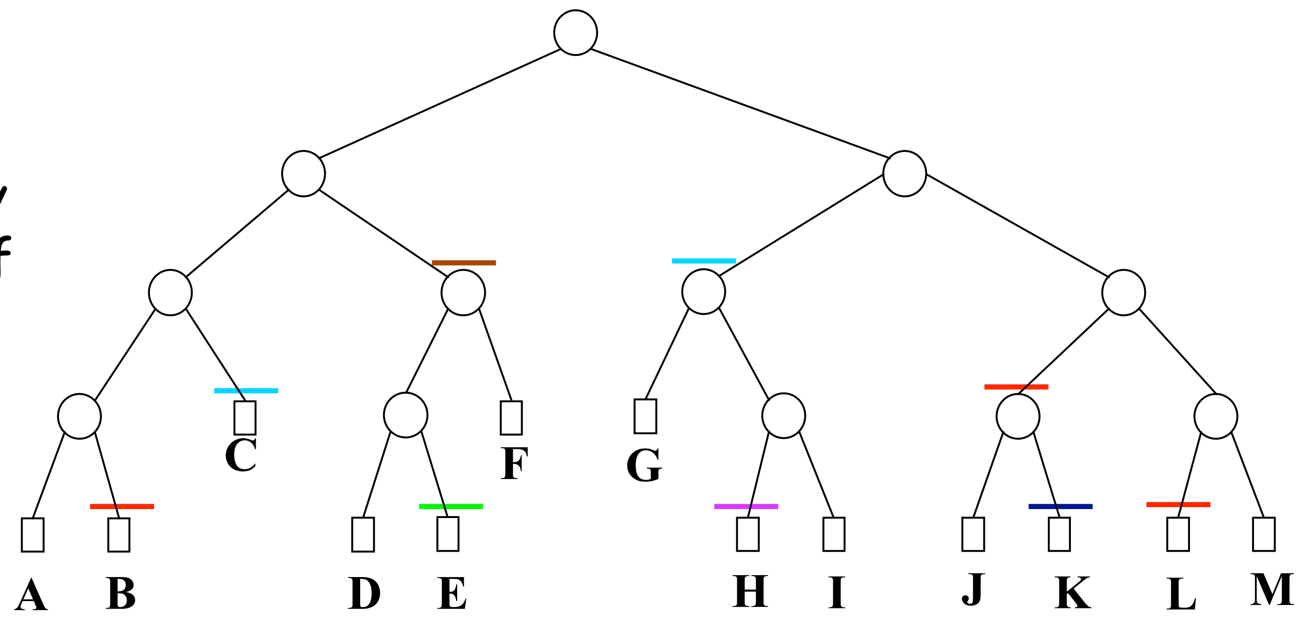
We can maintain only the shortest among all segments mapped to a node



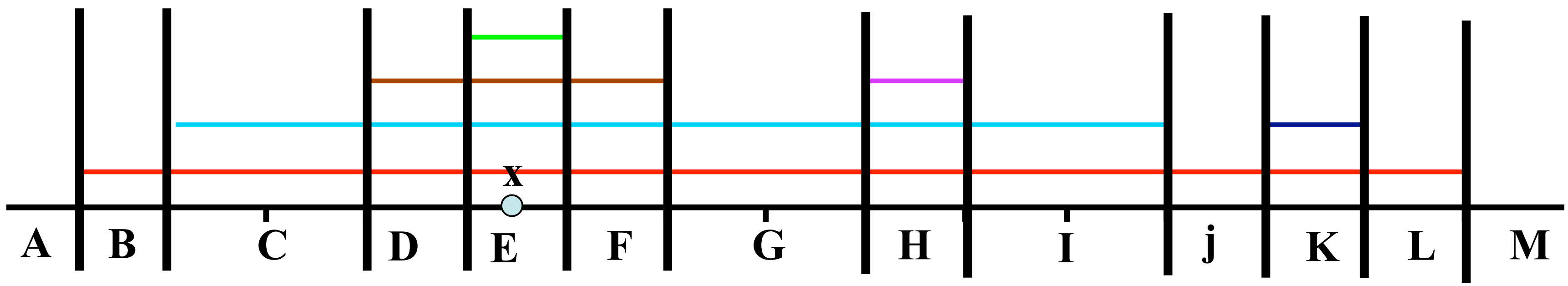


Observe (1) – any segment appears somewhere

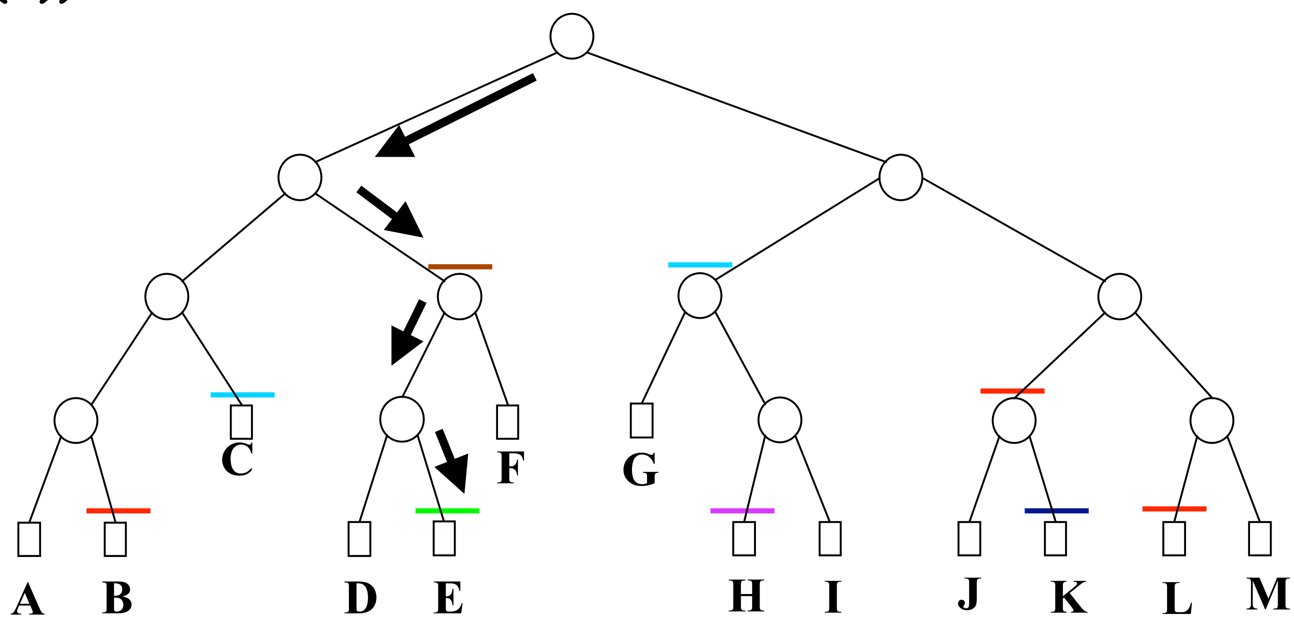
Observe (2) – Only one among a pair of siblings has a segment



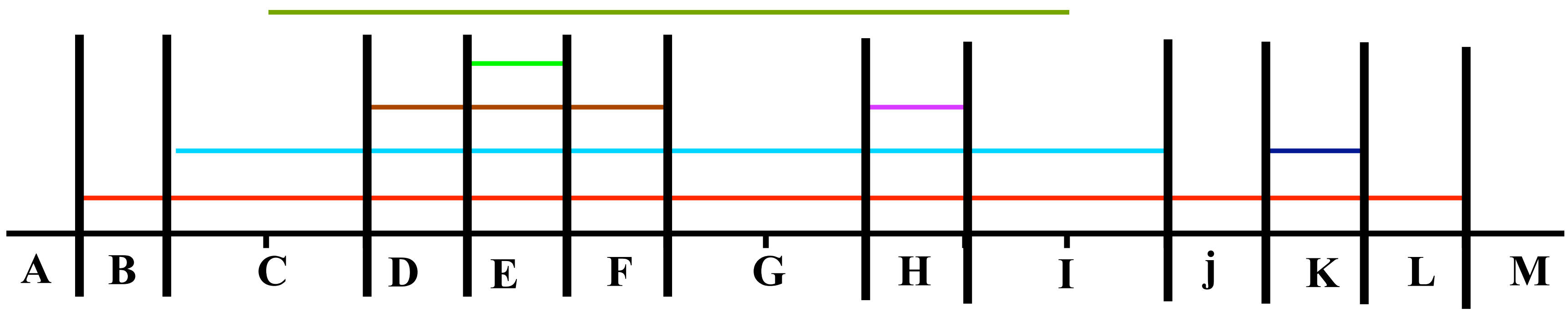
Query

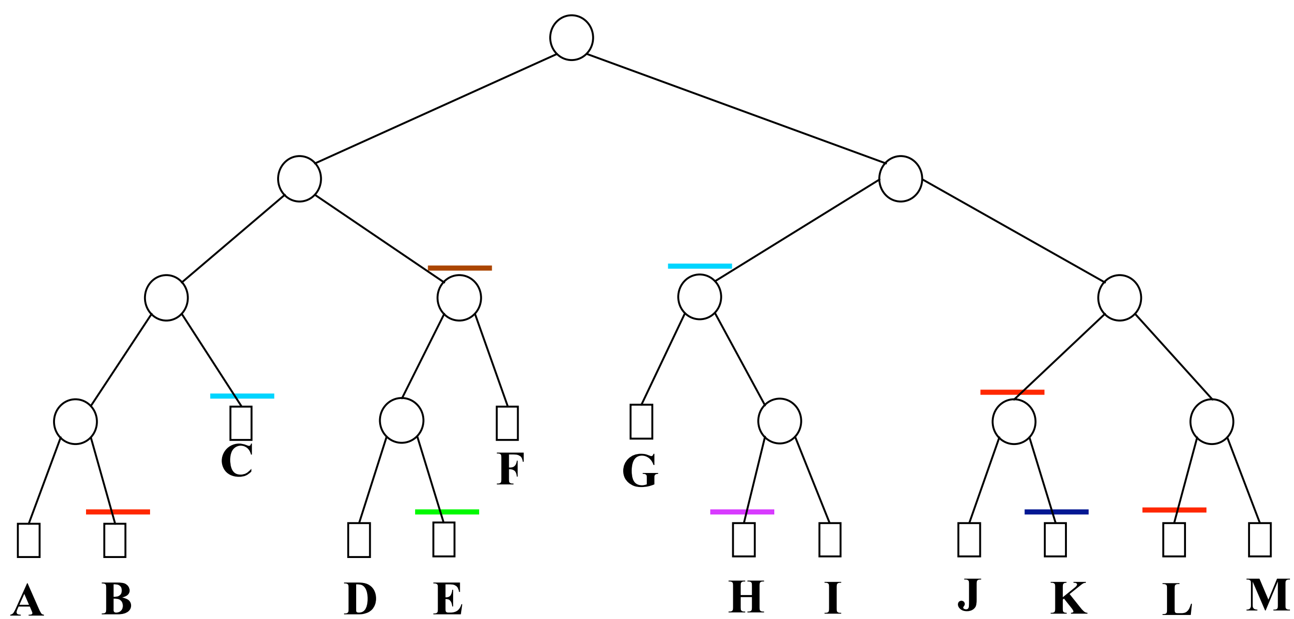


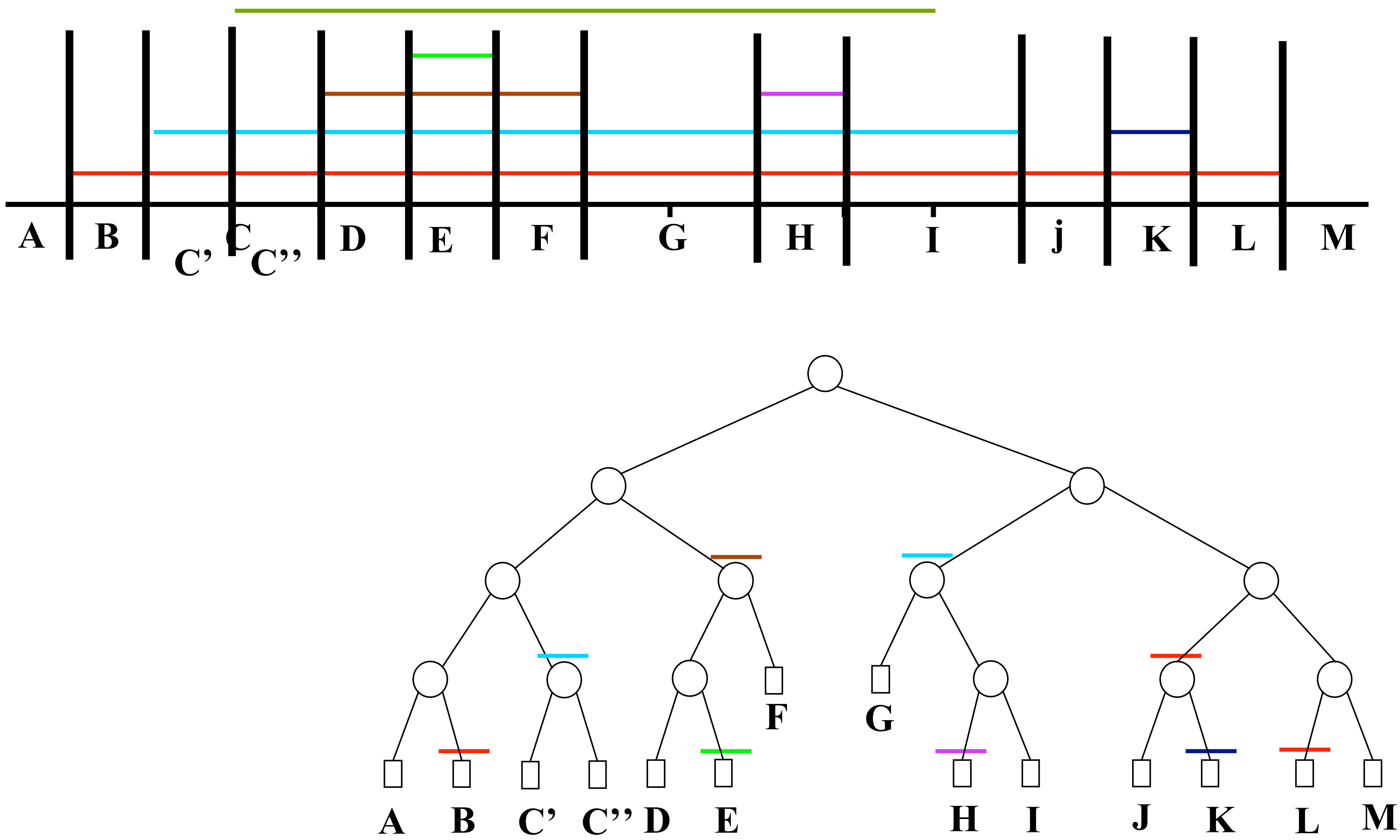
As before in O(log(n)) time



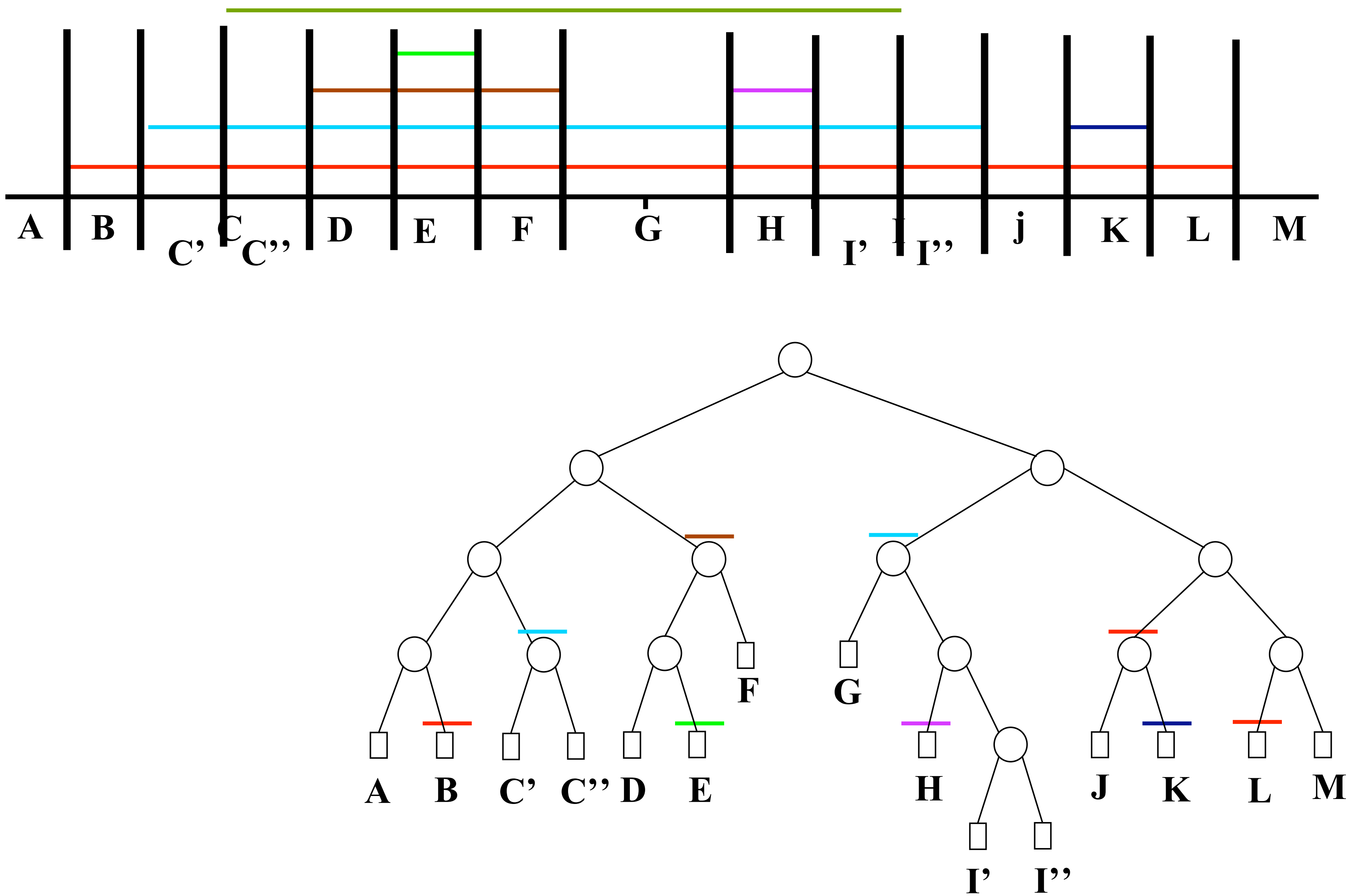
Insert



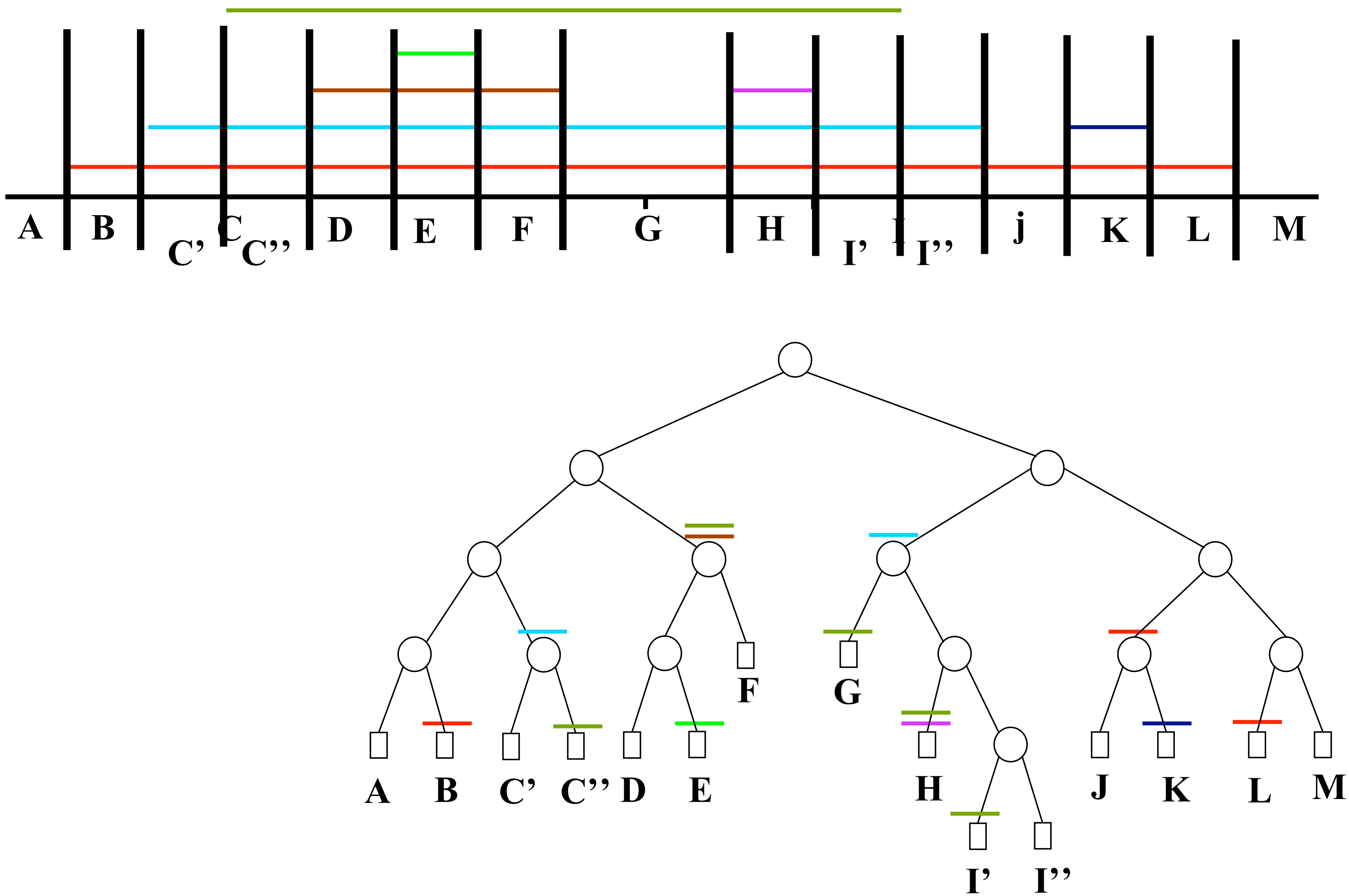


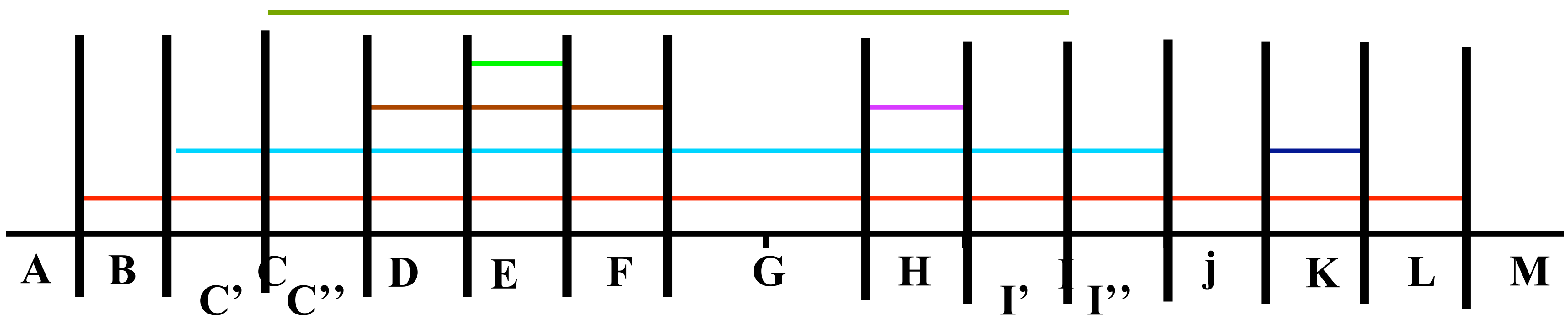


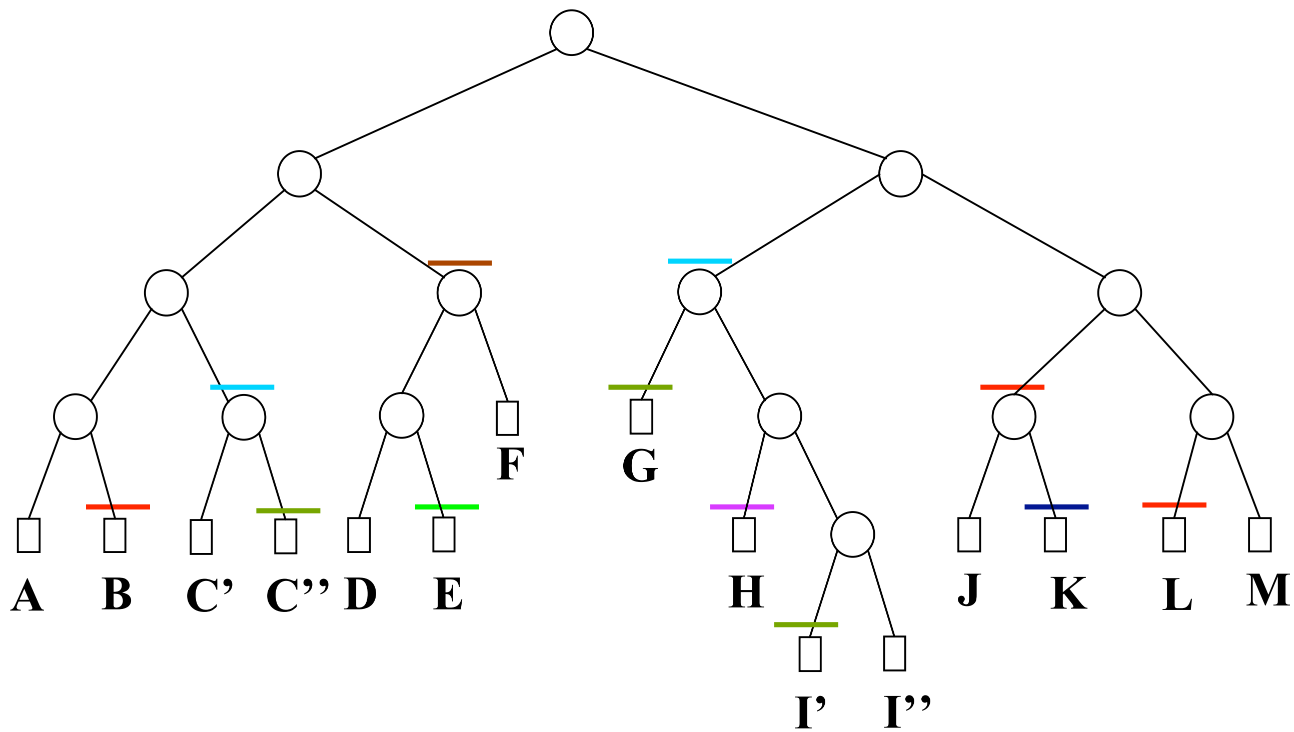
A
B
C'
C
C''
D
E
F
G
H
I
j
K
L
M
F
G
A
B
C'
C''
D
E
H
I
J
K
L
M



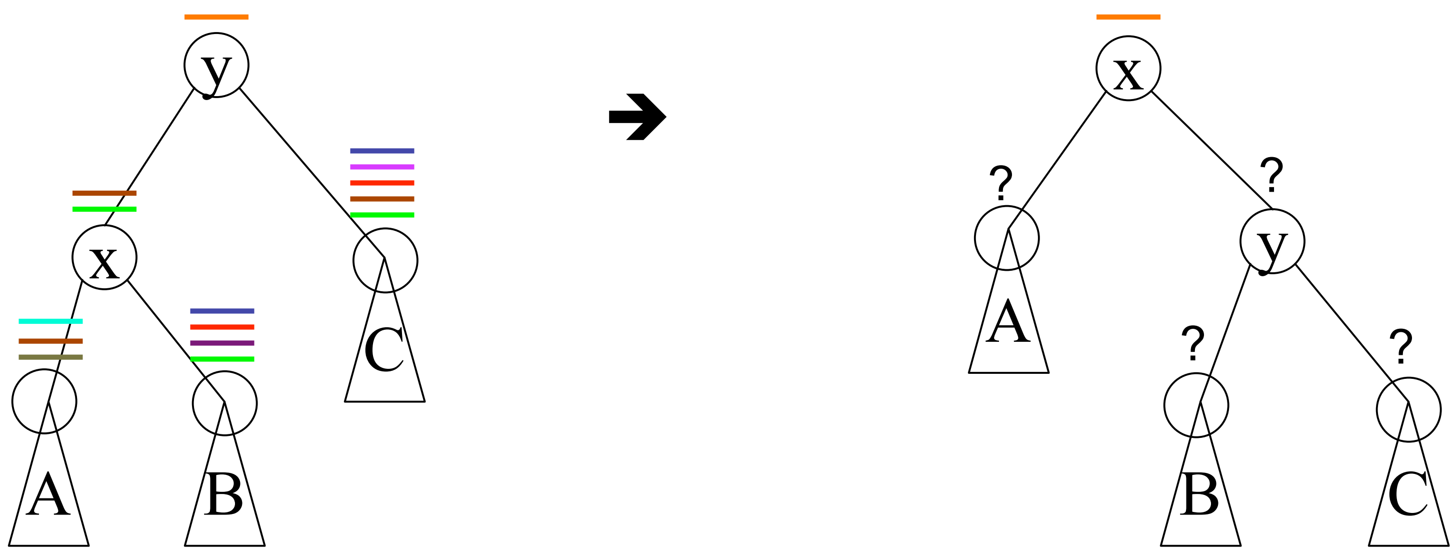
A
B
C'
C
C''
D
E
F
G
H
I'
I
I''
j
K
L
M
A
B
C'
C''
D
E
F
G
H
I'
I''
J
K
L
M


A
B
C'
C
C''
D
E
F
G
H
I'
I
I''
j
K
L
M
F
G
A
B
C'
C''
D
E
H
J
K
L
M
I'
I''

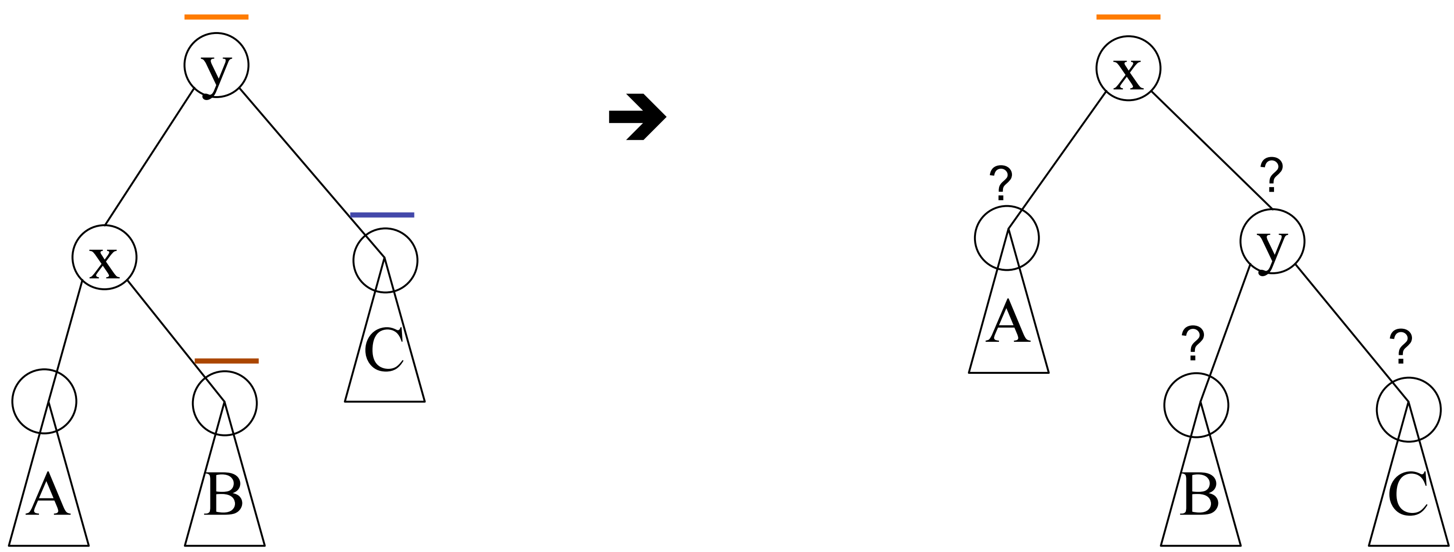

A
B
C'
C
C''
D
E
F
G
H
I'
I
I''
j
K
L
M


A
B
C'
C''
D
E
F
G
H
I'
I''
J
K
L
M

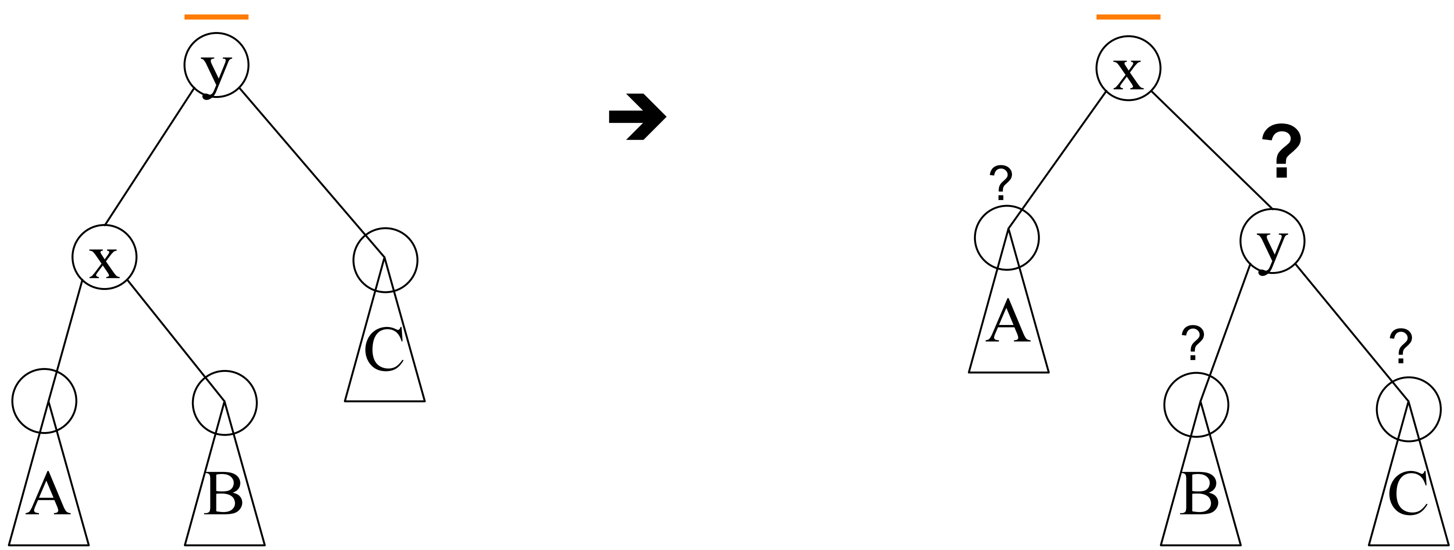
Rotations ?

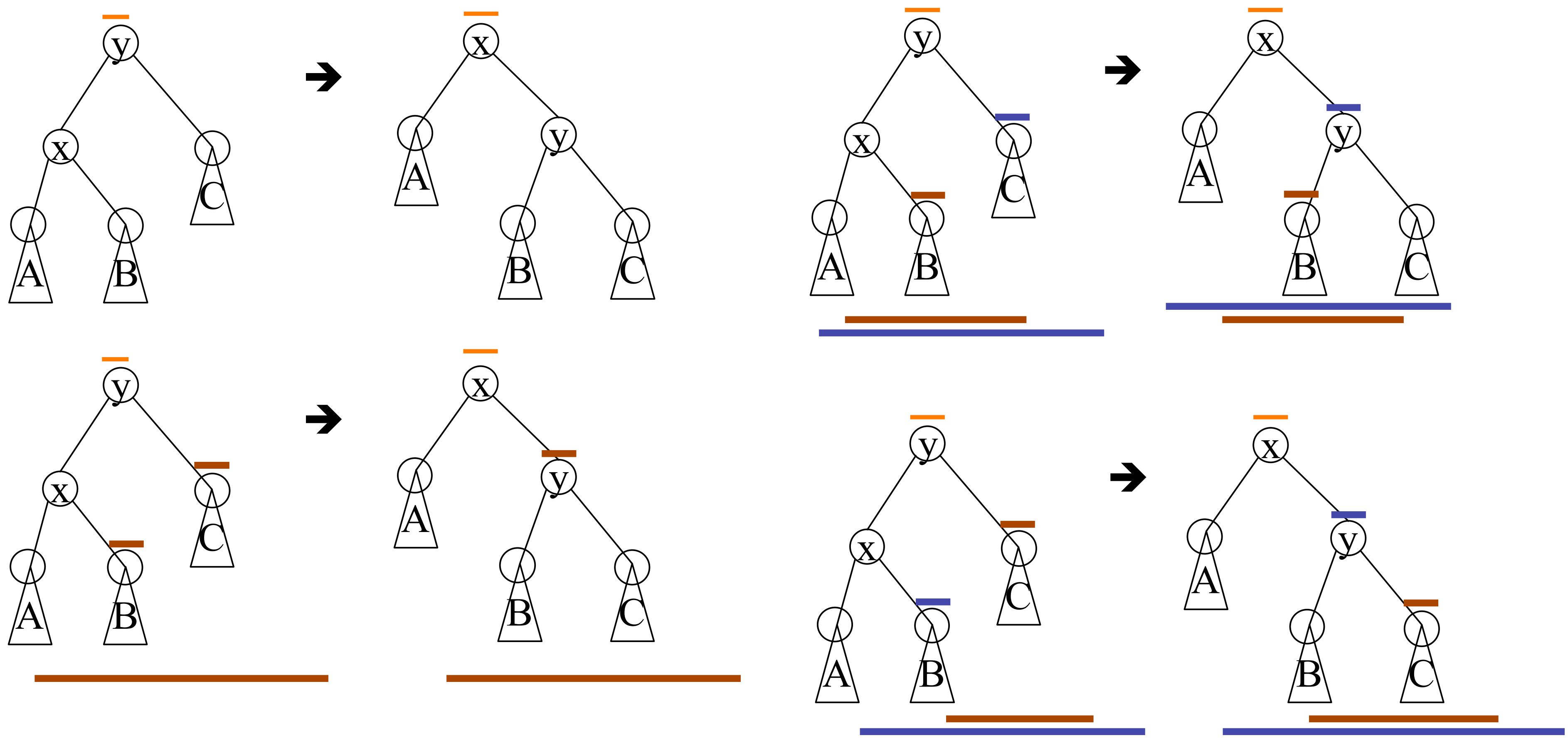


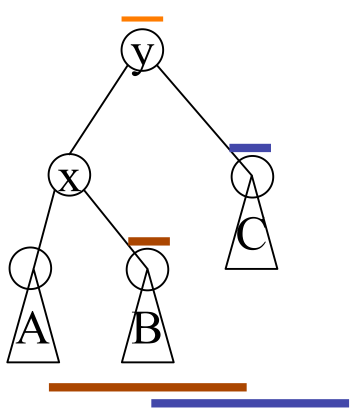
Shortest Nested Segments - Rotations



Shortest Nested Segments - Rotations

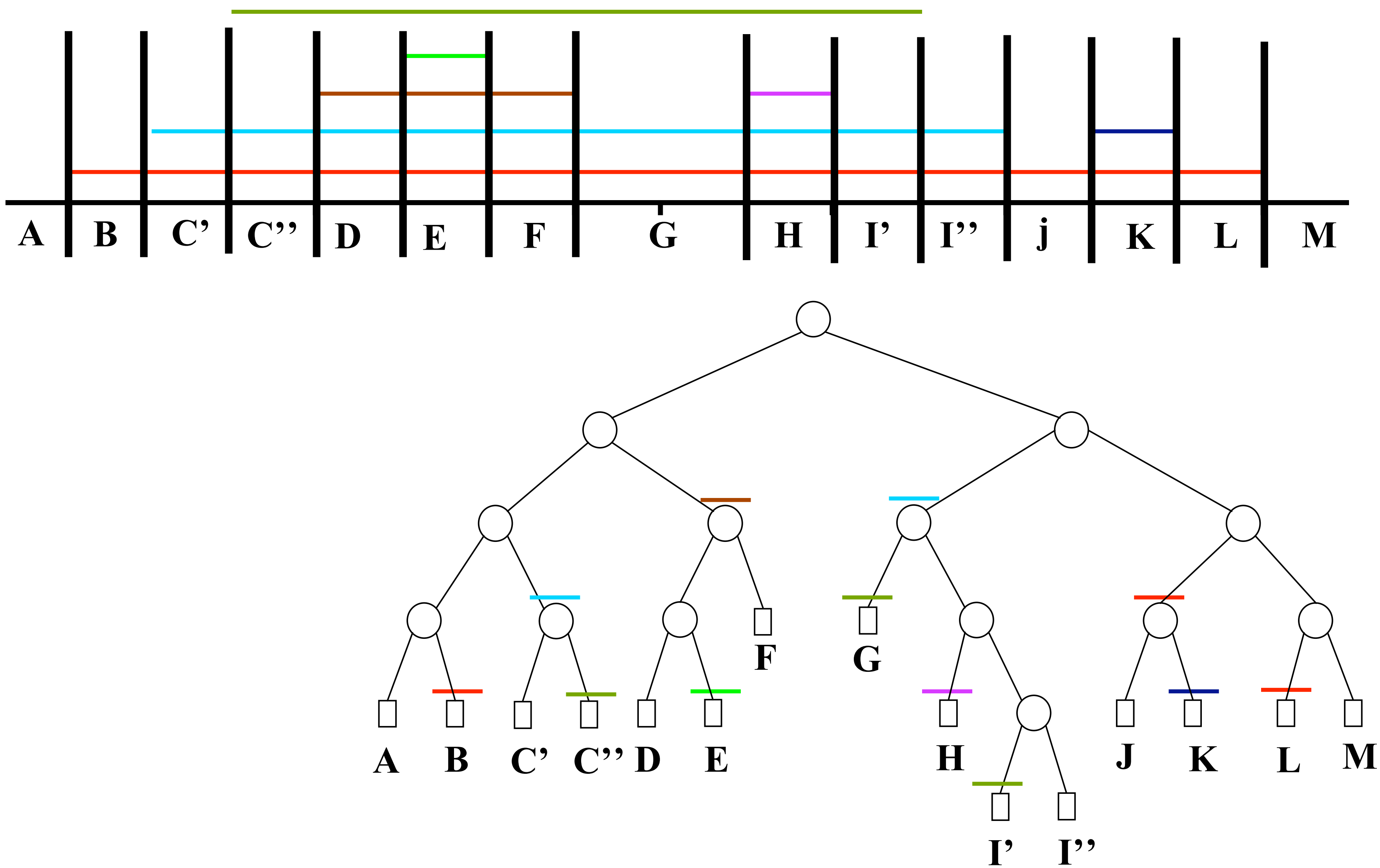




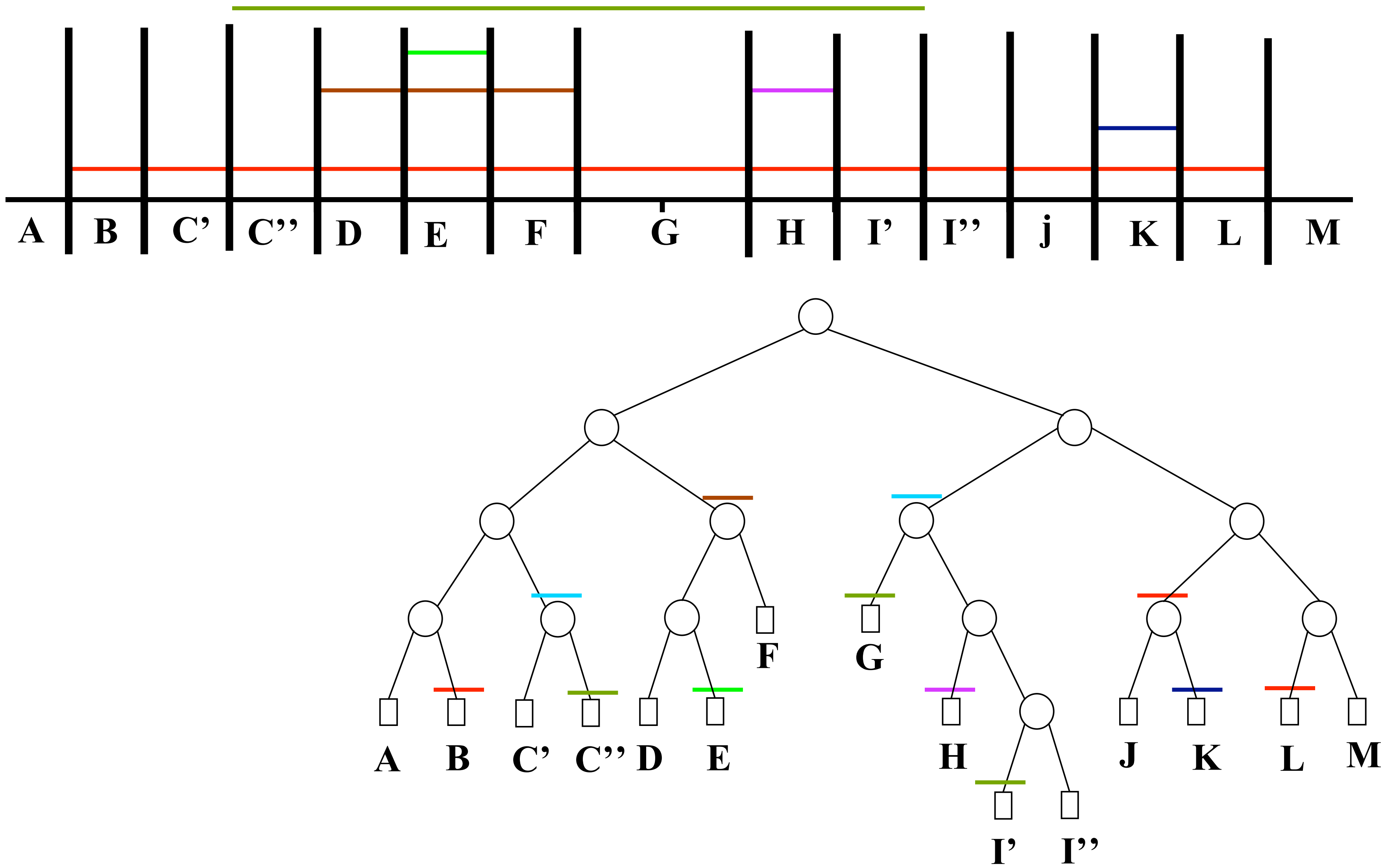


Impossible

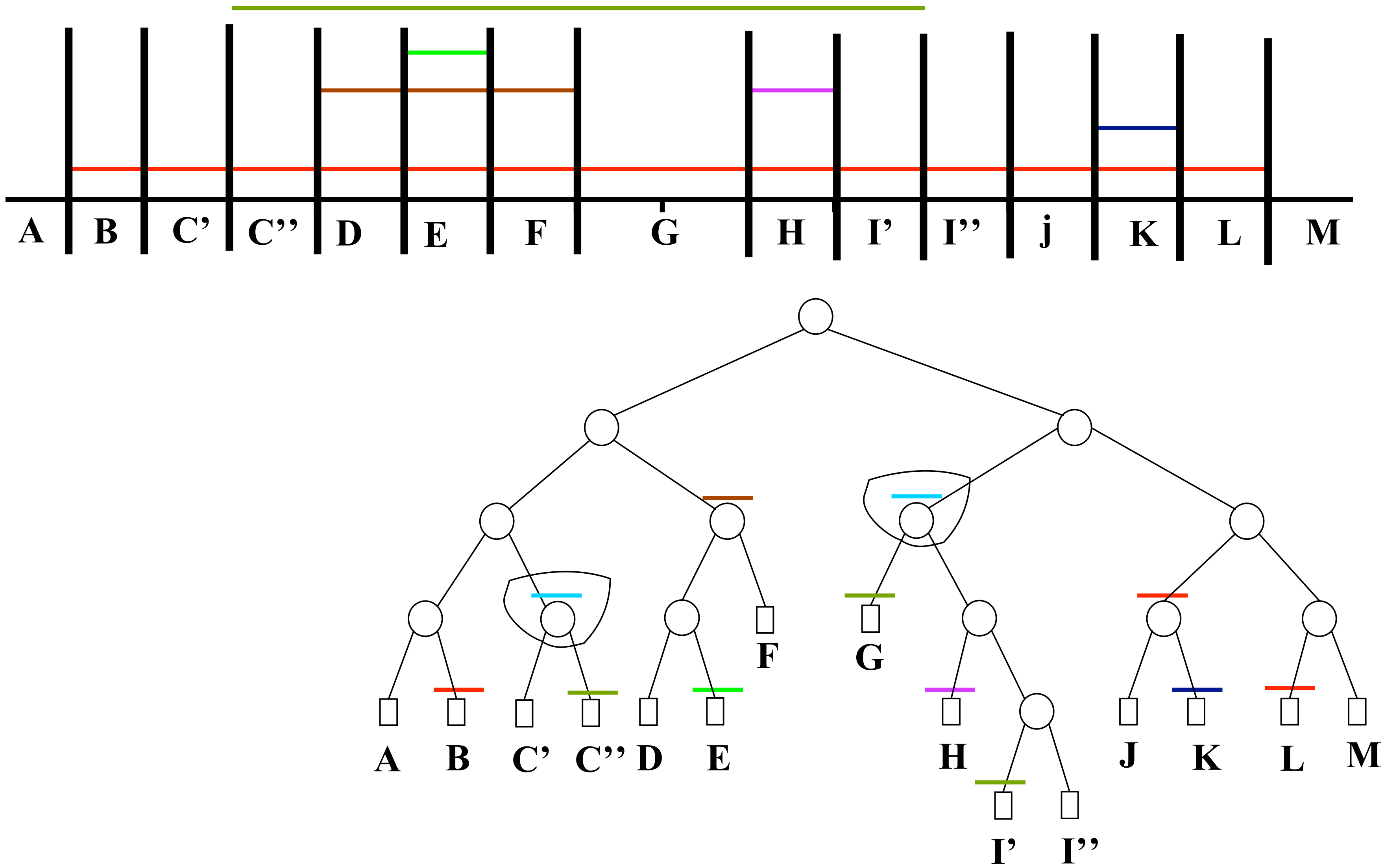
Delete



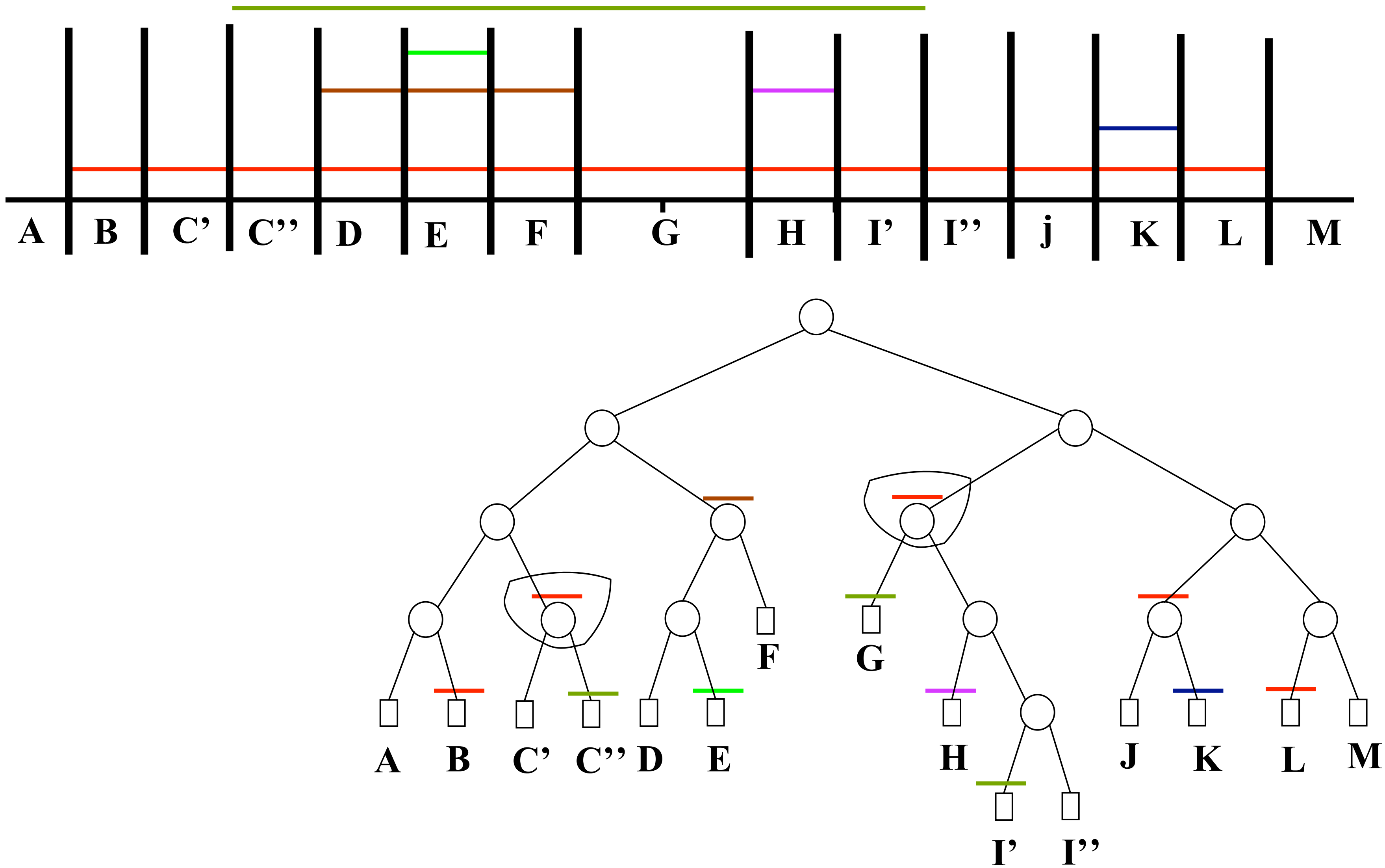
Delete



Delete



Delete



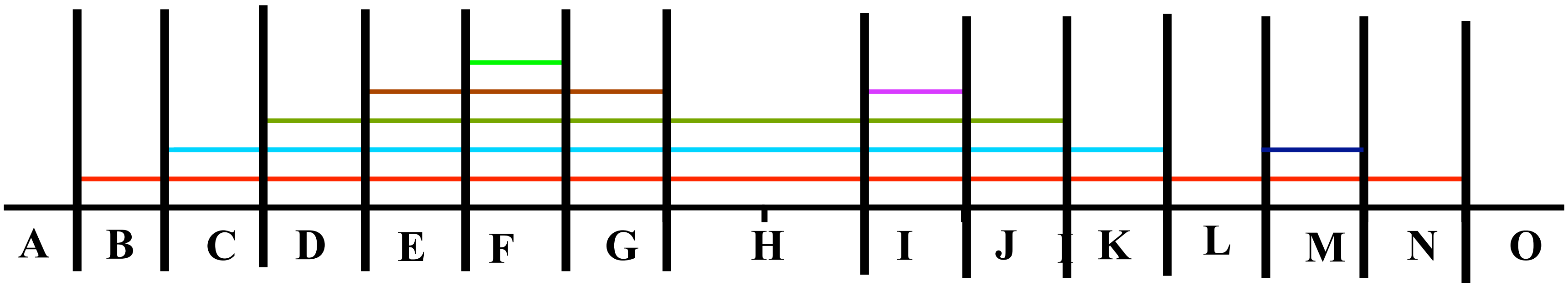
Results (1) (SWAT 2008, HK)

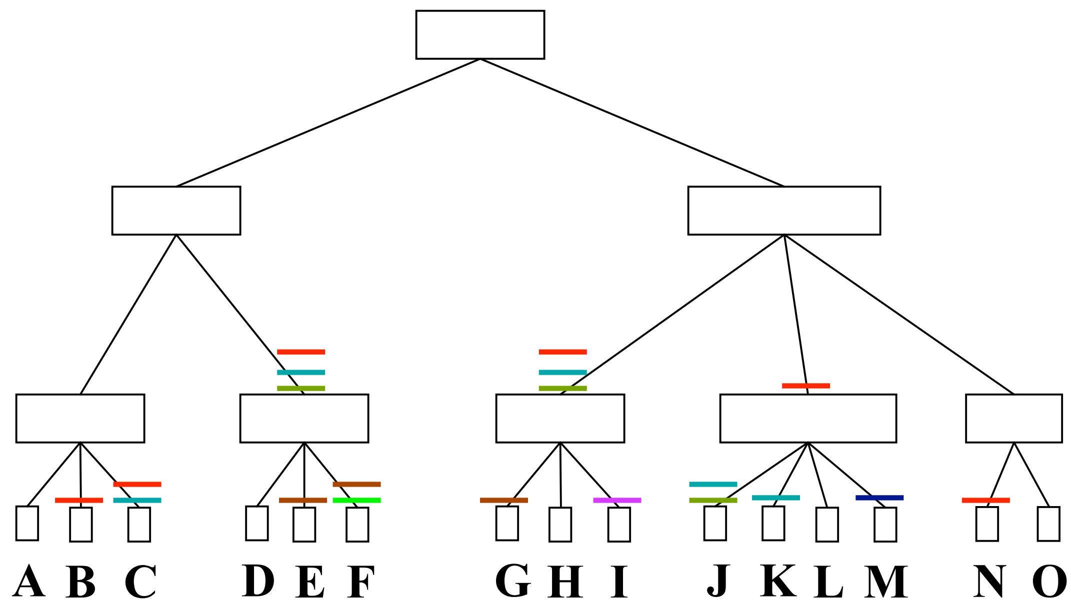
- A very simple data structure for shortest segment (in a nested family)
  $O(\log(n))$ time, **and $O(\log_B(n))$ I/Os per op**

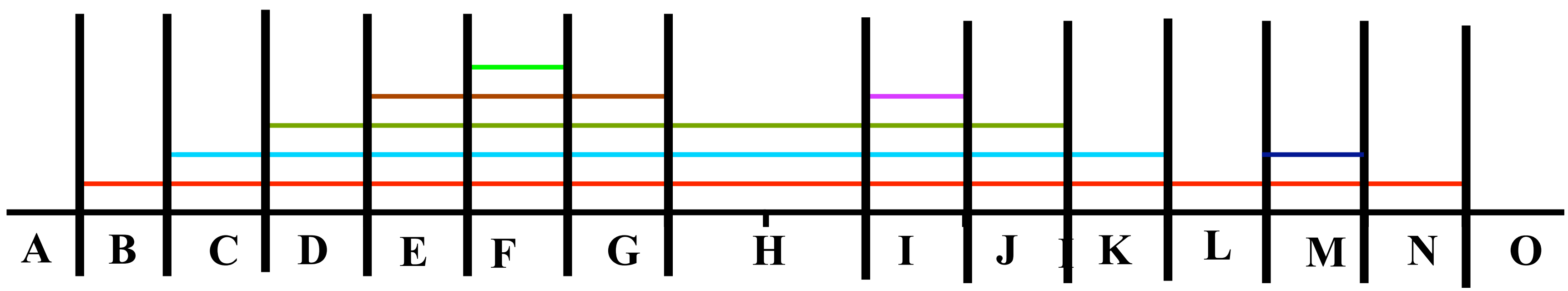
- A data structure for longest prefix in a collection of arbitrary strings
  $O(\log(n) + |q|)$ time and $O(\log_B(n) + |q|/B)$ I/Os per op

both take linear space

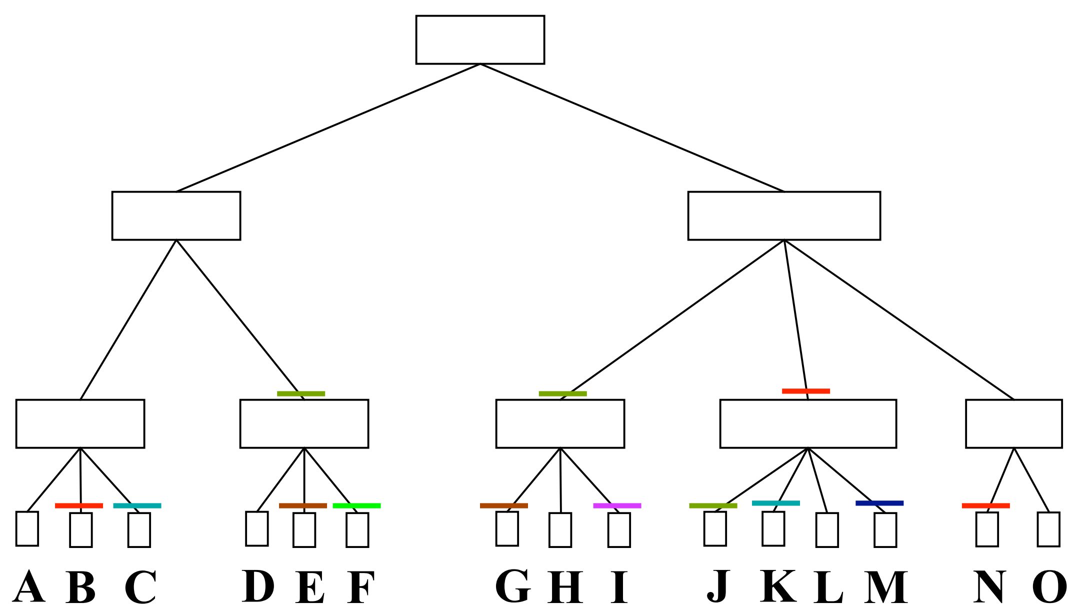
Use the B-tree as a segment tree



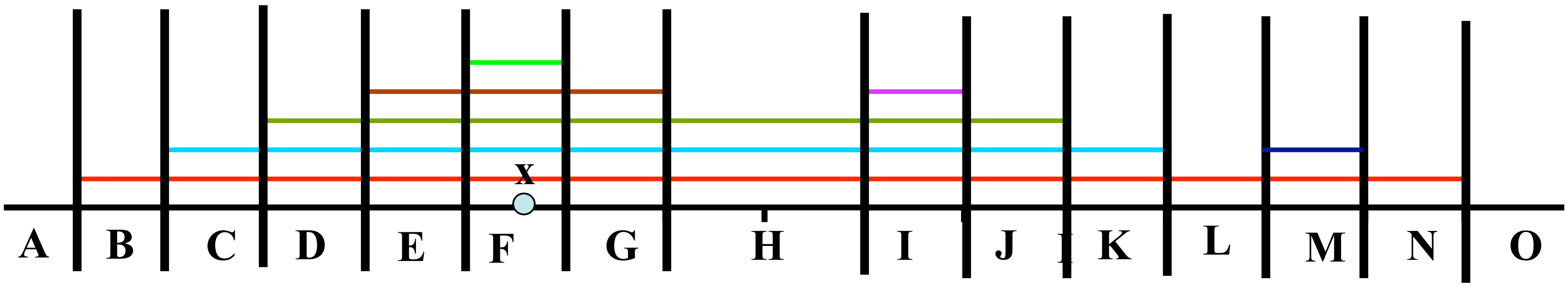




Keep only the shortest at each node

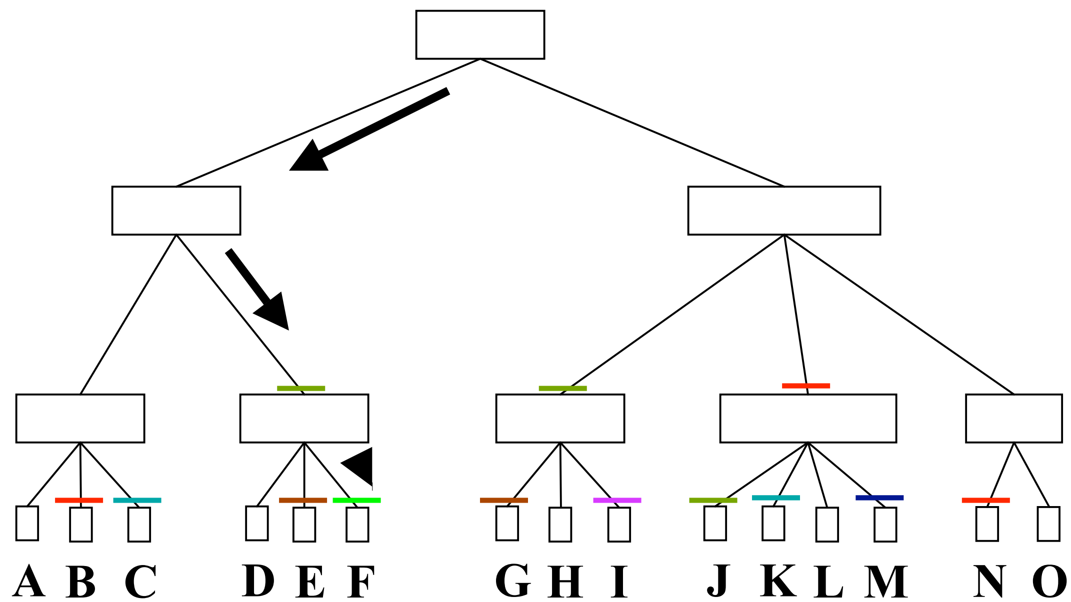


Query

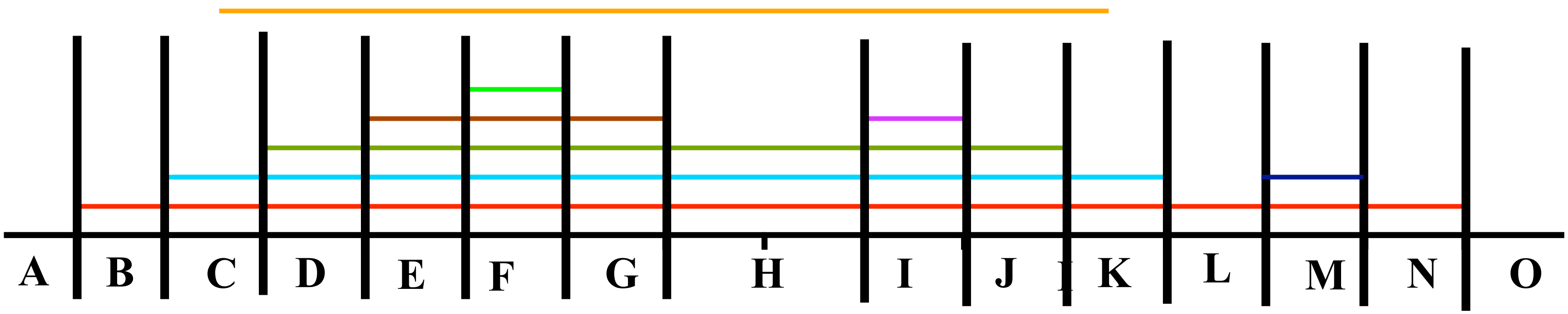


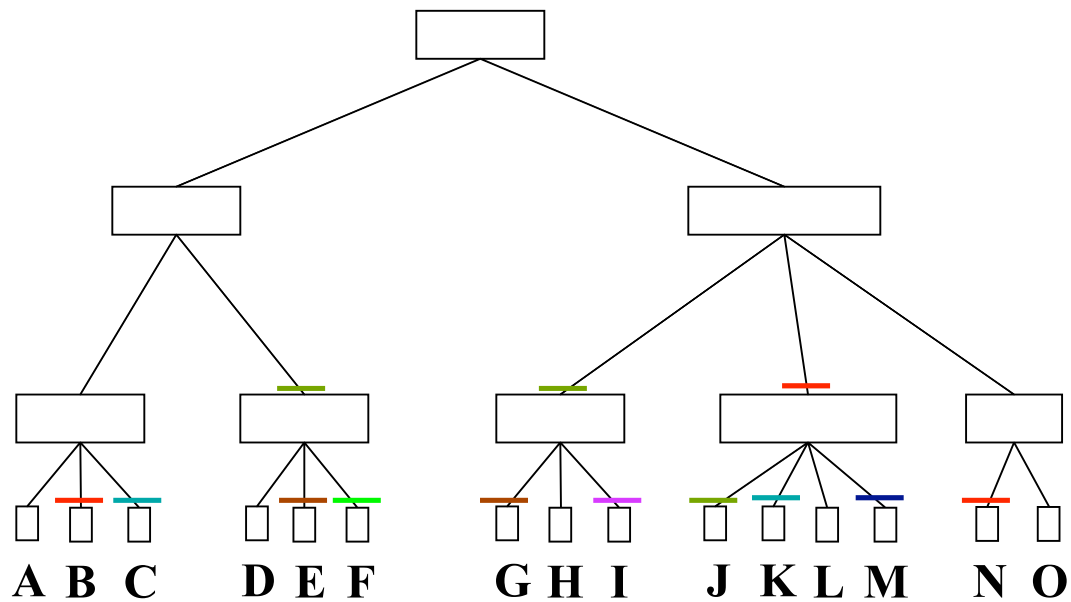
Same as before.

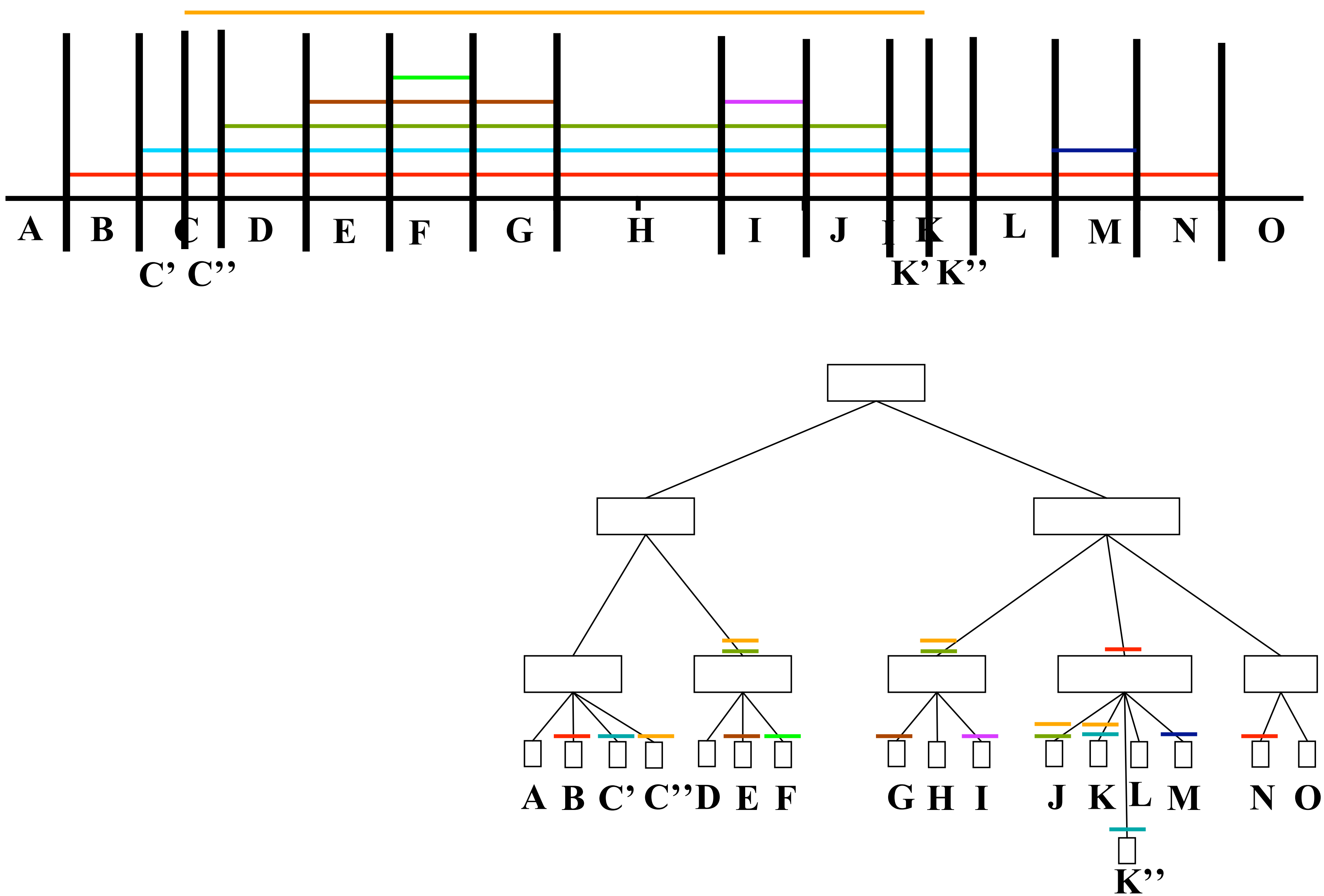
O($\log_B(n)$) I/Os.



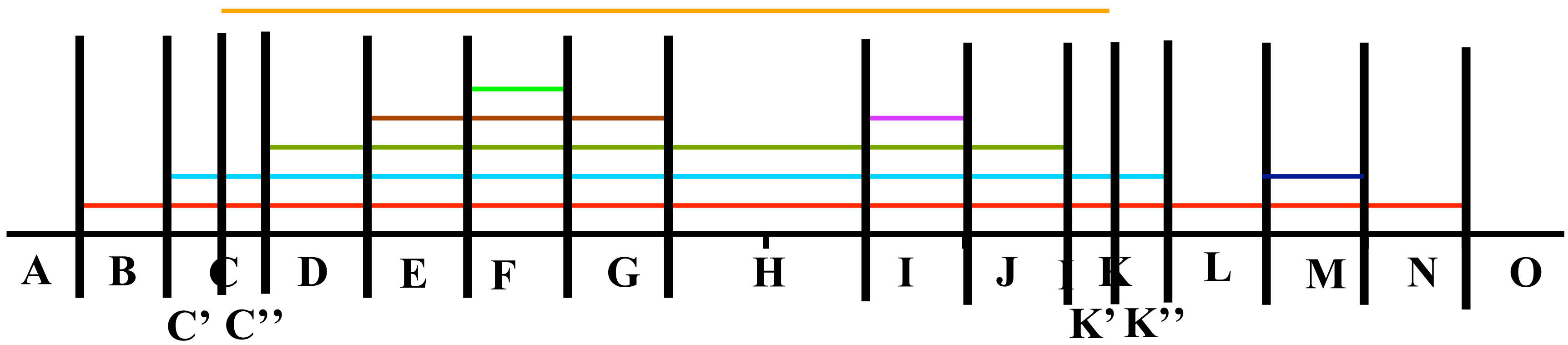
Insert



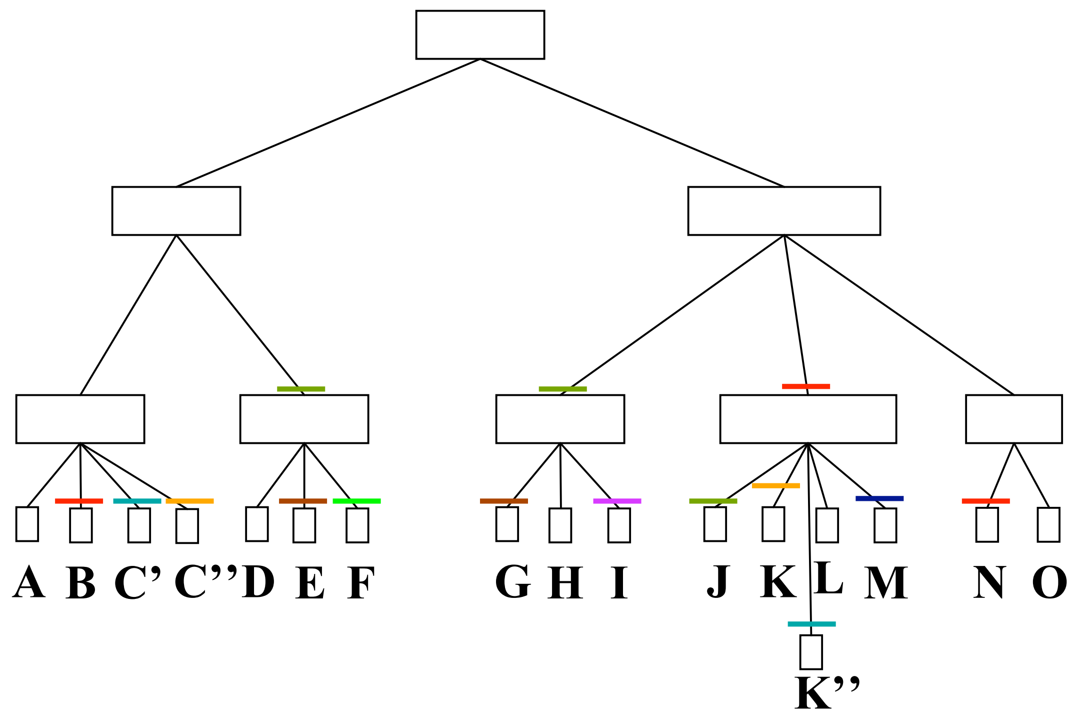




A B C D E F G H I J J K L M N O
C' C''
K' K''
A B C' C''D E F G H I J K L M N O
K''

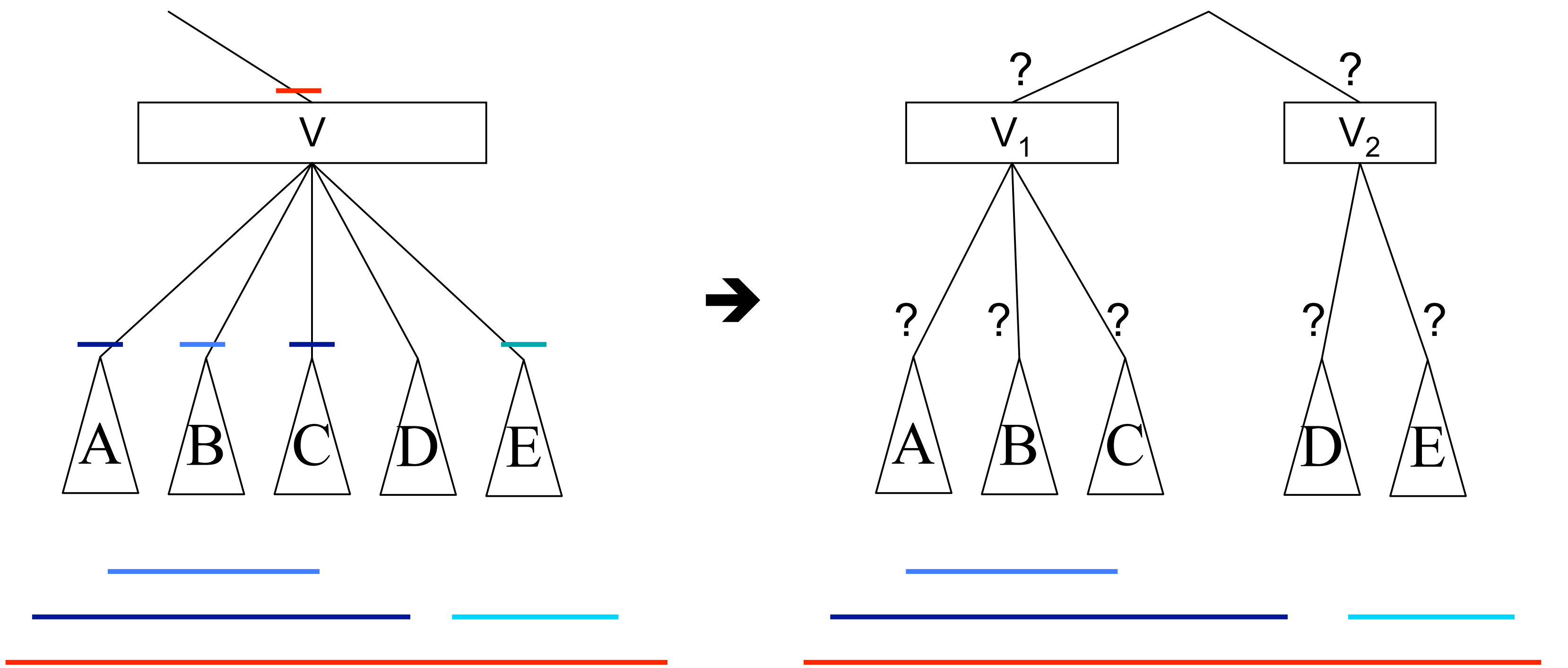


A B C D E F G H I J J K L M N O
C' C''
K' K''

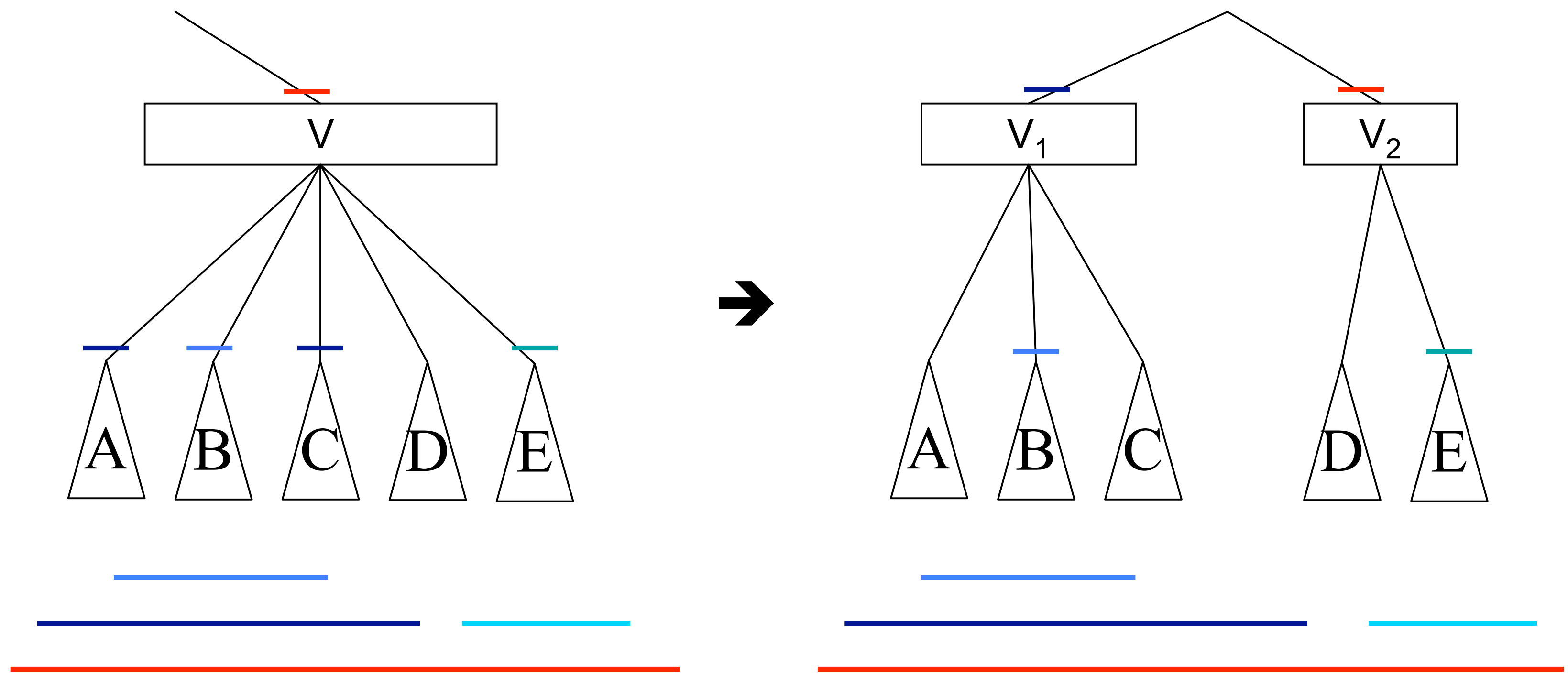


A B C' C'' D E F G H I J K L M N O
K''

Split/merge/borrow analogous to rotations



Split/merge/borrow analogous to rotations



Results (1) (SWAT 2008, HK)

- A very simple data structure for shortest segment (in a nested family)
  $O(\log(n))$ time, and $O(\log_B(n))$ I/Os per op

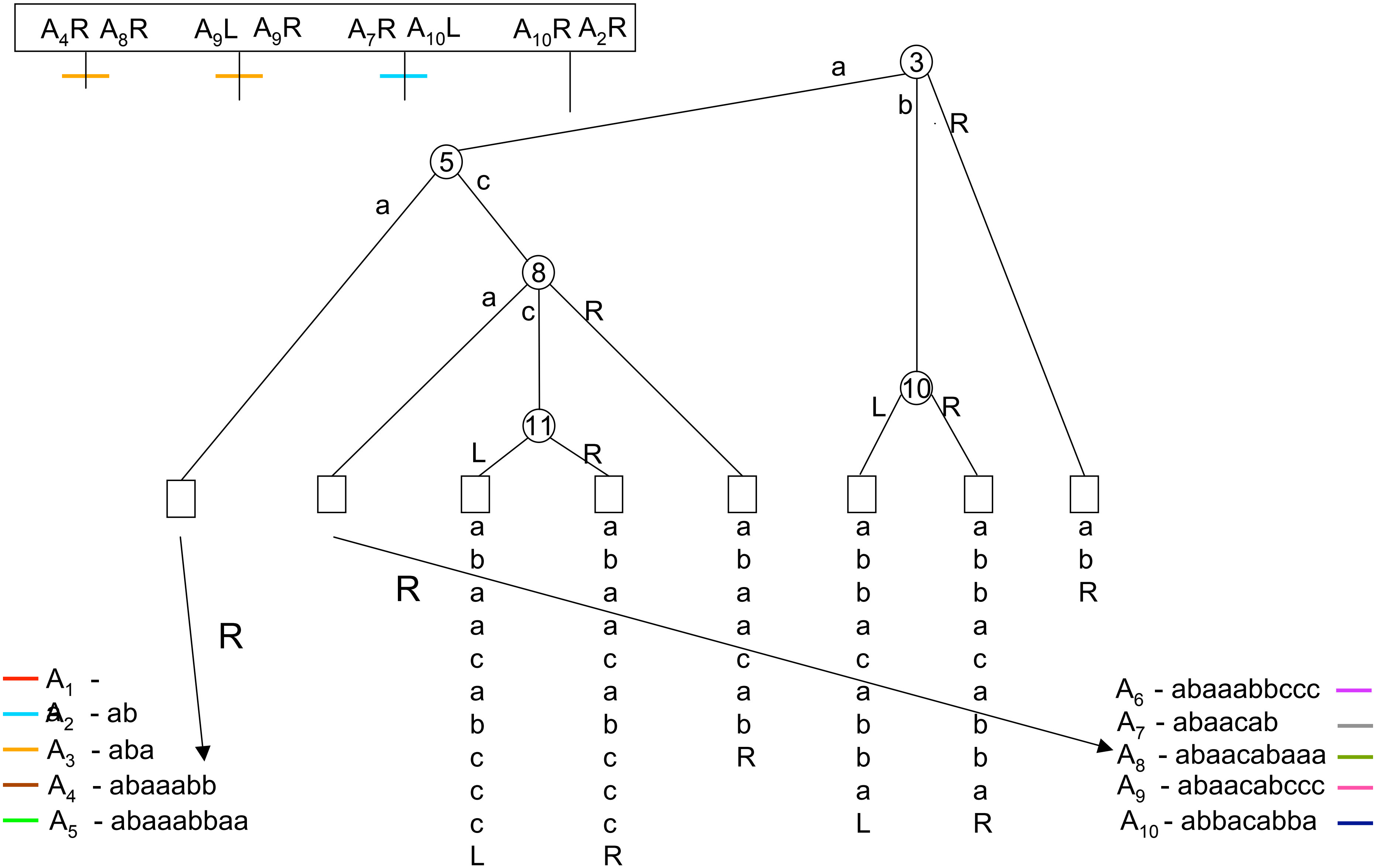
- **A data structure for longest prefix in a collection of arbitrary strings**
  $O(\log(n) + |q|)$ time and $O(\log_B(n) + |q|/B)$ I/Os
  per op

both take linear space

Combine

- Combine with the string B-tree of Ferragina and Grossi (JACM 99)

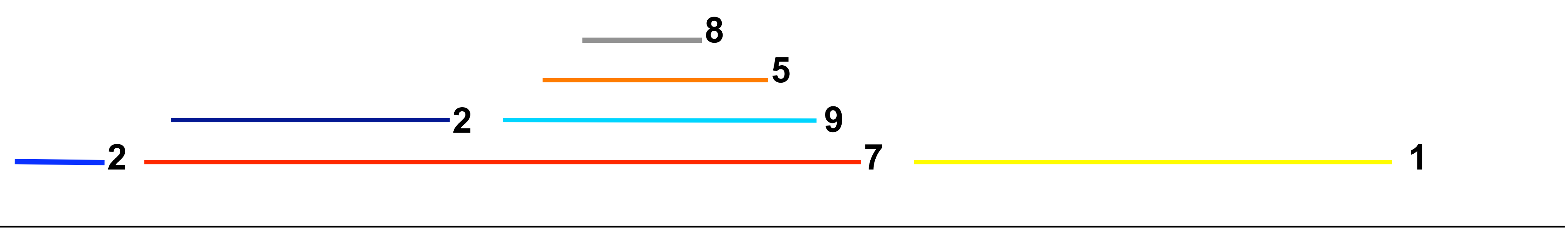
A Patricia trie of the keys



Results (2) (STOC 2003,KMT)

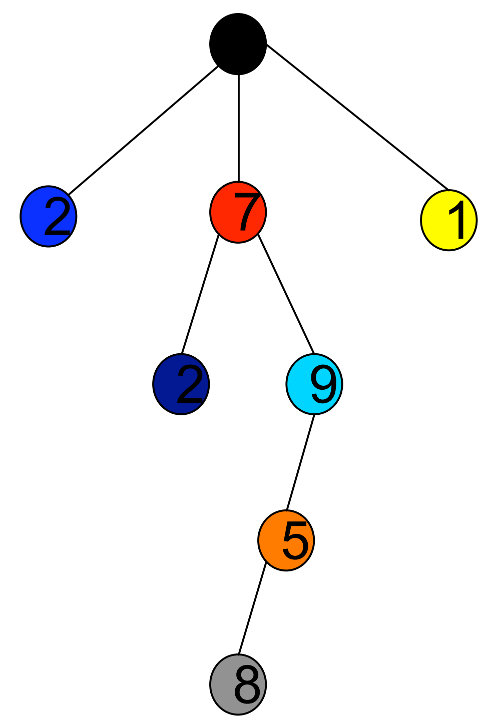
- **A simple data structure for nested segments with priorities**
  $O(\log(n))$ time per op,
  $O(n)$ space (uses dynamic trees)

- A data structure for general segments $O(\log(n))$ time per query/insert but delete takes $O(\log(n)\log\log(n))$ time, $O(n\log\log(n))$ space

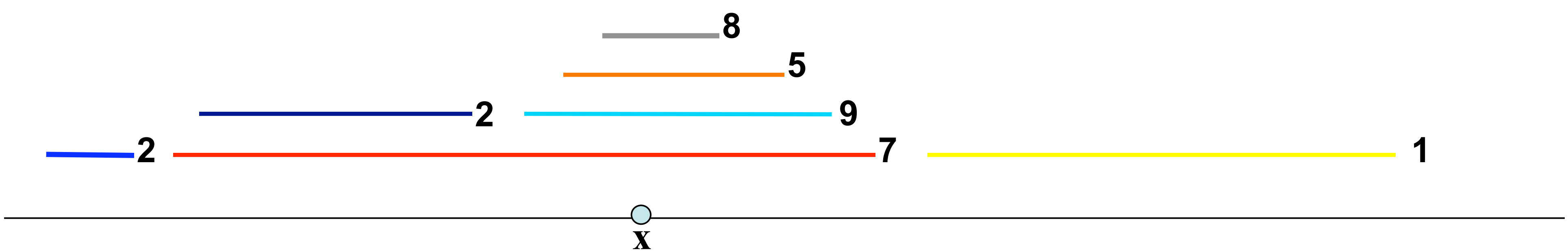


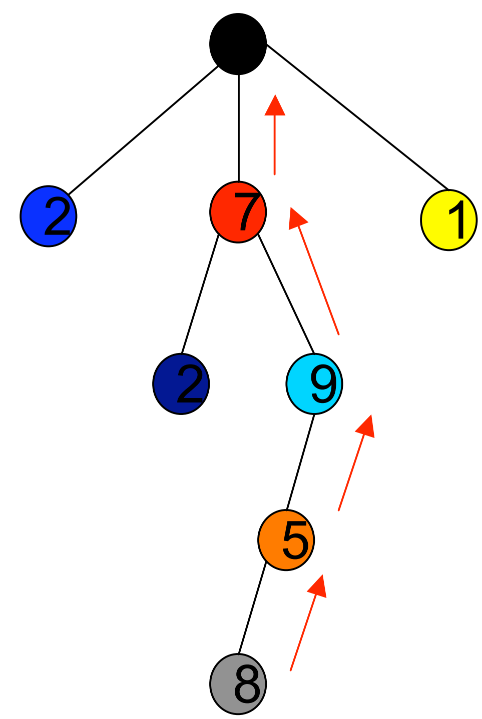
Containment tree:

The parent of a segment *v* is the smallest segment containing *v*



Nested Intervals





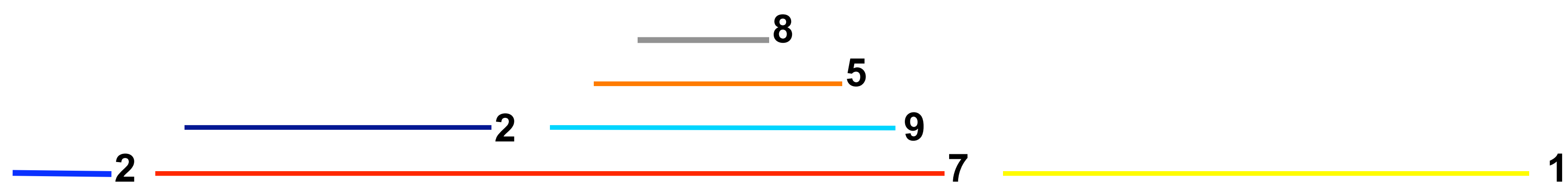
Query:

Starting node *s* = smallest interval containing the query point

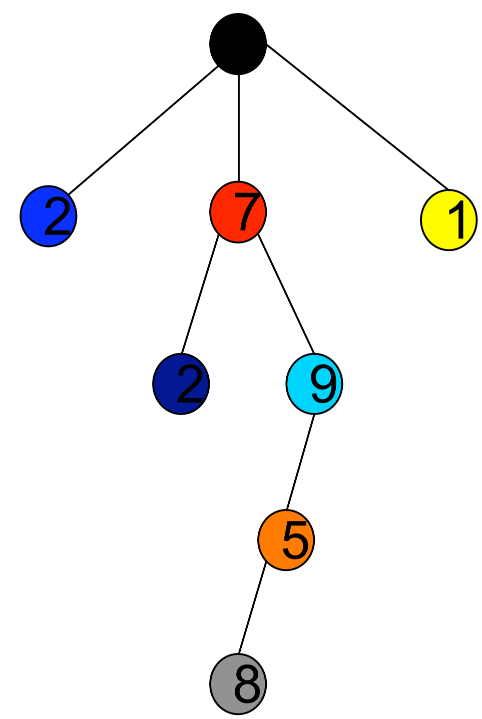
Relevant priorities are on the path from *s* to the root.

Problem: path may be long…

Dynamic trees know how to do that

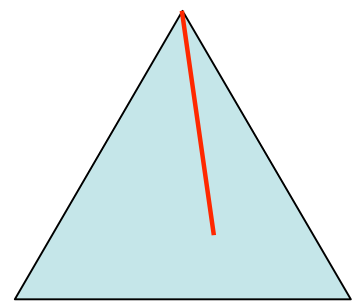


Want to use a dynamic tree to represent the containment tree.

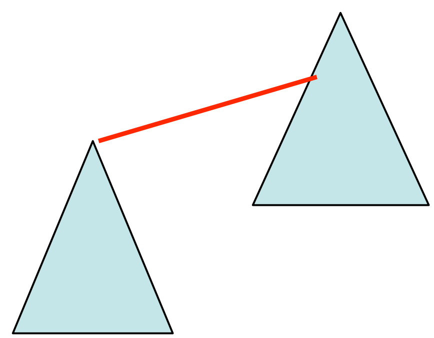


Dynamic trees

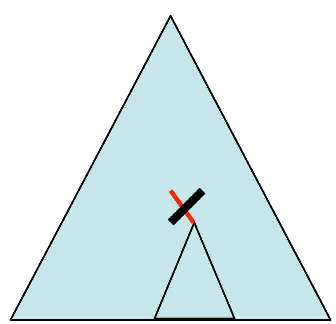
find min along path



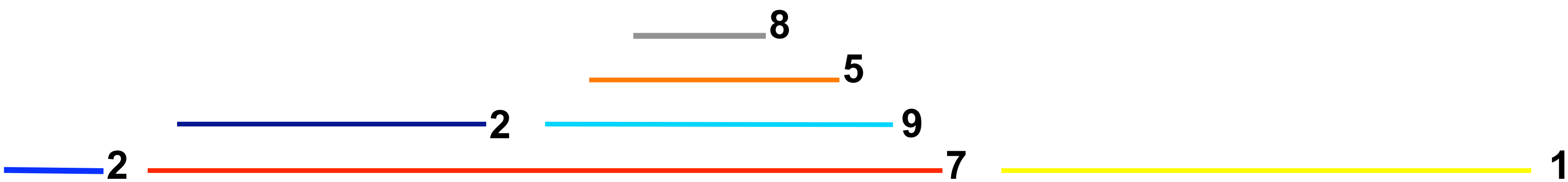
link



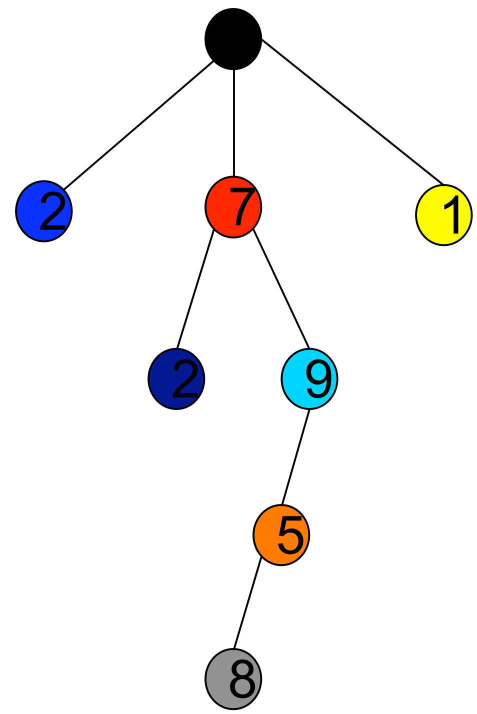
cut



O(log n) time per operation



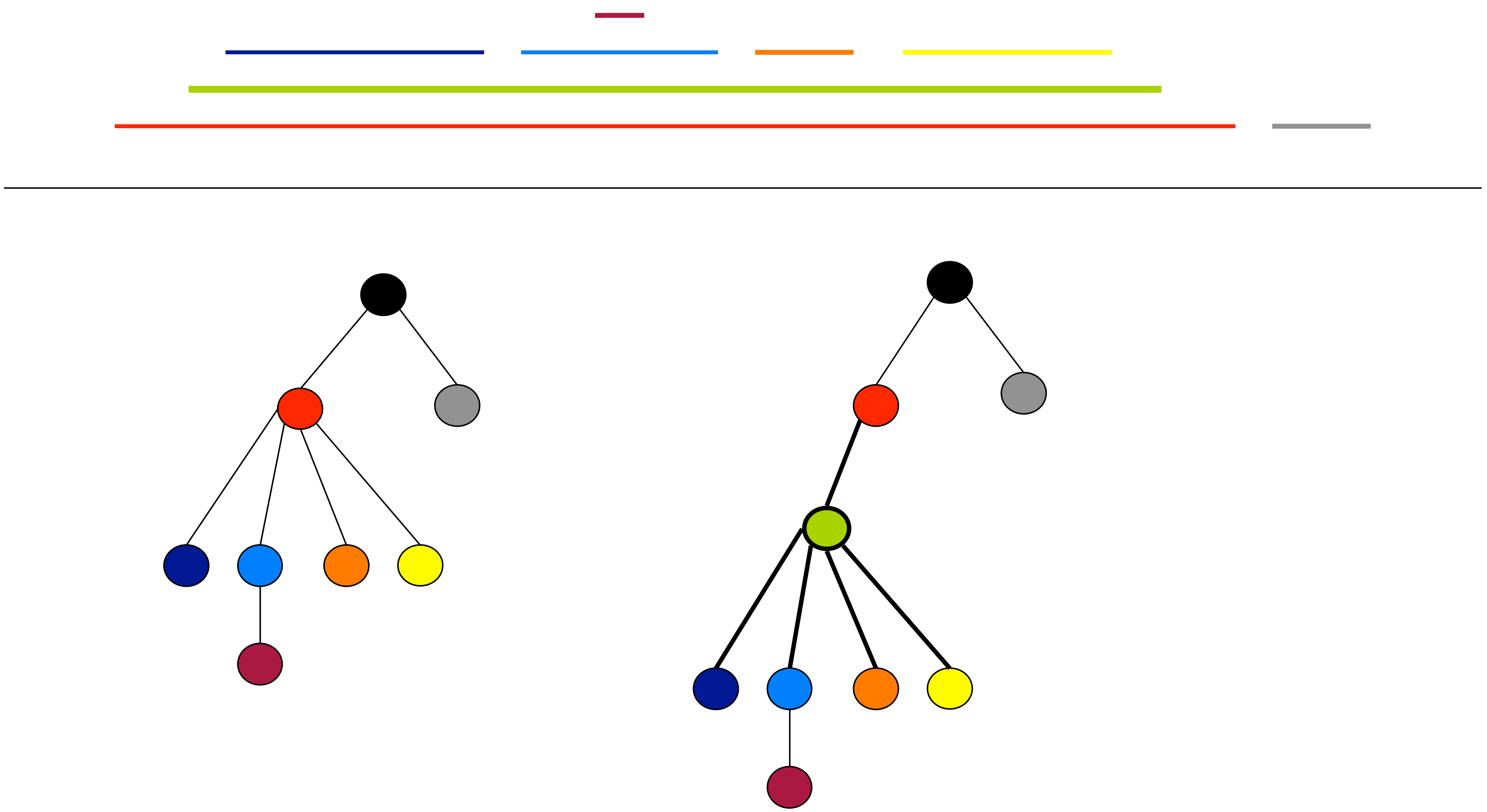
Use a dynamic tree to represent the containment tree



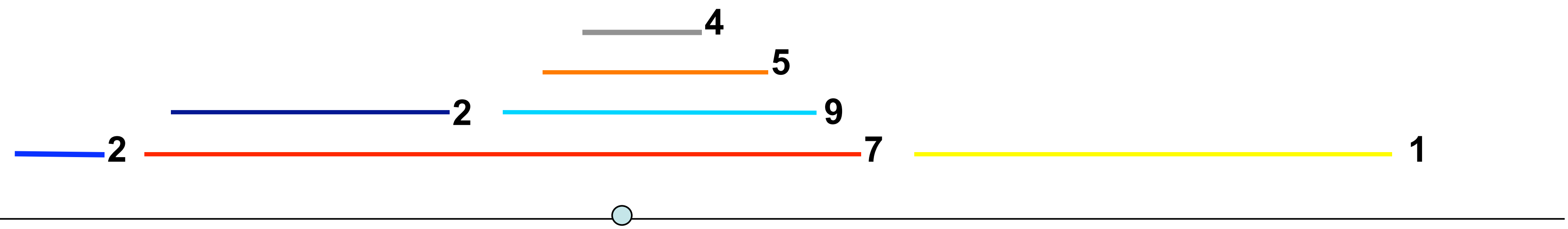
Problem:

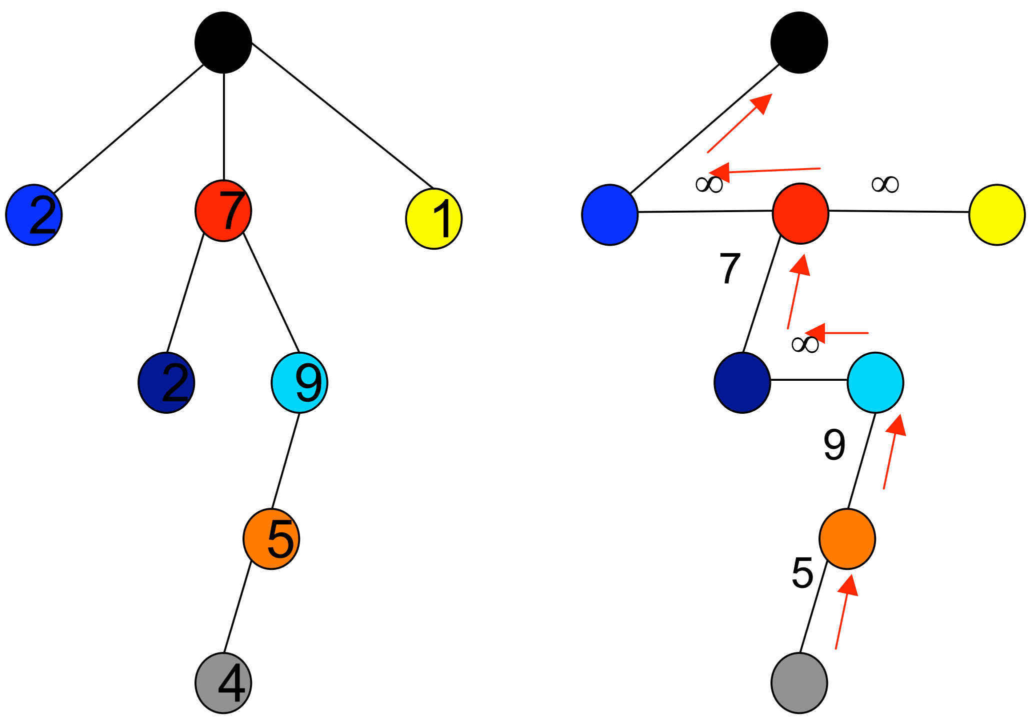
Updates => Many cuts & links

Insert



Binarization





Node v => node v

Leftmost child of v => Left child of v

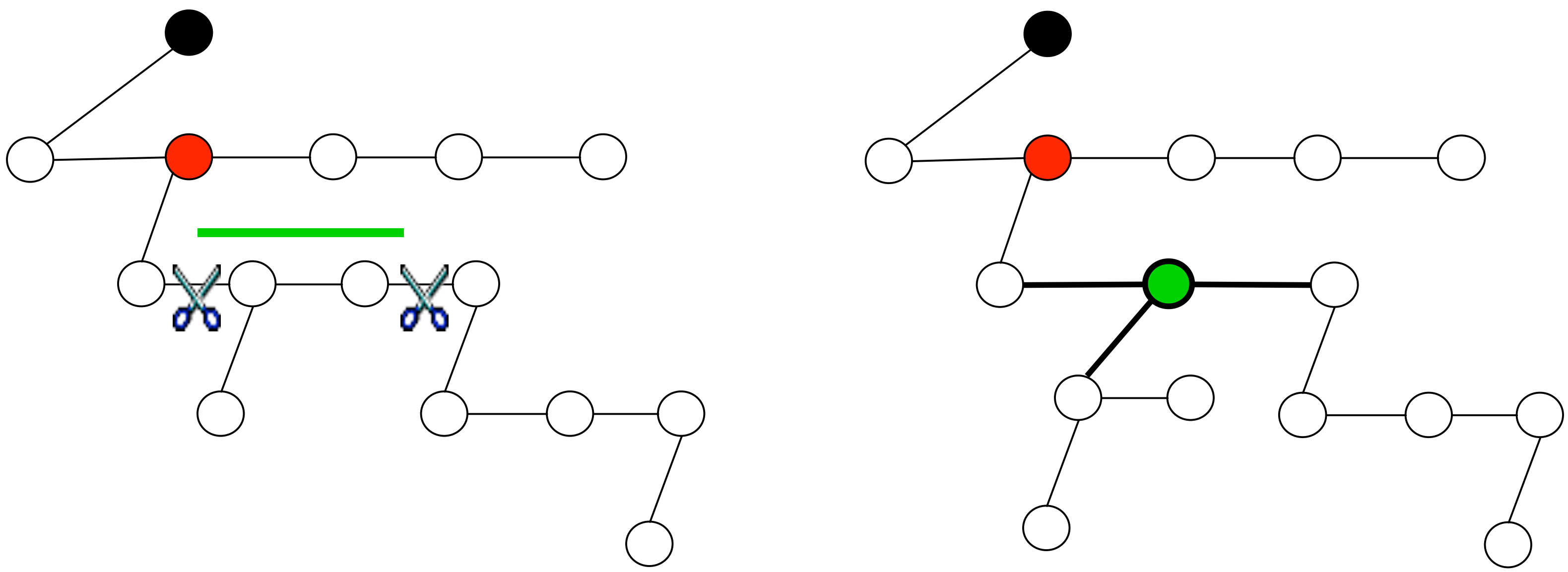
Any other child of v => right child of its left sibling

Adjust costs:

Left edge => priority of parent

Right edge => ∞

Insert (Cont.)

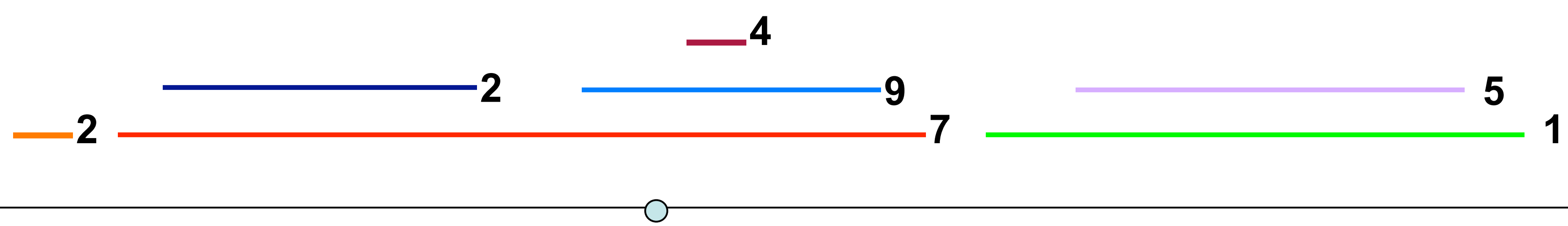


Constant number of links and cuts

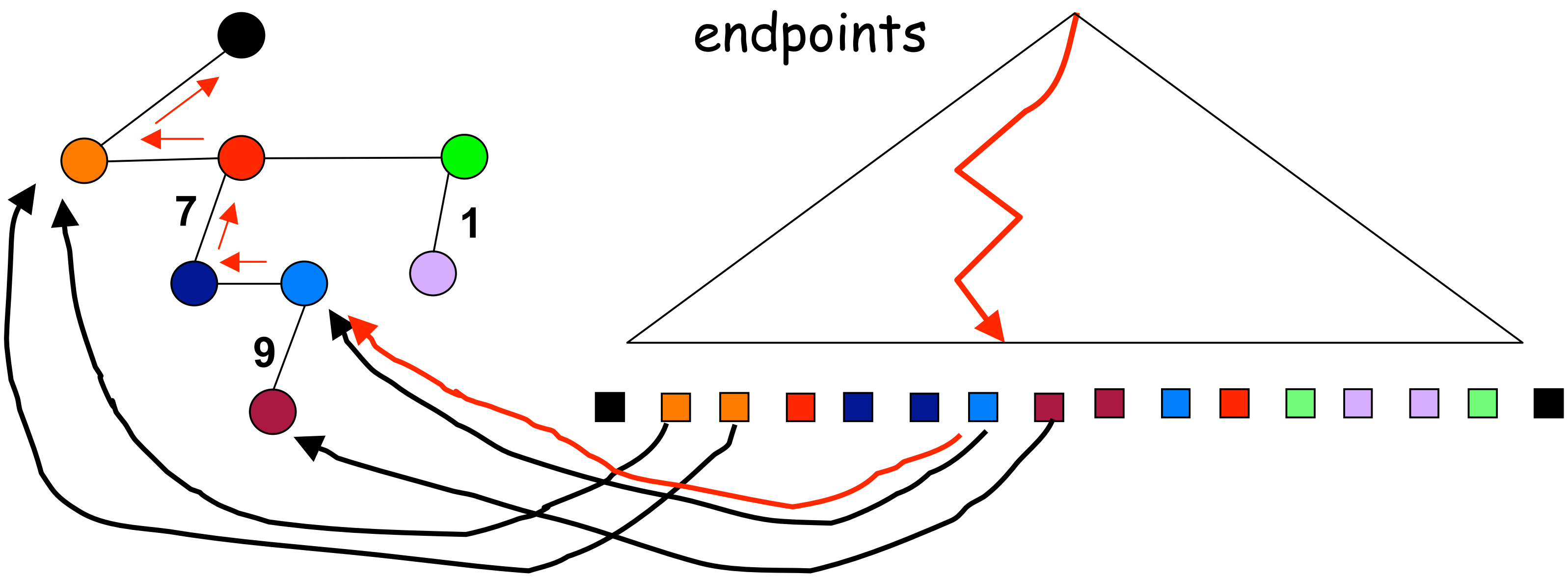
Summary

- Containment tree C
  - Query = min cost on path from starting point to root
- Represent C by binarized version B
- Represent B by dynamic tree D
- How do you find the point to start the query ?
- How do you find the edges to cut ?

How do you start the query ?

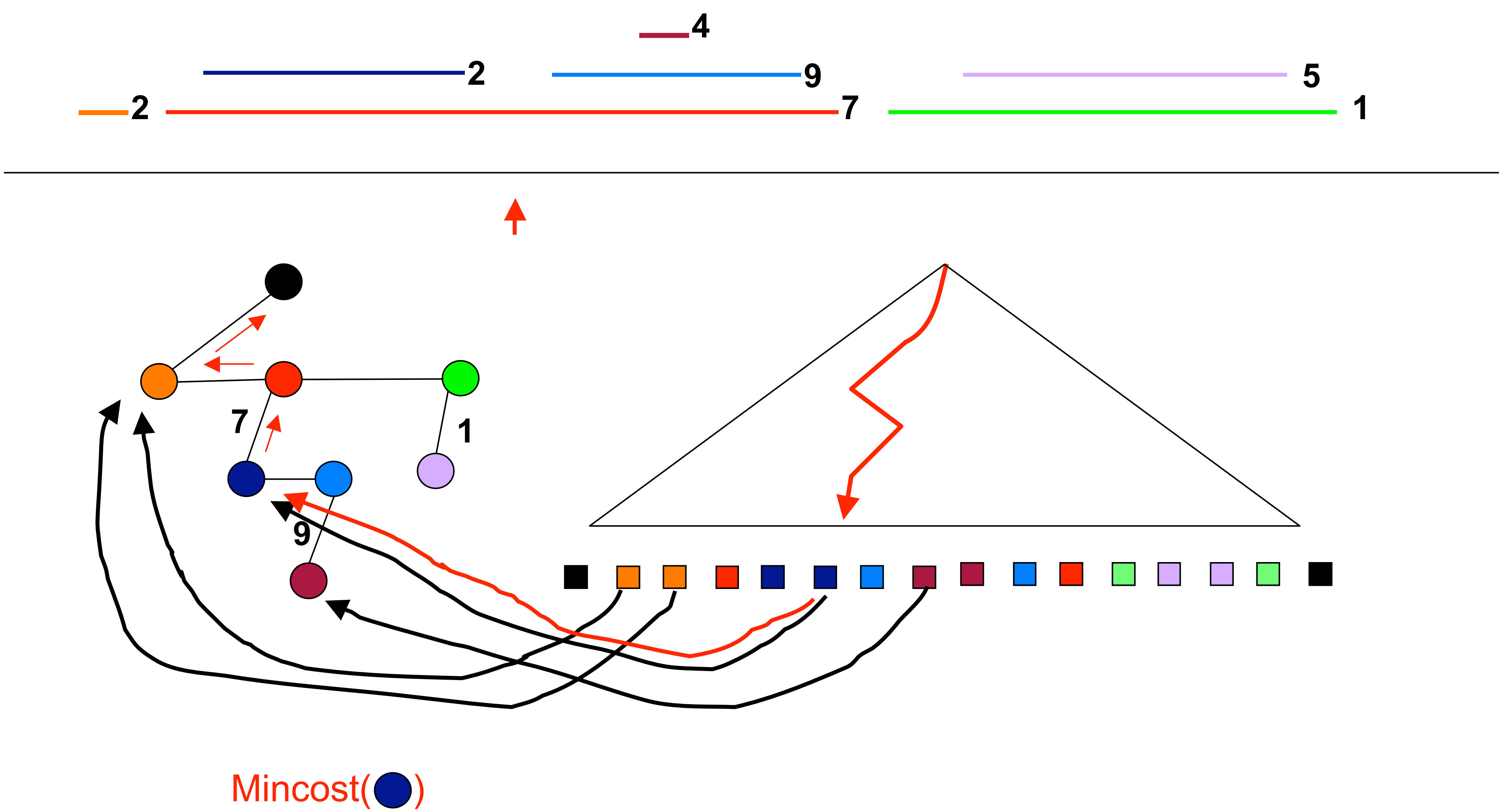


Use a balanced search tree on the endpoints



Min(Mincost( ), pri( ))

query (cont)



Results (2) (STOC 2003,KMT)

- A simple data structure for nested segments with priorities
  O(log(n)) time per op,
  O(n) space (uses dynamic trees)

- A data structure for general segments O(log(n)) time per query/insert but delete takes O(log(n)loglog(n)) time, O(nloglog(n)) space

Results (3) (SODA 2005, AAY)

- A data structure for general segments O(log(n)) time per query/insert but delete takes O(log(n)log~~log~~(n)) time, O(nlog~~log~~(n)) space

- O($\log_B$(n)) I/Os per operation

Further research

- Cache oblivious solution for strings (static solution by Brodal & Fagerberg SODA'06)
- Simplify the solutions
- Implement the shortest segment data structure
- Better solutions for higher dimensions