Augmented Data Structures

- You rarely have to design a data structure from scratch but often need to augment its functionality by providing new operations
- Do this by adding information to the data structure to support the new operations
- The new information needs to be maintained correctly without loss of efficiency for the other operations
- Will illustrate with RB-trees but can do with other data structures: stacks, queues, heaps, etc.

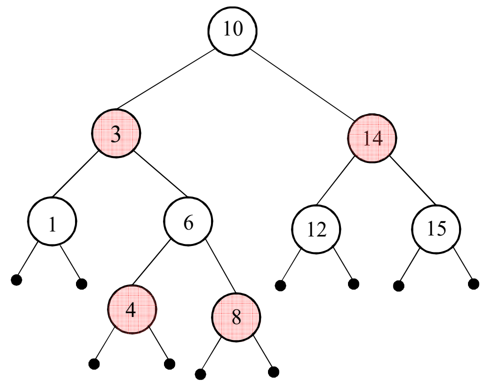
1

Dynamic Order Statistics

- Add the following operations to a RB-tree
  1) Select(x, i)
     returns the i-th smallest element in subtree with root x
  2) Rank(x)
     returns the rank of x within the dynamic set
- Still need to perform Insert, Delete, and the other RB-tree operations efficiently!

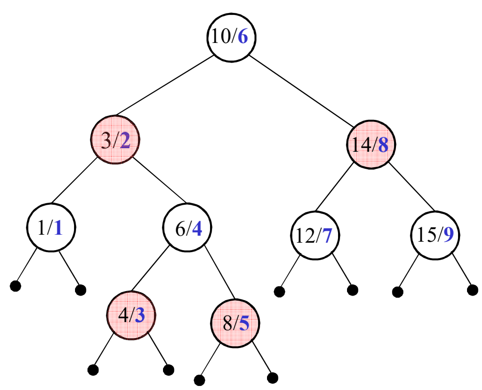
2

Example



3

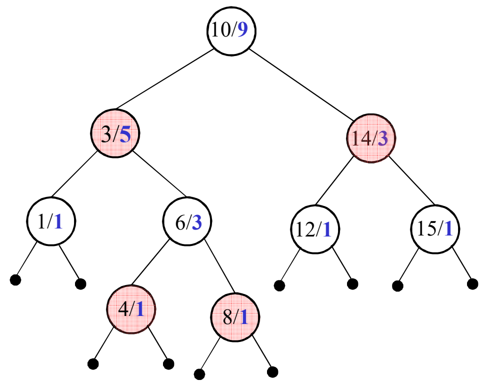
Example: Augment with Rank



- Why doesn’t this work?

4

Example: Augment with Size



- Why?

  Can't compute the rank of *all* nodes but can compute the rank of the root

5

Pseudocode

```
SELECT(x, i)
r ← size[left[x]] + 1
if i = r
    then return x
    else if i < r
            then return SELECT(left[x], i)
            else return SELECT(right[x], i − r)
```

Time?

6

Pseudocode

RANK(T, x)

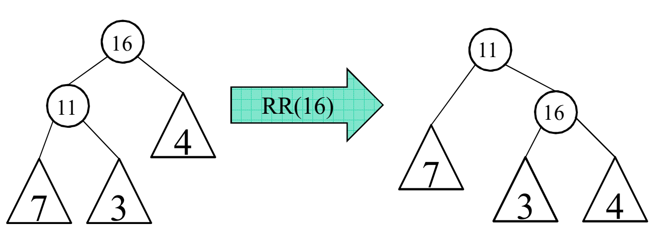
```
r ← size[left[x]] + 1
y ← x
while y ≠ root[T]
        do if y = right[p[y]]
                then r ← r + size[left[p[x]]] + 1
            y ← p[y]
return r
```

Time?

7

Updates

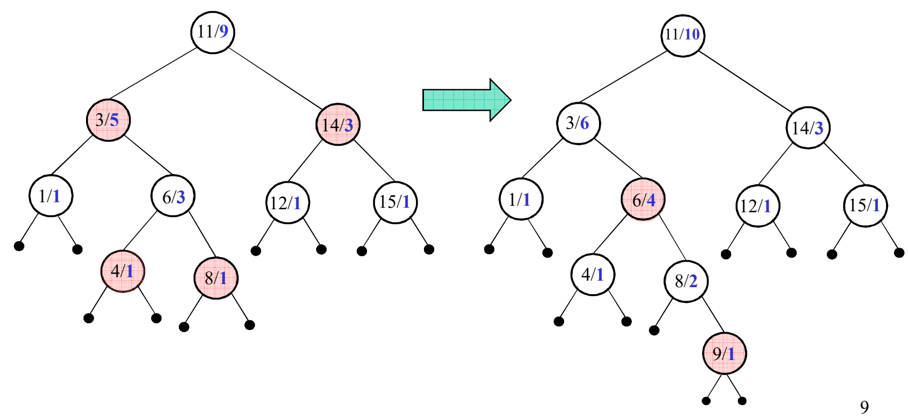
- Can the new information be maintained efficiently in the presence of updates?
  - While traversing the path down for insertion, increment node sizes along the path by 1
  - Color changes cause no problems
  - Must handle rotations



8

Example

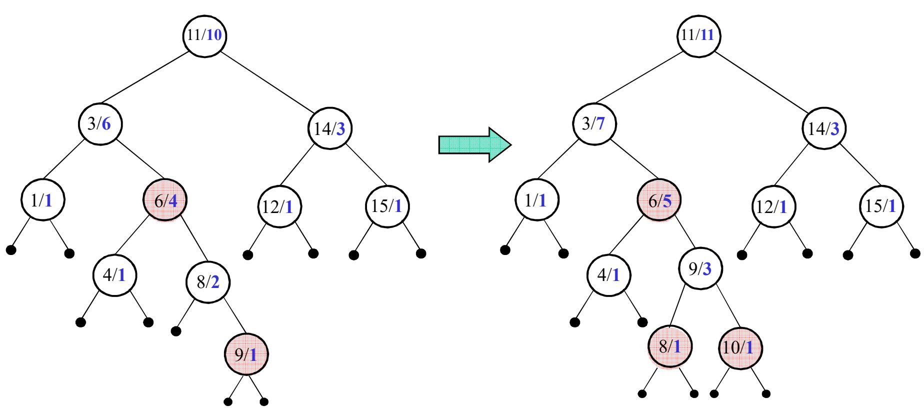
- Insert 9
- Find 7th , 6th , 8th smallest elements
- Insert 10



9

Example…

- Insert 10



10

Methodology Checklist

1. Choose underlying data structure (e.g., RB-tree)
2. Determine which additional information to store and maintain (e.g., subtree sizes)
3. Verify that the new information can be maintained for data structure modifying operations (e.g., insert, delete, rotations)
4. Develop new operations (e.g., Select, Rank)

11

Theorem. Augment a RB-tree with field f, where $f[x]$ can be computed in $O(1)$ time from information in x, *left*[x], and *right*[x] (including f[*left*[x]] and f[*right*[x]]). Then we can maintain the values of f in all nodes during insert and delete without their affecting their $O(\log n)$ performance.

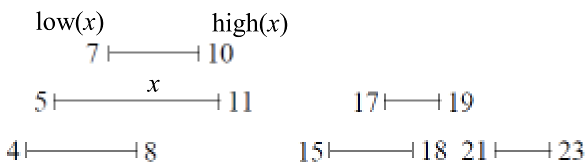
Proof. Since $f[x]$ depends only on x and its children, when we alter information in x, changes propagate only upward (to $p[x]$, $p[p[x]]$, . . . , *T.root*).
$h = O(\log n) \Rightarrow O(\log n)$ updates, at $O(1)$ each.

12

Interval Intersection Search

- Given a collection of intervals I over the real line and a query interval q find *an* interval in I that overlaps q or determine that such interval does not exist.
- Interval $x \in I$ denoted by $x = [\text{low}(x), \text{high}(x)]$

  Example: {[7,10],[5,11],[4,8],[17,19],[15,18],[21,23]}

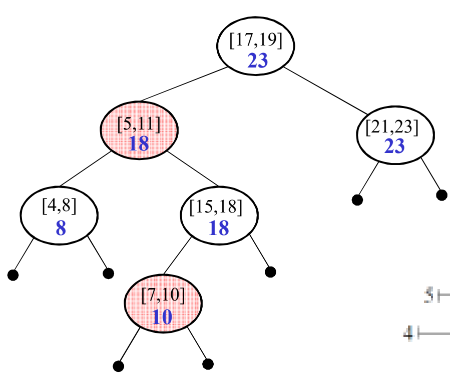


- *Variant*: report all intervals that intersect q

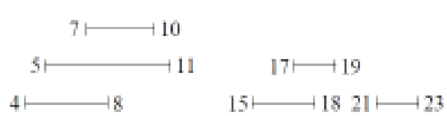
13

Methodology Checklist

1. Select RB-trees as underlying data structure
   - Use low[x], $x \in I$, as keys
2. What information augments the tree?
   - Store in node x the largest high value in subtree rooted at x



$$m[x] = \max \begin{cases} high[x] \\ m[left[x]] \\ m[right[x]] \end{cases}$$

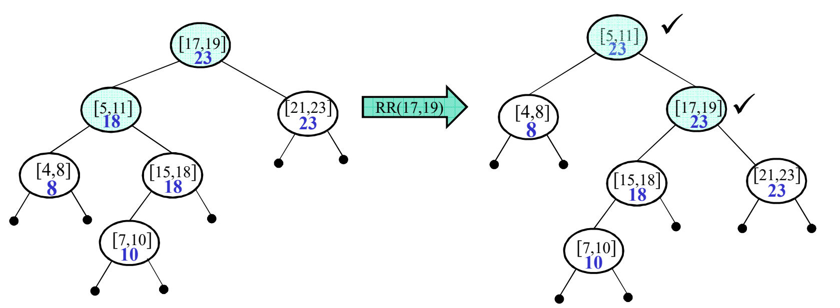


14

Methodology Checklist…

3. Verify that new field can be maintained during updates.

Insertion. fix *m* along the insertion path on the way down .
Delete. Handle three delete cases and walk up the tree to adjust *m*.
Rotations. Recompute *m* locally at two nodes.



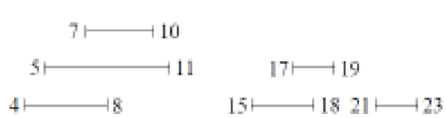
15

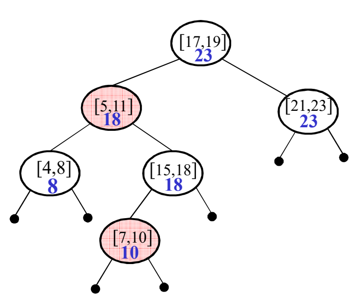
Methodology Checklist…

4. Develop new operations.

SEARCH(T, q)
  $x \leftarrow root[T]$
  while $x \neq nil$ and $(low[q] > high[x]$ or $low[x] > high[q])$
    do if $left[x] \neq nil$ and $low[q] \leq m[left[x]]$
      then $x \leftarrow left[x]$
      else $x \leftarrow right[x]$
  return x

<u>Example</u>: q = [14,16] returns [15,18]
q = [12,14] returns nil





16

Correctness

Theorem. Let $L = \{i \in \text{left}[x]\}$ and $R = \{j \in \text{right}[x]\}$. Then,

a) Going right $\Rightarrow$ $\{i \in L, i \text{ overlaps } q\} = \varnothing$.

b) Going left $\Rightarrow$ $\{i \in L, i \text{ overlaps } q\} = \varnothing \Rightarrow \{i \in \text{right}[x], i \text{ overlaps } q\} = \varnothing$

```
SEARCH(T, q)
  x ← root[T]
  while x ≠ nil and (low[q] > high[x] or low[x] > high[q])
      do if left[x] ≠ nil and low[q] ≤ m[left[x]]
            then x ← left[x]
            else  x ← right[x]
  return x
```

17