

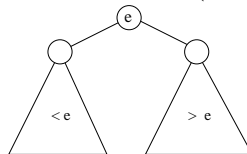
Augmented Search Trees

(CLRS 14)

1 Red-Black Trees

- Last time we discussed red-black trees:

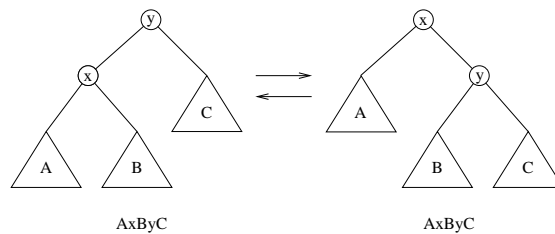
- Balanced binary trees—all elements in left (right) subtree of node x are $< x$ ($> x$).



- Every node is colored RED or BLACK and we maintained red-blue invariant:

- * Root is BLACK.
- * A RED node can only have BLACK children.
- * Every path from the root to a leaf contains the same number of BLACK nodes.

- We saw how the red-blue invariant guaranteed $O(\log n)$ height.
- We could reestablish the red-blue invariant after an insertion or deletion in $O(\log n)$ time
 - $O(\log n)$ node recolorings (no structural changes).
 - $O(1)$ rotations:

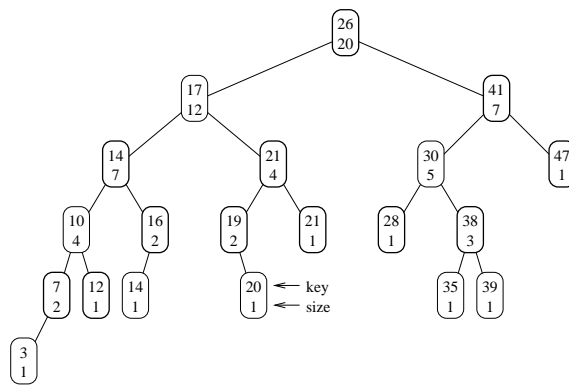


- Red-black tree also supports SEARCH, SUCCESSOR, and PREDECESSOR in $O(\log n)$ as in binary search trees.
- We will now discuss how to develop data structures supporting other operations by *augmenting* red-black tree.

2 Augmented Data Structures

- We want to add an operation $\text{SELECT}(i)$ to a red-black tree
 - We have previously seen how to select the i 'th element among n elements in $O(n)$ time.
 - Can we support it faster if we have the elements stored in a data structure?
 - We can of course support the operation in $O(1)$ time if we have the elements sorted in an array but what if we also want to be able to insert and delete elements?
- We augment every node x in red-black tree with a field $\text{size}(x)$ equal to the number of nodes in the subtree rooted in x
 - $\text{size}(x) = \text{size}(\text{left}(x)) + \text{size}(\text{right}(x)) + 1$

Example:

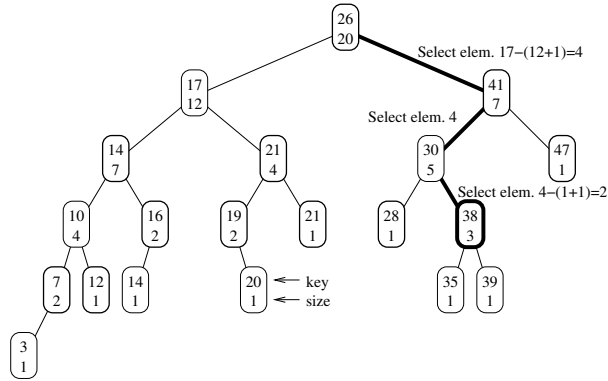


- We can use this field to implement $\text{SELECT}(i)$:

```

SELECT( $x, i$ )
   $r = \text{size}(\text{left}(x)) + 1$ 
  IF  $i = r$  THEN Return  $x$ 
  IF  $i < r$  THEN Return Select( $\text{left}(x), i$ )
  IF  $i > r$  THEN Return Select( $\text{right}(x), i - r$ )
  
```

Example ($\text{SELECT}(17)$):



↓

Since we only follow one root-leaf path the operation takes $O(\log n)$ time.

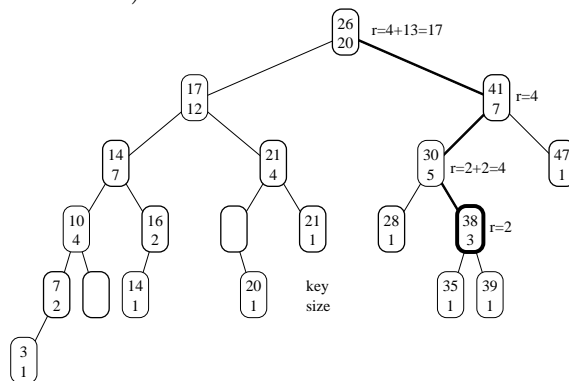
- Actually, we can also use the field to perform the “opposite” operation in $O(\log n)$ time—determining the *rank* of the element in node x :

```

RANK( $x$ )
   $r = size(left(x)) + 1$ 
   $y = x$ 
  WHILE  $y \neq$  root of tree DO
    IF  $y = right(parent(y))$  THEN
       $r = r + size(left(parent(y))) + 1$ 
       $y = parent(y)$ 
    FI
  OD
  Return  $r$ 

```

Example (RANK of element 38):

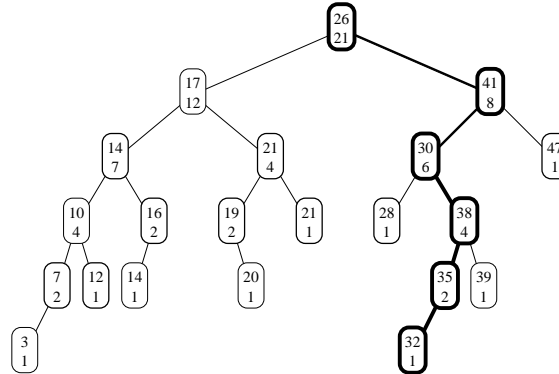


- We need to maintain the extra field during updates:

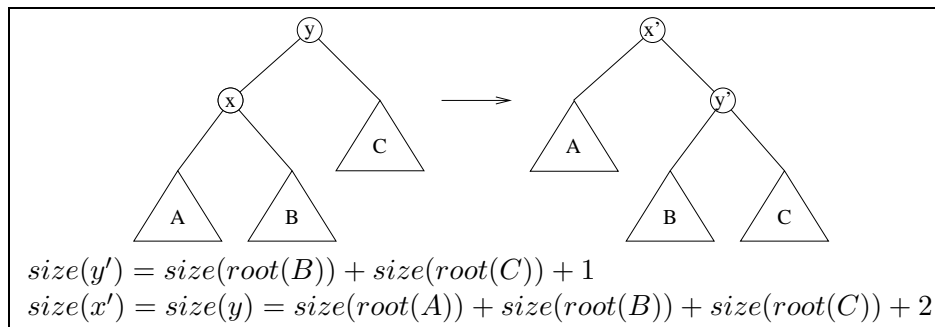
– INSERT(i):

- * Search down one root-leaf part as usual for position where i should be inserted.
- * Increment $size(x)$ for all nodes x on root-leaf path (these are the *only* nodes for which the size field change).

Example (Insertion of element 32)



- * Rebalancing using Red-black tree rules—recall that we do $O(\log n)$ recolorings and $O(1)$ rotations:
 - Color change rules do not affect extra field
 - Rotations do affect size extra fields but we can still easily perform a rotation in $O(1)$ time



\Downarrow
 INSERT performed in $O(\log n)$ time.

– DELETE(i):

- * Find element to delete and decrement size field on one root-leaf path (recall that conceptually we always delete a node with at most one child).
- * Rebalance using rotations.

\Downarrow
 DELETE performed in $O(\log n)$ time.

- Note: The key to maintaining the size field during updates is that the field of node x only depend on the field of the children of $x \Rightarrow$

– Insertion or deletion only affect one root-leaf path.

- Rotations can be handled in $O(1)$ time locally.
- In general we can easily prove the following:

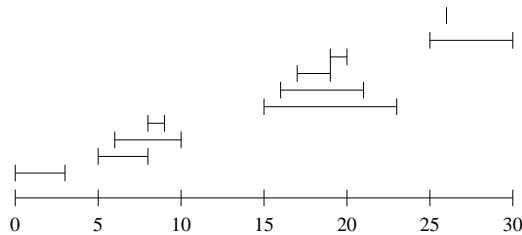
A field f in a red-black tree can be maintained in $O(\log n)$ time during updates if $f(x)$ can be computed using only information in x , $left(x)$ and $right(x)$ (including $f(left(x))$ and $f(right(x))$)

- When changing field in a node x , f can only change for the $O(\log n)$ ancestors of x on the path to the root.
- Rotations can be handled in $O(1)$ time locally.

3 Interval Tree

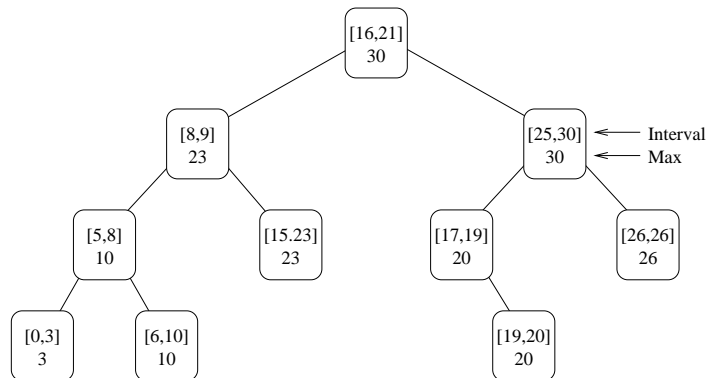
- We now consider a slightly more complicated augmentation. We want so solve the following problem:
 - Maintain a set of n intervals $[i_1, i_2]$ such that one of the intervals containing a query point q (if any) can be found efficiently.

Example: A set of intervals. A query with $q = 9$ returns $[6, 10]$ or $[8, 9]$. A query with $q = 23$ returns $[15, 23]$.

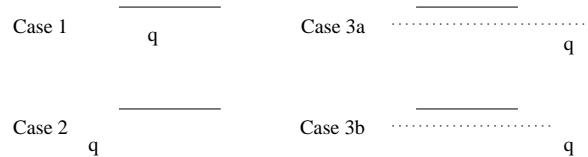


- To solve the problem we use the so-called “Interval tree”:
 - Red-black tree with intervals in nodes
 - * Key is left endpoint
 - Node x augmented with maximal right endpoint in subtree rooted in x

Example: Interval tree on intervals from previous figure:



- We can maintain the interval tree dynamically during insertions and deletions in $O(\log n)$ time
 - because augmented field in x only depends on augmented fields in the children of x and the interval stored in x .
 - $max(x) = \max(rightendpoint(x), max(left(x)), max(right(x)))$
- We can also answer a query in $O(\log n)$ time:
 1. We first check if q is contained in interval stored in root r —if it is we are done.
 2. Next we check if q is on left side of left endpoint of interval in r —if it is we recursively search in left subtree (q cannot be contained in any interval in right subtree).
 3. If q is to the right of left endpoint of interval in r we have two cases:
 - (a) If $max(left(r)) > q$ there must be a segment in left subtree containing q and we recurse left.
 - (b) If $max(left(r)) < q$ there is no segment in left subtree containing q and we recurse right.



```

QUERY( $x, q$ )
  IF  $q$  contained in  $x$  interval THEN Return  $x$ 
  IF  $max(left(x)) \geq q$  THEN
    Return Query( $left(x), q$ )
  ELSE
    Return Query( $right(x), q$ )
  FI

```

↓

We search down one root-leaf path $\Rightarrow O(\log n)$ time.

Example: Query with $q = 23$:

